Protocol-Guided Analysis of Post-silicon Traces Under Limited Observability

***Abstract*—We consider the problem of reconstructing system-level behavior of an SoC design from a partially observed signal trace. Solving this problem is a critical activity in post-silicon validation, and currently depends primarily on human creativity and insights. In this paper, we provide an algorithm to automatically infer system-level transactions from incomplete, ambiguous, and noisy trace data. We demonstrate the approach on a multicore virtual platoform developed within the GEM5 environment.**

I. Introduction

Post-silicon validation makes use of pre-production silicon integrated circuit (IC) to ensure that the fabricated system works as desired under actual operating conditions with real software. Since the silicon executes at target clock speed, post-silicon executions are billions of times faster than RTL simulations, and even provide speed-up of several orders of magnitude over other pre-silicon platforms (*e.g.*, FPGA, system-level emulation, etc.). This makes it possible to explore deep design states which cannot be exercised in pre-silicon, and identify errors missed during pre-silicon validation and debug. Post-silicon validation is a critical component of the design validation life-cycle for modern microprocessors and SoC designs. Unfortunately, it is also a highly complex component, performed under aggressive schedules and accounting for more than 50% of the overall design validation cost. Consequently, it is crucial to develop techniques for streamlining and automating post-silicon validation activities.

A key component of post-silicon validation of SoC designs is to correlate traces from silicon execution with the intended system-level transactions. An SoC design is typically composed of a large number of pre-designed hardware or software blocks (often referred to as "intellectual properties" or "IPs") that coordinate through complex protocols to implement the system-level behavior. Any execution trace of the system involves a large number of interleaved instances of these protocols. For example, consider a smartphone executing a usage scenario where the end-user browses the Web while listening to music and sending and receiving occasional text messages. Typical post-silicon validation use-cases involve exercising such scenarios. An execution trace would involve activities from the CPU, audio controller, display controller, wireless radio antenna, etc., reflecting the interleaved execution of several communication protocols. On the other hand, due to observability limitations, only a small number of participating signals can be actually traced during silicon execution. Furthermore, due to electrical perturbations, silicon data can be noisy, lossy, and ambiguous. Consequently, it is non-trivial to identify all participating protocols and pinpoint the interleaving that results in an observed trace.

In this paper, we consider the problem of reconstructing protocol-level behavior from silicon traces in SoC designs. Given a collection of system-level communication protocols and a trace of (partially observed) hardware signals, our approach infers, with a certain measure of confidence, the protocol instances (and their interleavings) being exercised by the trace. The approach is based on a formalization of system-level transactions via labeled Petri-Nets, which are capable of describing sequencing, concurrency, and choices over system events. We develop algorithms to infer system-level transactions from traces with missing, noisy, and ambiguous signal values. We demonstrate our approach on a multicore virtual platform constructed within the GEM5 environment [1].

II. Background

A. SoC Protocols and Post-silicon Trace Analysis

An SoC design involves integration of a number of IPs that communicate through complex protocols. Such system-level protocols are typically specified in architecture documents as message flow diagrams. As illustration, Fig 1(a) shows one such diagram for a protcol to authenticate and load a firmware during system boot for firmware upgrade. During validation, the system under debug (SUD) exercises some complex system-level use-case which involves interleaved execution of possibly a large number of such protocols. A trace of a small number of hardware signals is then shipped off-chip analysis. The off-chip analysis includes two broad phases: (1) trace abstraction, and (2) trace interpretation. Trace abstraction maps the hardware trace into higher-level architectural constructs, *e.g.*, messages, operations, etc.: a message such as `Authorization request` may be implemented in hardware through a Boolean or temporal combination of specific hardware signals in the NoC fabric between `Device` and `CE`, *e.g.*, as a sequence containing a header, a specific value of a sequence of data words, etc. We will refer to such architectural constructs as *protocol events* or *flow events*. Note that due to limited observability, it may not be possible to map a given set of (observed) hardware signals uniquely to a flow event. Finally, the trace may a result from several instances of the same protocol executing concurrently, *e.g.*, a firmware authentication protocol may be invoked when another instance of the protocol has not completed.

Trace interpretation entails mapping flow events created during trace abstraction to system-level protocols in order to identify the set of protocol instances (and interleavings) responsible for creating the observed behavior. The trace may identify a problem in the protocols themselves, *e.g.* an interleaving of some protocol executions may lead to an unexpected message being sent or cause the system to crash. More commonly, one finds a bug in the *implementation* of

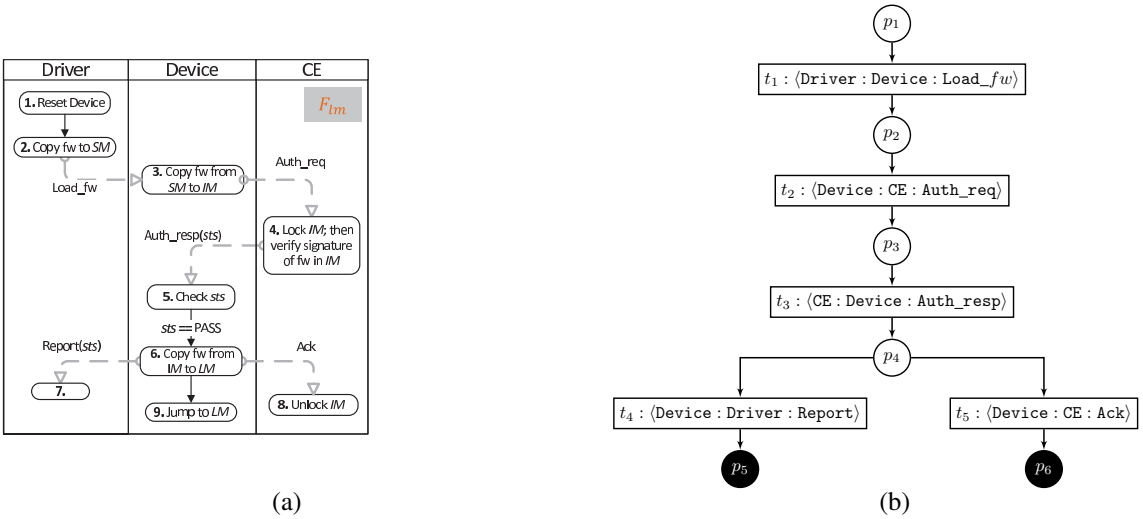

Fig. 1. (a) A graphical representation of a SoC firmware load protocol [2]. (b) LPN formalization. Each event has a form of $\langle \texttt{src}, \texttt{dest}, \texttt{cmd} \rangle$ where `cmd` is a command sent from a source component `src` to a destination component `dest`. The solid black places without outgoing edges are *terminals*, which indicate termination of protocols represented by the LPNs.

the protocol, *i.e.*, a trace inconsistent with any possible interleaving of the protocol executions. Identifying these problems involves significant human expertise, and can often take days to weeks of effort.

B. Labeled Petri-Nets

Labeled Petri-nets (LPN) is a formalization of state transition systems that is capable of describing sequencing, concurrency, and choices. Fig. 1(b) illustrates how to use LPN to formalize protocols. Formally, an LPN is a tuple (P, T, s_0, E, L) where P is a finite set of *places*, T is a finite set of *transitions*, $init$ is the set of initially marked places, also referred to as the *initial marking*, E is a finite set of *events*, and $L : T \to E$ is a labeling function that maps each transition $t \in T$ to an event $e \in E$. For each transition $t \in T$, its preset, denoted as $\bullet t \subseteq P$, is the set of places connected to t, and its postset, denoted as $t\bullet \subseteq P$, is the set of places that t is connected to. A marking $s \subseteq P$ of a LPN is a subset of places marked with tokens, and it is also referred to as a state of a LPN. The initial marking $init$ is also the initial state of the LPN.

III. Flow-Directed Trace Interpretation

In this section we formalize the trace interpretation problem in terms of labeled Petri-nets, and discuss our algorithms to address the problem. For pedagogical reasons, here we assume full observability of all hardware signals involved in the flow events. In the next section we will extend the approach to partial observability.

Notations and formalization. We assume that the set of system flows is provided as a collection $\vec{F}$ of LPNs. We define a *flow execution scenario* to be a set $\{(F_{i,j}, s_{i,j})\}$ where $F_{i,j}$ is the jth instance of flow $F_i \in \vec{F}$, and $s_{i,j}$ is a state of $F_{i,j}$. A flow execution scenario indicates the set of protocols and the number of instances of a particular protocol are activated and their corresponding current states. Since we assume full observability, we view an *observed trace* $\rho = e_1 e_2 \dots e_n$ is a sequence of events. Given an observed trace ρ, the goal of of trace interpretation is to construct a set of candidate flow execution scenarios whose execution can create the sequence of events in ρ. We call such execution scenarios *compliant* with ρ. Let $accept(F_{i,j}, s_{i,j}, e)$ be a function that determines if event e can be emitted by $F_{i,j}$ in state $s_{i,j}$. Formally, $accept(F_{i,j}, s_{i,j}, e)$ returns $(F_{i,j}, s'_{i,j})$ where $s'_{i,j} = (s_{i,j} - \bullet t) \cup t\bullet$ if there exists a transition t in F_i such that $L(t) = e$ and $\bullet t \subseteq s_{i,j}$. It returns $\emptyset$ if no such t exists in F_i.

Given an obsered trace ρ and the set $\vec{F}$ of LPNs, Algorithm 1 provides a basic procedure for computing a set of compliant flow execution scenarios. The algorithm operates by keeping track (in variable $Scen$) of a set of candidate flow execution scenarios compliant with each prefix of ρ. At each iteration, for each event e_h in the observed trace, we update $Scen$ by either extending a member of $scen$ or initiating a new protocol instance for each $scen \in Scen$ with respect to e_h in every possible way. If e_h cannot be emitted by any existing or new flow instances, then we report that the trace is `inconsistent`, *i.e.*, there is no possible interleaving of the protocol instances from $\vec{F}$ that is compliant with ρ.

To illustrate the basic idea of the trace analysis method, consider the system flow shown in Figure Fig. 1(b). Let F_1 denote such flow. Suppose that a hardware system implements flow F_1, and the following flow trace is abstracted from an observed signal trace as a result of executing such system.

$$t_1\ t_2\ t_1\ t_2\ t_3\ t_3\ t_4\ t_5\ t_5\ t_4 \dots$$

where transition names in the LPN are used to represent the flow events in the trace. The first four events results in the

Algorithm 1: CHECK-COMPLIANCE($\vec{F}$, ρ)

Create an empty scenario $scen$
$Scen = \{scen\}$
foreach h, $1 \leq h \leq n$ **do**
  $found \leftarrow$ `true`
  $Scen' = \emptyset$
  foreach $scen \in Scen$ **do**
    foreach $(F_{i,j}, s_{i,j}) \in scen_1$ **do**
      if $accept(F_{i,j}, s_{i,j}, e_h) = (F_{i,j}, s'_{i,j})$ **then**
        Let $scen'$ be a copy of $scen$
        $scen' \leftarrow (scen' - (F_{i,j}, s_{i,j})) \cup (F_{i,j}, s'_{i,j})$
        $Scen' \leftarrow scen' \cup Scen'$
        $found \leftarrow$ `false`
    foreach $F_i \in \vec{F}$ **do**
      create a new instance $F_{i,j+1}$
      if $accept(F_{i,j+1}, init_{i,j+1}, e_h) =$
      $(F_{i,j+1}, s'_{i,j+1})$ **then**
        Let $scen'$ be a copy of $scen$
        $scen' \leftarrow scen' \cup (F_{i,j+1}, s'_{i,j+1})$
        $Scen' \leftarrow scen' \cup Scen'$
        $found \leftarrow$ `false`
  if $found ==$ `true` **then**
    return `Inconsistent`
  $Scen = Scen'$
return $scen$

following flow execution scenario

$$\{(F_{1,1}, \{p_3\}),\ (F_{1,2}, \{p_3\})\}.$$

For the first event t_3, it results in two execution scenarios below depending on which flow instance emits t_3.

$$\{(F_{1,1}, \{p_4\}),\ (F_{1,2}, \{p_3\})\}$$
$$\{(F_{1,1}, \{p_3\}),\ (F_{1,2}, \{p_4\})\}.$$

After handing the next event t_3, the above two execution scenarios are reduced to the one as shown below.

$$\{(F_{1,1}, \{p_4\}),\ (F_{1,2}, \{p_4\})\}.$$

By following Algorithm 1 to handle the remaining four events, the following execution scenario can be derived.

$$\{(F_{1,1}, \{p_5, p_6\}),\ (F_{1,2}, \{p_5, p_6\})\}$$

IV. TRACE ANALYSIS WITH PARTIAL OBSERVABILITY

The trace interpretation discussion in the preceding section assumed full observability. In this section, we discuss how to adapt the method to deal with silicon traces under limited (partial) obsrvability.

In general, a signal trace of partial observability corresponding a set of traces of flow events due to the ambiguous interpretation of signal events. In the following, we discuss two cases for trace abstraction on partial observability: mapping a single signal event to a flow event or mapping a sequence of signal events to a flow event. A signal event is defined as a state on or an assignment to a set of signals.

Hereafter, the term *flow traces* is used to refer to traces of flow events. Consider the following example for the first case. Suppose that there are three flow events: e_1, e_2, and e_3, which are implemented in hardware by the signal events shown in the list below. We use Boolean expressions to represent signal events for the discussion.

$$\begin{aligned} e_1 &: \ abc \\ e_2 &: \ \bar{a}bc \\ e_3 &: \ a\bar{b}c \end{aligned}$$

Now suppose that only signals b and c are observable, and a signal trace of this partial observability is obtained below.

$$bc\ bc\ \bar{b}c$$

During the trace abstraction step, the first two signal events bc can be mapped to $\{e_1, e_2\}$ since a is not observable, and the last one $b'c$ is mapped to $\{e_3\}$. Therefore, this signal trace is abstracted to four flow traces, $\{e_1, e_2\} \times \{e_1, e_2\} \times \{e_3\}$.

Next, we consider the case where a flow event is mapped from a sequence of signal events. Now suppose that two other flow events are implemented by sequences of signal events as defined in the list below.

$$\begin{aligned} e_4 &: \ abc\ \bar{a}bc \\ e_5 &: \ abc\ abc\ abc\ \bar{a}bc \end{aligned}$$

Given an observed trace of the same observability shown below

$$bc\ bc\ bc\ bc,$$

it is abstracted to the following flow traces.

$$e_4e_4,\ \llcorner e_4 \lrcorner,\ e_4 \lrcorner\lrcorner,\ \lrcorner\lrcorner e_4,\ e_5$$

where $\lrcorner$ denotes signal events that are not mapped to any flow events. Note that the above abstraction leads to three distinct flow traces as the middle three correspond to the same trace of flow events.

From the above discussions, it can be seen that a signal trace of partial observability is generally mapped to a set of flow traces. As the the number of signals available for observation becomes smaller, the number of flow event traces corresponding to a signal trace as a result of the abstraction can be enormous. This phenomenon can increase the complexity of the trace interpretation as it can cause the number of possible flow execution scenarios generated during the analysis to explode. Possible solutions to address this issue include better trace signal selection for observation and assistance from debuggers' insights. Trace signal selection itself is an important and difficult subject, and a detailed discussion of it is out of scope of this paper. Next, we briefly describe how the debuggers' insights of a system's architecture can help to address the complexity issue in the trace analysis.

A. Inputs from Validators

During the trace interpretation, the number of intermediate flow execution scenarios may become too large due to ambiguous interpretation from signal events to flow events or

from flow events to flow execution scenarios. The explosion of the intermediate results can significantly slow down the performance of the trace analysis. To address this problem, the debuggers can use their insights and understanding of the SUD to trim the total number of possible system executions derived from the trace analysis. When a SUD is validated, a debugger is assumed to have deep knowledge about the system's architecture and microarchitecture, and how the test environment affects the system executions. For instance, he/she may have knowledge that in a test environment, the maximal number of instances of a flow can be activated by the SUD, or a flow can be activated after certain other flows have terminated, etc. These debugging insights can be encoded as constraints into the trace analysis method, which can then be used to eliminate a large number of flow execution scenarios that violate these constraints during the trace interpretation step.

This approach can be flexible in that it allows a debugger to analyze the observed traces in a trail-and-error manner if the precise knowledge of the system (micro-)architecture is hard to come by. For instance, the debugger might initially make a very restricted assumption on how the SUD executes a flow specification, and these assumptions can potentially lead to an empty set of flow execution scenarios. Depending on which of these assumptions triggered during the trace interpretation step, the debugger can study these assumptions more carefully, and relax some or all of them for the next run of analysis. This iteration can be repeated as many times as necessary until some results deemed meaningful are produced.

Alternatively, if all derived execution scenarios seem to be plausible, the implication that a debugger may draw from this result is that the failure may be independent of the flows being observed. Therefore, the testing environment can be adjusted in order for a different part or different behavior of the SUD to be observed. This idea, closely related to trace signal selection, is critical for post-silicon validation, and a detailed discussion can only be presented in a separate paper.

V. Case Study

The proposed method is demonstrated on a transaction level model of a simple SoC design with CPU core, memory, and a peripheral device connected by a highly concurrent interconnect in SystemC. Since the proposed method is communication centric, the detailed computations of these blocks are not modeled. Instead, the modeling is focused on how they participate in system level protocols. This simple case study shows that the proposed method is capable of providing useful information to help reveal and locate various bugs in the SystemC model, thus making the debugging more efficient. Once a bug-free model is available, the trace analysis method can correctly derive the instances of system flows activated by various components in the design from the observed traces. More details about the design model and system flows implemented by the design model will be provided in a supplement document stored online.

In order to find out the efficiency of the trace analysis method for more realistic examples, in the case study described in this section, a more detailed transaction level model of a SoC is constructed within the GEM5 environment [1]. This SoC model consists of two ARM Cortex-A9 cores, each of which contains two separate 16KB data and instruction caches. The caches are connected to a 1GB memory through a memory bus model. The architecture of this SoC model is shown in Fig. 2.

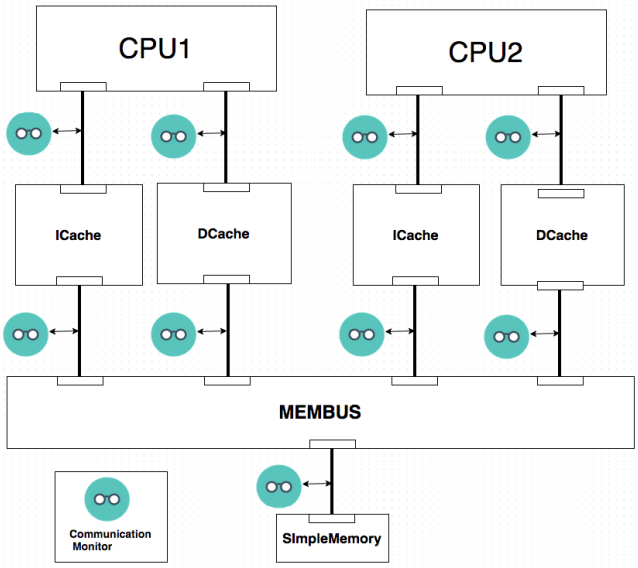

Fig. 2. SoC platform structure.

In this model, components communicate with each other by sending and receiving various request and response messages. In order to observe and trace communications occurring inside this model during execution, monitors are attached to links connecting the components. These monitors record the messages flowing through the links they are attached to, and store them into output trace files.

For this model, we consider the flow specifications describing the cache coherence protocols supported in GEM5 that is used to build the model in Fig 2. These flow specifications describe data/instruction read operations and data write operations initiated from CPUs. Three such flows describe the cache coherent protocols for each CPU. As there are two CPUs in this model, there are actually six such flows considered for this model. These flow specifications are derived from the cache coherence protocols supported in GEM5[1]. More detailed information about the flow specifications used in this case study will be provided as a supplement document if this paper is accepted for publication.

Two simple programs are written, one for each CPU. These simple programs read numbers from a file, perform some operations on these numbers, and store the results back to the file. How GEM5 supports shared memory multi-threaded program execution is unclear. Therefore, there is no data shared in both caches in this test. Furthermore, GEM5 does

[1]The GEM5 cache coherence protocols can be found at http://www.gem5.org/docs/html/gem5MemorySystem.html.

TABLE I
THE NUMBER OF FLOW INSTANCES DERIVED BY THE TRACE ANALYSIS WITH THE FULL OBSERVABILITY.

Flows	#Instances
CPU1 Data Read	17582
CPU1 Instruction Read	4002
CPU1 Write	3370
CPU2 Data Read	17386
CPU2 Instruction Read	3955
CPU2 Write	3308

TABLE II
THE NUMBER OF FLOW INSTANCES DERIVED BY THE TRACE ANALYSIS WITH CERTAIN MONITORS DISABLED.

Flows	#Instances
CPU1 Data Read	829
CPU1 Instruction Read	169
CPU1 Write	82
CPU2 Data Read	803
CPU2 Instruction Read	190
CPU2 Write	83

not support true concurrency. When there are two programs running on the CPUs, GEM5 alternates the executions between the two CPUs. To simulate asynchronous concurrency with the interleaving semantics, those two simple programs are instrumented with pseudo-blocking commands, one placed before each statement. A pseudo blocking command includes a random number generator that returns either 0 or 1 and a loop that only exits when the returned random number is 0.

After the SoC model finishes executing the program, there are totally 343581 messages collected in the trace file. Not all of the messages are relevant to the flow specification as many are used by GEM5 to initialize its simulation environment. After removing those irrelevant messages, the number of messages in the trace file is to reduced to 121138.

The time taken to remove the irrelevant messages from the trace is negligible. The total runtime and the peak memory usage for the trace analysis algorithm to finish the reduced trace are 3 seconds and 12MB, respectively. From the trace, Table I shows the number of instances extracted for the six flows describing cache coherent read/write operations initiated from both CPUs.

To take the partial observability into account, the four monitors attached to the links between two CPUs and their caches are disabled. Then, the trace is generated by the remaining five monitors from the SoC model executing the same program. After removing all non-observable messages, the trace only contains 15089 messages. The numbers of the flow instances extracted by the trace analysis method are shown in Table II. From these results, the numbers of the flow instances are dropped significantly compared to the results extracted from the trace with the full observability as shown in Table I. This difference is due to that some communications occurred in the system when executing the program involve the CPUs and their corresponding caches only, and the traffic on the links between the CPUs and their corresponding caches is not observable. Therefore, the instances of the flow specifications characterizing these communications do not exist in the trace. In other words, all extracted flow instances in Table II characterize the communications that pass through the memory bus in the system model. The runtime and memory usage is similar to that for analyzing the trace of the full observability as shown above.

In the third experiment, further partial observability is taken into consideration. In this experiment, only the five links involving the memory bus are still considered. However, an assumption is made that all events passing the same link are not distinguishable due to the limitation of the observability. More specifically, the monitors are modified such that whenever an event is captured on one of the links, it dumps a set of events passing through the same link into the trace file. Therefore, each line of the trace file corresponds to a set of events. After applying the trace analysis to this trace, a total of 13944 flow execution scenarios are extracted. This large number, compared to the number of extracted execution scenarios shown in the Table I and II, is due to the ambiguous interpretation of the events with limited observability.

The whole experiment takes about 15 minutes and 420 MB to finish, significantly higher than the numbers for analyzing traces where there is no ambiguity in the observed events. This is due to the fact that a trace of ambiguous events is in fact a set of traces of original events, which lead to large numbers of execution scenarios either during or at the end of the analysis. In this experiment, the peak number of execution scenarios during the analysis process is 70384. Compared to the final number, many of the intermediate execution scenarios are invalid, and removed eventually. However, controlling the number of intermediate execution scenarios during the trace analysis is critical in order for the analysis to be tractable. Here, insights from validators can help.

As shown above, the ambiguous interpretation of events can lead to large numbers of intermediate and final execution scenarios, which not only make the trace analysis more time consuming but also make it difficult to gain insightful understanding from the derived execution scenarios. Careful selection of what to observe may have big impact on results from the trace analysis. In this last experiment, we relax the assumption made in the previous experiment such that the events passing each link are partitioned into two groups, one for read operations and one for write operations. Similar to the assumption made in the previous experiment, events in the same group are assumed to be non-distinguishable. The monitors are modified accordingly such that they output all events in the same group into the trace file if an event from that group is captured. After the trace analysis on this new partially observed trace is finished, only one execution scenario is derived where the distribution of the numbers of flow instances

is the same as those shown in Table II. The peak number of execution scenarios encountered during the trace analysis is 4. The total runtime is about 1 second. Compared to the results from the previous experiment, the precision and the performance of the trace analysis are improved dramatically as a result of careful selection of observable signals.

VI. Related Work

Our work is closely related to communication-centric and transaction based debug. An early pioneering work is described in [4], which advocates the focus on observing activities on the interconnect network among IP blocks, and mapping these activities to transactions for better correlation between computations and communications. Therefore, the communication transactions, as a result of software execution, provide an interface between computation and communication, and facilitate system-level debug. This work is extended in [5], [6]. However, this line of work is focused on the network-on-chip (NoC) architecture for interconnect using the run/stop debug control method.

A similar transaction-based debug approach is presented in [7]. Furthermore, it proposes an automated extraction of state machines at transaction level from high level design models. From an observed failure trace, it performs backtracking on this transaction level state machine to derive a set of transaction traces that lead to the observed failure state. In the subsequent step, bounded model checking with the constraints on the internal variables is used to refine the set of transaction traces to remove the infeasible traces. This approach requires user inputs to identify impossible transaction sequences, and may not find the states causing the failure if the transaction traces leading to the observed failure state is long. Backtracking from the observed failure state requires pre-image computation, which can be computationally expensive. A transaction-based online debug approach is proposed in [8] to address these issues. This approach utilizes a transaction debug pattern specification language [9] to define properties that transactions should meet. These transaction properties are checked at runtime by programming debug units in the on-chip debug infrastructure, and the system can be stopped shortly after a violation is detected for any one of those properties. In this sense, it can be viewed as the hardware assertion approaches in [10] elevated to the transaction level.

In [11], a coherent workflow is described where the result from the pre-silicon validation stage can be carried over to the post-silicon stage to improve efficiency and productivity of post-silicon debug. This workflow is centered on a repository of system events and simple transactions defined by architects and IP designers. It spans across a wide spectrum of the post-silicon validation including DFx instrumentation, test generation, coverage, and debug. The DFx instruments are automatically inserted into the design RTL code driven by the defined transactions. This instrumentation is optimized for making a large set of events and transactions observable. Test generation is also optimized to generate only the necessary but sufficient tests to allow all defined transactions to be exercised. Moreover, coverage for post-silicon validation is now defined at the abstract level of events and transactions rather than the raw signals, and thus can be evaluated more efficiently. In [12], a model at an even higher-level of abstraction, *flows*, is proposed. Flows are used to specify more sophisticated cross-IP transactions such as power management, security, etc, and to facilitate reuse of the efforts of the architectural analysis to check HW/SW implementations.

VII. Conclusion

This paper presents a trace analysis based method for post-silicon validation by interpreting observed raw signal traces at the level of system flow specifications. The derived flow execution scenarios provide more structured information on system operations, which is more understandable to system validators. This information can help to locate design defects more easily, and also provides a measurement of validation coverage.

Due to partial observability, this approach may derive a large number of different flow execution scenarios for a given signal trace. Insights from system validators can help to eliminate some false scenarios due to the partial observability. An interesting future direction is formalization of the validators' insights using temporal logic on flows so that the validators can express their intents more precisely and concisely.

The trace analysis approach presented in this paper needs to be iterated with different observations selected in different iterations in order to eliminate the false scenarios and to root cause system failures as quickly as possible. The observation selection and stitching signal traces of different observations together for the above goal will also be pursued in the future.

References

[1] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Comput. Archit. News*, vol. 39, no. 2, pp. 1–7, Aug. 2011.

[2] S. Krstic, J. Yang, D. Palmer, R. Osborne, and E. Talmor, "Security of soc firmware load protocols," in *Hardware-Oriented Security and Trust (HOST), 2014 IEEE International Symposium on*, May 2014, pp. 70–75.

[3] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, no. 4, pp. 541–580, Apr 1989.

[4] K. Goossens, B. Vermeulen, R. v. Steeden, and M. Bennebroek, "Transaction-based communication-centric debug," in *Proceedings of the First International Symposium on Networks-on-Chip*, ser. NOCS '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 95–106.

[5] B. Vermeulen and K. Goossens, "A network-on-chip monitoring infrastructure for communication-centric debug of embedded multi-processor socs," in *VLSI Design, Automation and Test, 2009. VLSI-DAT '09. International Symposium on*, ser. VLSI-DAT '09, 2009, pp. 183–186.

[6] K. Goossens, B. Vermeulen, and A. B. Nejad, "A high-level debug environment for communication-centric debug," in *Proceedings of the Conference on Design, Automation and Test in Europe*, ser. DATE '09. 3001 Leuven, Belgium, Belgium: European Design and Automation Association, 2009, pp. 202–207.

[7] A. M. Gharehbaghi and M. Fujita, "Transaction-based post-silicon debug of many-core system-on-chips," in *ISQED*, 2012, pp. 702–708.

[8] M. Dehbashi and G. Fey, "Transaction-based online debug for noc-based multiprocessor socs," in *Proceedings of the 2014 22Nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, ser. PDP '14. Washington, DC, USA: IEEE Computer Society, 2014, pp. 400–404.

[9] A. M. Gharehbaghi and M. Fujita, "Transaction-based debugging of system-on-chips with patterns," in *Proceedings of the 2009 IEEE International Conference on Computer Design*, ser. ICCD'09. Piscataway, NJ, USA: IEEE Press, 2009, pp. 186–192.

[10] M. Boule, J.-S. Chenard, and Z. Zilic, "Assertion checkers in verification, silicon debug and in-field diagnosis," in *Proceedings of the 8th International Symposium on Quality Electronic Design*, ser. ISQED '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 613–620.

[11] E. Singerman, Y. Abarbanel, and S. Baartmans, "Transaction based pre-to-post silicon validation," in *Proceedings of the 48th Design Automation Conference*, ser. DAC '11. New York, NY, USA: ACM, 2011, pp. 564–568.

[12] Y. Abarbanel, E. Singerman, and M. Y. Vardi, "Validation of soc firmware-hardware flows: Challenges and solution directions," in *Proceedings of the The 51st Annual Design Automation Conference on Design Automation Conference*, ser. DAC '14. New York, NY, USA: ACM, 2014, pp. 2:1–2:4.