End-to-End Formal using Abstractions to Maximize Coverage

(Invited Tutorial)

Prashant Aggarwal
Oski Technology
Gurgaon, India
prashant@oskitech.com

Darrow Chu
Cadence Design Systems
San Jose, CA, USA
darrow@cadence.com

Vijay Kadamby
Cisco
San Jose, CA, USA
vkadamby@cisco.com

Vigyan Singhal
Oski Technology
Mountain View, CA, USA
vigyan@oskitech.com

***Abstract*—Model checking tools are gaining traction as a practical formal verification solution for industrial designs. However, the use of abstraction models is key to overcoming complexity barriers in applying these tools. Coverage has been a useful metric to determine when simulation-based verification is complete. In this paper, we show how similar coverage metrics can be used to determine the completeness of a formal verification setup. We also show how coverage can be used to determine effectiveness of different abstraction models are. This methodology can be used to set formal verification goals, and to measure the progress of the work, thereby placing formal verification in a chip design schedule. We use a real-world design with a large state space, and present quantitative coverage metrics to illustrate the methodology, and its benefits for faster run-time, faster discovery of bugs, and higher coverage.**

I. Introduction

During the last decade, formal verification tools have been increasingly more popular for the pre- and post-silicon verification of a diverse class of IC designs, varying from custom processor designs to general-purpose ASICs. While multiple formal verification technologies are used in the industry (e.g. model checking, theorem proving, C-vs-RTL sequential equivalence checking), model checking tools account for most of the usage, judging from the number of available commercial tools as well as verification users in place. Furthermore, major EDA vendors (Cadence, Mentor Graphics and Synopsys) as well as a few startups (Averant, Jasper, OneSpin and Real Intent) offer competitive solutions. In this paper, we will use the term model checking synonymously with formal verification.

The extent to which an ASIC design tapeout schedule depends on formal verification is greatly contingent upon the scope of verification addressed by formal. Most often, formal is used as a supplement to simulation, to prove some specific difficult-to-verify behavior, local embedded RTL assertions, or interface protocol checks between blocks. Less often is formal used for end-to-end verification to replace simulation, so that formal verifies most or all functionality of a design and simulation is used only for higher chip-level or system-level verification. End-to-end formal usually requires almost the entire logic in the design to be analyzed by the formal tool, and poses significant complexity barriers.

Formal verification tool developers as well as users have long used abstraction techniques to overcome the computational complexity problem. Most tools deploy sophisticated abstraction-refinement algorithms under the hood [5], [19]. On top of that, formal users can deploy manually crafted abstractions [4], [6], [7], [11], [13] to further reduce the complexity of the proofs. In this paper, we will take a complex design with a large state space, and show how the use of abstraction models can help achieve end-to-end formal for this design.

Coverage metrics are widely used in simulation-based verification to improve the quality of the test suite and estimate the progress of the verification task [9], [18]. Coverage can help identify important gaps in the stimuli provided to the design-under-test, although it has a known limitation that coverage does not evaluate the quality of the simulation checkers. The same coverage metrics can be deployed for formal verification with the same limitation [16]. Besides identifying unintentional over-constraints in a formal environment, formal coverage can estimate the effectiveness of the abstraction techniques being deployed – for example, a set of abstraction techniques is useful, if it enables many more lines or expressions of code to be reachable in the same amount of CPU time.

In this paper, we use formal coverage metrics to quantitatively demonstrate that suitable abstraction models achieve convergence. We begin by introducing end-to-end formal verification in Section II, and the components required to build such an environment. We mention the role of abstraction techniques to solve end-to-end formal in Section III. Next, in Section IV we discuss how coverage is used for formal verification, and introduce a coverage-driven flow for formal verification. In Section V, we introduce the design we have. This design has a state space that is fairly large for a typical model checker to handle, more than 1 million flops. The design is an integral part of a large real-world ASIC switch. Section VI describes some of the constraints and checkers needed for formal verification, including the most important end-to-end data checker. In Section VII, we describe the abstraction models deployed to overcome complexity barriers. We present the coverage results in Section VIII.

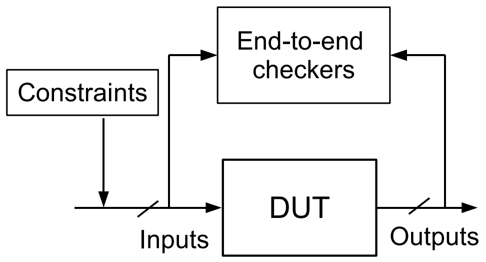

Fig. 1. End-to-end verification setup

II. End-to-End Formal

A. Checkers and Constraints

Besides reading the design-under-test (DUT), a model checker requires a set of checkers and constraints as inputs. The checkers and constraints can be written as properties in SystemVerilog Assertion language (SVA) [8]. However, often these checkers and constraints require supporting modeling code written in synthesizable SystemVerilog.

Checkers can vary widely in scope:

- Local checkers, also known as assertions. These checkers verify local properties of the design, and belong to one of the following:
  - Embedded RTL assertions. These assertions are local properties about the implementation details in the DUT, such as a state machine always stays one-hot encoded, or that a full FIFO is never written to. These assertions are typically written by the RTL designer, and embedded in the RTL code [21].
  - Interface assertions. These assertions encode the handshake protocol requirements for any of the interfaces of a design. These requirements can vary from a simple request-acknowledgement protocol to a more complex ARM AMBA AXI [15] or DDR2 protocol [6].
- End-to-end checkers (Fig. 1). These checkers primarily use a significant modeling code to encode a reference model for the required behavior of the design, by relating the correctness of the output data path of a design, given the transactions on the input datapath.

Bugs found through any of these checkers are useful. However, if formal is to be relied upon as a primary verification methodology for a design, simply verifying local checkers is not enough – a significant number of end-to-end checkers must be used to achieve adequate verification. Not surprisingly, proving the end-to-end checkers is usually computationally much more complex than local checkers, although there may be exceptions to this, and some local checkers may be difficult to prove too.

B. Complexity

The largest barrier to formal verification achieving the desired results is the complexity barrier faced by the tools. All known algorithms are worst-case exponential in the size of the cone-of-influence of the checks and constraints. For end-to-end formal verification, the model checking engine which is often the most effective is Bounded Model Checking (BMC) [1]. Although BMC can only find counterexamples, and not establish the full proof of any checks, the bounded proofs are good enough if the bounds are greater than the interesting corner-case behavior of the design, as judged by the verification or the design engineer.

Two complexity problems can interfere with BMC reaching acceptable proof bounds:

- the size of the logic in the cone-of-influence, including the number of flops as well as the combinational logic; and
- the state space diameter of the design, especially in presence of large counters, or sequentially deep logic.

The use of abstraction techniques, discussed in the next section, is the best strategy to overcome these complexity problems.

III. Abstraction Techniques

Abstraction techniques [3] are used to reduce the state space of the design, so that formal verification tools can solve a computationally easier problem. An abstraction is considered *sound* if does not reduce any design behavior, even if it adds to the design behaviors. We will only consider sound abstractions in this paper. Such abstractions can find proofs or failures faster. Every proof is guaranteed to be a proof on the original design. Each failure can be debugged to determine if it is a true counterexample due to an RTL bug, or a false counterexample due to an over-abstraction.

Examples of various abstraction techniques include:

1) Cut-points. Any internal logic in the design can be replaced by a cut-point, allowing that net to freely take a random value at any time [6], [12]. If a checker proves with such an abstraction, we can achieve significant reductions in run-time (of course, it also implies the need for additional checkers, since the proven checker is clearly independent of the excised logic).
2) Counter abstraction. Many designs have deep counters, for example, the initialization phase for DDR2 memory controllers last for hundreds of milliseconds, consuming millions of clock cycles. Many useful checks can be proved by abstracting the 2^n-state graph of an n-bit counter to a few states, e.g. 0, 1, *at-least-one*, *at-least-zero* [14].
3) Symmetric datatypes. Certain systems [7], [13] allow the users to specify that certain data types in the design are symmetric, and the values of these types are used only in certain symmetric ways (e.g. only compared for equality, or used as indices for arrays). This allows the system to reduce multiple symmetric proofs into a single one.
4) Data independence. When data moves across a design, and the design does not use the data contents for controlling the movement of the data, a few finite instantiations

of data values are sufficient to establish the correctness of any checkers [20]. This technique has been used to prove data correctness for many data transport hardware designs [11], [17].

5) Tagging. Often systems deal with a finite but large set of distinct data values [13]. Portions of such systems can be abstracted by simplifying the structure with respect to a specific or a symbolic tag.

Often, using an abstraction technique requires cut-pointing a section of the design, and adding constraints on the cut-points. The abstraction can be used to prove the desired checks. To complete the compositional proof [13] however, a second step is required – the constraints need to be converted into checks, and proven on the previously excised logic.

IV. Coverage

A. Coverage in Simulation

In simulation-based verification, coverage metrics are used heavily to determine when simulation is complete. The most common coverage metric is code coverage, including line, expression, FSM and toggle coverage. Line coverage, for example, computes what percentage of RTL statements in the DUT were exercised by a given set of tests. For example, consider:

```
1:  always @(posedge clk) begin
2:    if ((a && b) || c)
3:      e <= d1;
4:    else
5:      e <= d2;
6:  end
```

This example results in two line coverage targets, corresponding to lines 3 and 5. If a test causes `c` to be 1, the line 3 will be marked as covered. If no test in a test suite covers line 5, line coverage for the suite will be reported at 50%.

100% *judged* line coverage (given, say 99% *automated* coverage) is frequently a requirement for an ASIC tapeout – each line that is not automatically reported as covered in simulation, must be manually judged to be either redundant, or legacy code, or symmetric to another tested line. Tapeout would be delayed until more tests are written to cover the remaining lines. 100% line coverage does not imply an absence of bug. Still, line coverage helps measure the continuous progress of verification completeness in a dynamic chip design schedule, and often points to important coverage holes.

B. Formal Coverage Metrics

The same coverage metrics used in simulation can be applied to answer the question of whether the planned formal verification tasks are complete, or how much the formal verification tasks complement the simulation effort [16].

Simulation-based line (or expression) coverage metrics can be used to mean exactly the same in formal – given the constraints used and the proof depths reached in BMC (say, n cycles), report what percentage of line (or expression) targets are reachable in n cycles. For the example in Section IV-A, if `((a && b) || c)`, in line 2, is reachable in n cycles, this line would be reported as covered, and otherwise, not. Thus, line coverage numbers would mean the same in simulation – whether a certain coverage target is exercised or not. And for formal, this would measure the quality of constraints (i.e. absence of over-constraints), as well as the BMC proof depths. Abstraction techniques, described in Section III, can help in achieving higher proof depths, improving the coverage results and thereby increasing the value of formal verification. Commercial formal tools are beginning to support the measurement of formal coverage.

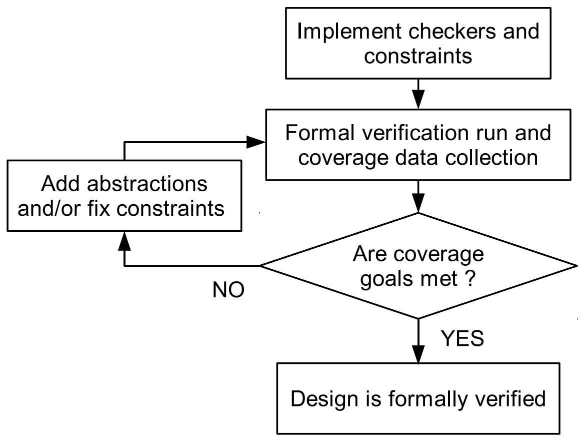
Implement checkers and constraints
Formal verification run and coverage data collection
Add abstractions and/or fix constraints
Are coverage goals met ?
NO
YES
Design is formally verified

Fig. 2. Formal verification coverage flow

C. Formal Coverage Flow

Refer to Fig. 2 for the flow we use for a coverage-driven formal verification deployment. Like simulation, code coverage results are used to identify missing gaps in the formal verification implementation. Abstraction models are used heavily to increase the coverage to acceptable levels on complex designs where formal would otherwise be infeasible.

Since we are using the same coverage metrics, we can even merge coverage results. It is often the case that one block is verified end-to-end with formal, and a larger block containing this block is verified with simulation. Even if the line coverage with formal is not 100% for the block, as long as the unified simulation and formal line coverage is 100%, verification is considered complete from the perspective of line coverage goals. This of course relies on an important assumption – *that the set of formal checkers is as complete as the set of simulation checkers*. Although formal coverage helps determine the quality of constraints as well as sequential depth reached, like simulation, coverage does not imply anything about the completeness of checkers. This has to be evaluated independently.

V. CellReformatter Design

The *Packet Rewrite Module* (PRM) design modifies incoming packets from multiple ports and reformats these packets before passing them on. Fig. 3 shows the sequence of operations on a packet when it passes through various

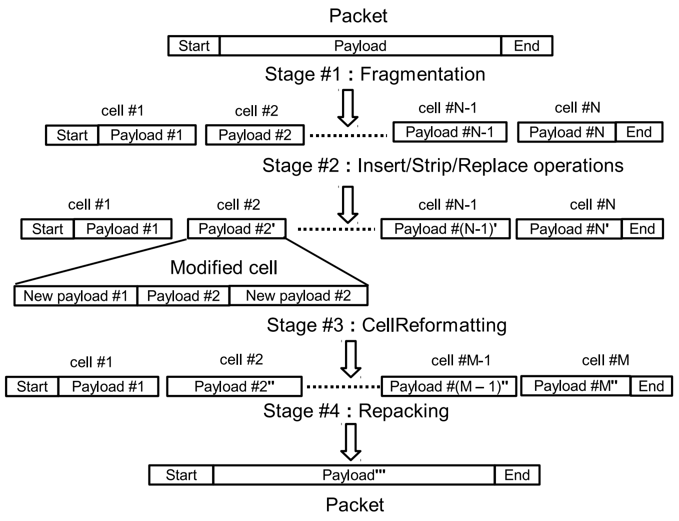

Fig. 3. Various stages of PRM

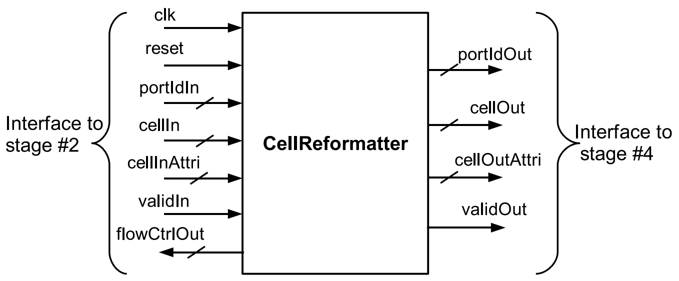

Fig. 4. Toplevel of CellReformatter

stages of PRM. The four stages are Fragmentation (Stage #1), Insert/Strip/Replace operations on packet payload (Stage #2), CellReformatting (Stage #3) and Repacking (Stage #4).

A. *Functional Specification*

By the end of Stage #1, each packet is fragmented into single/multiple subpacket(s), called *cells*, depending upon the payload size. A cell has three main attributes: start of packet (SOP), end of packet (EOP) and number of payload bytes carried (ValidBytes). Some desired properties of the cells are:

1) The first and only the first cell has SOP as 1
2) The last and only the last cell has EOP as 1
3) A cell with EOP as 0 will have ValidBytes as 128
4) A cell will have ValidBytes greater than 0

e.g. As an example, suppose at the end of Stage #1, cell #1 has SOP as 1, EOP as 0 and ValidBytes as 128, cell #2 has SOP as 0 and EOP as 0 and ValidBytes as 128 and cell #N ($N = 3$) has SOP as 0, EOP as 1 and ValidBytes as 120.

Stage #2 modifies bytes of payload of a cell by performing insert, strip and replace operations. ValidBytes of each cell also gets modified accordingly. In Fig. 3, for the simplicity of illustration, we show that only cell #2 is being modified – i.e., the payloads of other cells do not undergo any change. Payload of cell #2 gets modified to payload #2' by insertion of two new payloads, one before, and one after the original payload, as depicted by Modified cell in the figure. In the actual design, Stage #2 can modify any or all N cells. With a combination of insert, strip and replace operations, ValidBytes of a modified cell can vary between 1 and 256. Suppose, in our example, after Stage #2, ValidBytes of cell #2 is 144, resulting in ValidBytes of 128, 144 and 120, respectively, for the three cells. Due to these modifications, a cell may not satisfy the desired properties on ValidBytes listed in the previous paragraph, at the end of Stage #2. The purpose of the next Stage #3, which constitutes our DUT, the CellReformatter design, is to rectify this.

CellReformatting (Stage #3) reformats the modified cells so that they satisfy the desired ValidBytes properties and can be repacked into a packet in the next stage. The number of cells for a packet at end of Stage #3 may be different than the number of cells at the beginning of the stage, depending upon reformatting. In our example, the payload of the non-EOP cell #2, at start of stage #3, does not satisfy the non-EOP ValidBytes property. So, in the CellFormatter stage, this cell gets reformatted to comprise of the first 128 bytes of the input cell. The remaining 16 ($= 144-128$) bytes are appended before the payload of cell #3, resulting a modified cell #3 of 128 bytes. The 8 ($= 120 + 16 - 128$) trailing bytes of the original cell #3 constitute a new cell #4.

Repacking (Stage #4) repacks the reformatted cells into a packet that can be forwarded to port(s).

B. *Micro-Architecture*

CellReformatter supports reformatting of cells for packets from 56 different concurrent ports. Cells for a packet on one port may be interleaved with cells from other ports. This increases the design and verification complexity. Fig. 4 shows the interfaces of the CellReformatter design, the interface to Stage #2 on the left side, and the interface to the Stage #4 on the right side. *portIdIn* refers to incoming port. *cellIn* represents incoming cell, varying between 1 and 256 bytes long. *cellInAttri* is a structure consisting of cell attributes, including SOP, EOP, ValidBytes. *validIn* and *validOut* indicate the validity of inputs and outputs of CellReformatter respectively. Inputs are valid if they are transmitted when *validIn* is high. Similarly, outputs are valid if they arrive when *validOut* is high. *flowCtrlOut* is a feedback to Stage #2 to stop it from sending more cells for the relevant port. Thus this acts as a throttle and prevents the overflow of internal FIFO(s) for the port. *flowCtrlOut* is a 56-bit wide signal with each bit corresponding to a port.

Memory Design: CellReformatter has FIFOs for storing the reformatted cells (*dataFifo*) and its attributes (*statusFifo*). Each of *dataFifo* and *statusFifo* is implemented as an SRAM memory, with separate regions for different ports. The least-significant bit of *portId*, called *oddBank*, is used to determine which of the two banks is used, while the remaining higher-significant bits, called *streamId*, are used as memory address:

```
portId = {streamId, oddBank}
```

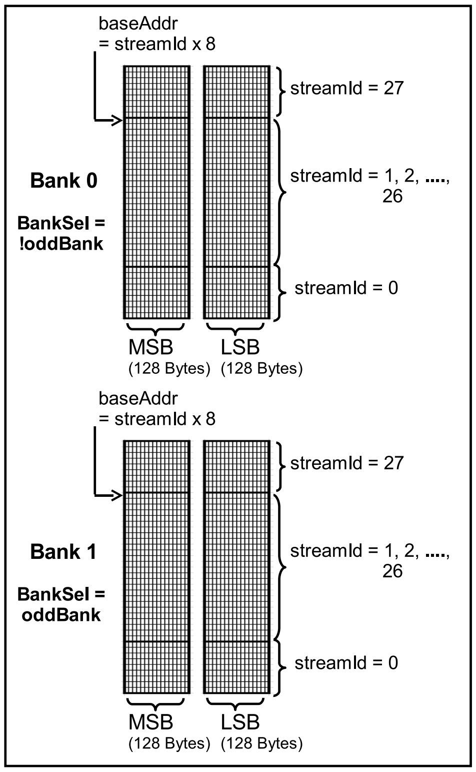

Fig. 5. Banked architecture of *dataFifo*

As shown in Fig. 5, the memory in each bank is logically divided into 28 *streamId*'s. Each bank of the *dataFifo* memory is further divided into two separate single-port SRAMs 128-bytes wide, called *MSB* and *LSB*. Further, each port occupies a depth of 8 entries in each of *MSB* and *LSB*. Note that in one clock at most 256 bytes will arrive from Stage #2 for a given port. Depending on where we wrote the last data for this port, this data will cause one or two writes into the *MSB* and/or the *LSB* section for that port. For the example in Section V-A, when cell #1 arrives, all of its 128 bytes are written into *LSB*, at depth of 0. When cell #2 arrives, 128 of its least significant bytes are written to *MSB* at depth of 0, and the remaining 16 bytes are written to *LSB* at depth of 1. Finally, when cell #3 arrives, its 112 ($= 128 - 16$) least significant bytes are shifted up by 16 bytes and written to *LSB* at depth of 1, and the remaining 8 ($= 120 - 112$) bytes are written to *MSB* at depth 1.

CellReformatter has another FIFO (*stateFifo*) for remembering the current write and read address pointers into *dataFifo* for a port. This FIFO is also implemented by a single-port two-bank SRAM memory.

Latency: The fastest end-to-end latency of CellReformatter is 6 clock cycles; the FIFO write operation has a 4-cycle latency and the FIFO read operation has a 3-cycle latency. A constraint on the design, that *oddBank* toggles every clock cycle, ensures that bank contention is avoided for simultaneous read and write operations.

TABLE I
DESIGN SUMMARY OF CELLREFORMATTER

Parameters	Values
Inputs	4,425
Outputs	3,488
Total flops	1,048,481

C. Challenges to Formal

The major challenges to achieving convergence with formal are:

1) *Large number of flops*. Greater than 1 million storage elements (Table I) is enough to create a state space search problem that cannot be solved without the use of abstraction models. This large count is dominated by the number of flops needed for *dataFifo*: due to number of ports (56), number of per-port cells stored (16) and the size of each cell (128 bytes).
2) *High sequential depth due to latency*. No input port at input is allowed to appear more than once in 4 consecutive clock cycles. This constraint, along with the latency of CellReformatter and the FIFOs depths, implies that a high sequential depth is required for proofs.

VI. CHECKERS AND CONSTRAINTS

The CellReformatter design has following interface constraints:

1) For a port, between 2 cells at input with SOP as 1, there should be a cell with EOP as 1
2) For a port, between 2 cells at input with EOP as 1, there should be a cell with SOP as 1
3) For a port, the next valid cell after an EOP as 1 must have SOP as 1
4) For a port, input cell should have ValidBytes > 0
5) For a port, input cell should have ValidBytes < 256
6) The oddBank should toggle each cycle
7) A port at input should appear no more than once in 4 consecutive clock cycles

The interface checkers are as follows:

1) For a port, between 2 cells at output with SOP as 1, there should be a cell with EOP as 1
2) For a port, between 2 cells at output with EOP as 1, there should be a cell with SOP as 1
3) For a port, the next valid cell after an EOP as 1 must have SOP as 1
4) For a port, output cell should have ValidBytes > 0
5) For a port, output cell with EOP as 0 should have ValidBytes as 128

End-to-end checkers are written using a reference model that tracks the outstanding cells for a port, and also reformats them into 128-byte cell boundaries. Examples of end-to-end checkers:

1) For a port, the valid output (*validOut*) can be 1 only if there are outstanding cells in flight that have not been sent out
2) For a port, payload of a cell at the output should correspond to payload of expected cell in the reference model, computed based on payloads that arrived at the input in the past

Consider this last end-to-end checker, the most important checker for this DUT. The checker is written in SVA as:

```
property cellOutMatch_a;
 @(posedge clk) disable iff(reset)
  (validOut &&
   (portIdOut == watchedPort)) |->
  (cellOut[watchedByte][watchedBit] ==
   referenceBit);
endproperty
cellOutMatch_A:
 assert property(cellOutMatch_a);
```

We used the following symbolic variables in this checker:

1) *watchedPort*. This variable, varying between 0 and 55, represents the specific port that is being verified. While the design interleaves the inputs and outputs across multiple ports, in one trace, we can verify the outputs for a specific port.
2) *watchedByte*. This variable, varying between 0 and 127, represents the specific byte number in an output cell that is being verified in this trace.
3) *watchedBit*. This variable, varying between 0 and 7, represents the specific bit being verified in the *watchedByte* byte.

Since these variables are symbolic, all possible output data bits from all possible ports are verified with the end-to-end checker. In any given trace of execution, these variables can be kept constant with SVA constraints like the following:

```
property watchedPort_r:
 @(posedge clk) disable iff(reset)
  (##1 $stable(watchedPort));
endproperty
watchedPort_R:
 assume property(watchedPort_r);
```

This end-to-end checker also depends on the predicted value of the output bit from the reference model, *referenceBit*. The reference model is implemented in SystemVerilog, and using the three watched symbolic variables, implementing a queue of watched bits in flight in the design. The value of *referenceBit* equals the bit at the top of the queue. We will discuss an abstraction in Section VII-B, that shows how to implement this reference model more efficiently.

VII. Abstraction Models

We have a design with more than 1 million flops. This will lead to state space explosion with any existing formal verification tool. Abstractions are essential to achieve convergence on a design like this.

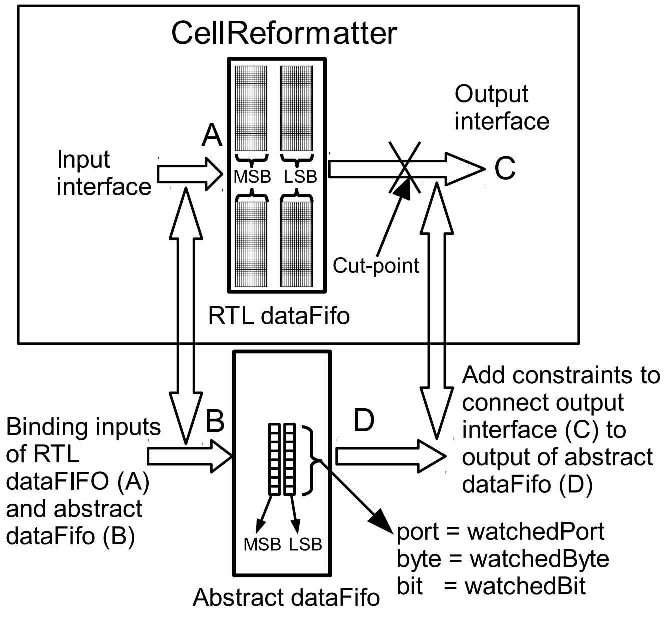

Fig. 6. Deploying memory abstraction for *dataFifo*

A. Memory Abstraction

The *dataFifo* memory stores up to 16 cells for every port, 8 cells in *LSB*, and 8 in *MSB*. The memory stores the reformatted cells, after performing the necessary shifting, described in Section V-B. Since the main end-to-end checker (*cellOutMatch_A* in Section VI) uses symbolic watched variables for the port number and the verified bit in a cell, each flop in *dataFifo* is essential to establish the correctness of the proof. This places a tremendous burden on a formal verification tool.

Using the three watched symbolic variables, we create an abstraction for *dataFifo*, shown in Fig. 6. This abstraction model contains only 16 flops, 8 for an abstraction of the *LSB* section of the memory banks, and 8 for an abstraction for the *MSB* section.

We tie the inputs of the abstract *dataFifo* to the inputs of the RTL *dataFifo* (implemented by the SystemVerilog bind construct). In addition, *watchedPort*, *watchedByte* and *watchedBit* are extra inputs to the abstract *dataFifo*.

When there is write to the RTL memory, if the write address input matches *watchedPort*, we pick the *watchedBit* bit from the *watchedByte* of the write data input to the memory, and store that in one of the 16 bits in the abstract memory (4 least significant bits of the write address input determine which of the 16 per-port cells was being written by the write command).

To enable the abstraction, we add cut-points at the read data outputs of the RTL *dataFifo*. Further, we add a constraint that if the read address input matches *watchedPort*, then *watchedBit* bit of *watchedByte* read data output byte equals the value stored in the i-th (of 16) abstract *dataFifo* bits (where i equals the 4 least significant bits in the read address input). This enables the read data for the watched bit to be faithful to what is in the RTL, and the remaining bits or read data output for a non-watched port to be arbitrary. But, since the checker

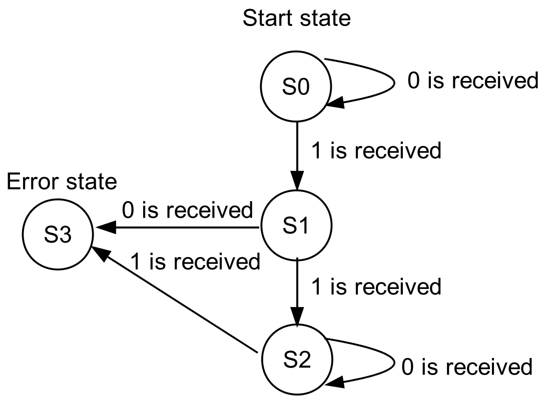

Fig. 7. State machine for pattern $0^{\star}110^{\omega}$ detection

is checking only the watched port and the watched bit, the abstraction should not give a false negative.

Using this abstraction model, we have reduced 917,504 flops in the RTL *dataFifo* that were in the cone-of-influence of the end-to-end checker to the 16 flops in the abstract *dataFifo*. More important, this abstraction does not introduce any false negatives with respect to the end-to-end checker. Similar abstraction models were built for *statusFifo* and *stateFifo*, albeit only with respect to *watchedPort*. See Table II for the reductions in the cone-of-influence; note that there are other peripheral flops in the memories because the memories have additional flops due to the latency, as well as some parity-checking flops.

Note that to complete the proof with abstractions using compositional reasoning, we also need to separately prove that the abstract *dataFifo* is a sound abstraction of the RTL *dataFifo*. We do this by removing the cut-points on the read data outputs, and reversing the constraints on the read data outputs to checkers, then proving them independently of the main end-to-end checker.

B. Data Independence Abstraction

The main end-to-end checker (*cellOutMatch_A* in Section VI) requires a reference model for the expected behavior of the *referenceBit* bit. Even after the memory abstraction in the previous section, we know that there are at least 16 bits in flight for the watched bit we want to track. However, this is just a lower bound, since there may be additional bits on the way to *dataFifo*, or on the way from *dataFifo*. Suppose there are at most n bits in flight we need to track; to implement the reference model with a FIFO, we will need at least n entries in the reference model FIFO.

Fortunately, we can use a variant of the data independence abstraction [20], to avoid the dependence on the unknown n, and more importantly to verify with more efficient state space. The data independence theorems state that for certain data-independent designs (when data is merely transported across the design, and not queried to make the routing decisions), a small set of finite data values is sufficient for end-to-end proofs. For our end-to-end checker, it is sufficient to prove the preservation of infinite streams of the form $0^{\star}110^{\omega}$ across the design. Each stream in this set has a finite but arbitrary number of 0's followed by two consecutive 1's, followed by an infinite sequence of 0's; for example, input sequences like $11000\cdots$, $011000\cdots$, and $000\cdots011000\cdots$. Note that given the three watched symbolic variables, we need to apply this abstraction only to the consecutive bits that will be written to the abstract *dataFifo* from the previous section.

We add a constraint to the inputs of the DUT so that watched bits create this sequence by using the state machine in Fig. 7. We constrain the inputs so that the error state $S3$ is never reachable. Next, we use an identical state machine to verify the output watched bit from the DUT. We modify the checker so that the expected *referenceBit* is not allowed to be 0 is state $S1$, or to be 1 in state $S2$ – all other values are allowed for *referenceBit*.

Using this data independence abstraction, we do not have to implement a reference FIFO, whose depth is design-dependent. We save additional flops in the cone-of-influence, and proofs run much faster.

VIII. Experimental Results

We used the Cadence® Incisive® Enterprise Verifier (IEV) tool [2] for this verification. Since the un-abstracted design has more than 1 million flops, hence it is not feasible to run formal without deploying the abstraction models described in Section VII.

The verification setup for the DUT consists of the CellReformatter RTL, checkers and constraints (using the necessary reference models), and the abstraction models described in Section VII. There are 23 checkers and 21 constraints. We found 15 bugs in the RTL design.

As expected, BMC was the most effective engine for verifying the main end-to-end checker. For the shortest possible packet, the data can be seen at the output of the design at a BMC proof depth of 7 clock cycles. However, the most interesting behavior of the design occurs when *dataFifo* is full before data is unloaded to the outputs. By understanding the design micro-architecture, including the latencies and memory depths, it was determined that a proof depth of 63 cycles is sufficient to hit this extreme behavior (the constraint that successive input data for the same port must be 4 cycles apart is responsible for much of this depth).

We use the IEV code coverage feature to report the amount of coverage hit to determine if the use of abstractions was successful in covering the design. Coverage results are reported in Table III. We notice that the expression coverage is 100% and the line coverage is almost 100% at a proof depth of 63. The missing coverage holes need to be judged and possibly waived by the design engineers. For the un-abstracted design, the BMC proof depths reached at similar run-times are close to 0, hence the corresponding coverage results are close to 0% (not surprising given the amount of state in the DUT).

The level of coverage reached is very much in line with the desired verification coverage, if we were verifying this design using simulation. We must remind the reader that the desired

TABLE II
COMPARISON OF RTL AND ABSTRACT MEMORIES

Memory	Flops in RTL	Flops in abstract memory
dataFifo	948,636	204
statusFifo	89,986	4,854
stateFifo	2,394	268

TABLE III
FORMAL COVERAGE RESULTS

Proof depth	Line coverage	Expression coverage
7	96.5%	100.0%
15	99.5%	100.0%
63	99.7%	100.0%

coverage result must be considered in conjunction with the confidence in the completeness of checkers. Unfortunately, as with simulation, formal code coverage by itself does not yet determine the completeness of checkers. However, we do know that with the use of the abstraction models, we were able to exercise almost all the RTL code. Without these abstraction models, we would not get much more than 0% coverage, and formal verification would not have been able to replace simulation on this design.

IX. CONCLUSION

In this work, we show how end-to-end formal can replace simulation efforts and provide faster verification with higher coverage. Without the use of abstraction models, formal verification is often infeasible for end-to-end verification. With the use of abstraction models, we can counter state space explosion, and reach acceptable levels of quantifiable code coverage metrics. These results can be integrated with simulation-based code coverage results on neighboring designs, or the rest of the system.

ACKNOWLEDGEMENT

The authors would like to thank Sandesh Borgaonkar, Anton Lopatinsky and Deepak Pant for their for help and support in making this work possible.

REFERENCES

[1] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, Y. Zhu. Bounded model checking. *Advances in Computers*, 58, 2003.
[2] *Cadence Incisive Enterprise Verifier datasheet*. Cadence Design Systems, Inc.
[3] E. M. Clarke, O. Grumberg, D. E. Long. Model checking and abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5), pp. 1512–1542, 1994.
[4] E. M. Clarke, O. Grumberg, H. Hiraishi, S. Jha, D. E. Long, K. L. McMillan, L. A. Ness. Verification of the Futurebus+ cache coherence protocol. *Formal Methods in System Design*, 6(2), pp. 217–232, 1995.
[5] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM* 50(5), pp. 752–794, 2003.
[6] A. Datta, V. Singhal. Formal Verification of a Public-Domain DDR2 Controller Design. In *Proc. VLSI Design*, pp. 475–480, 2008.
[7] C. N. Ip, D. L. Dill. Better verification through symmetry. *Formal Methods in System Design*, 9(1/2), pp. 41–75, 1996.
[8] *IEEE standard for SystemVerilog: unified hardware design, specification and verification language*. IEEE Std. 1800-2009.
[9] M. Kantrowitz, L. M. Noack. I'm done simulating; now what? Verification coverage analysis and correctness checking of the DECchip 21164 Alpha microprocessor. In *Proc. Design Automation Conf.*, pp. 325–330, 1996.
[10] S. Katz, O. Grumberg, D. Geist. Have I written enough properties? A method of comparison between specification and implementation. In *Proc. CHARME, LNCS 1703*, pp. 280–297, 1999.
[11] B. A. Krishna, A. Sullerey, A. Jain. Formal verification of an ASIC Ethernet switch block. In *Proc. FMCAD*, pp. 13–20, 2010.
[12] R. P. Kurshan. Formal verification in a commercial setting. In *Proc. Design Automation Conf.*, pp. 258–262, 1997.
[13] K. L. McMillan. Verification of an implementation of Tomasulo's algorithm by compositional model checking. In *Proc. CAV, LNCS 1497*, pp. 110-121, 1998.
[14] F. Pong, M. Dubois. A new approach for the verification of cache coherence protocols. *IEEE Trans. Parallel Distrib. Syst.* 6(8), pp. 773-787, 1995.
[15] C. Sayer, J. Sonander. Formal verification of AMBA 3 AXI bus systems. In *ARM Information Quarterly*, pp. 15-17, 4(2), 2005.
[16] V. Singhal, P. Aggarwal. Using Coverage to Deploy Formal in a Simulation World. In *Proc. CAV, LNCS 6806*, pp. 44-49, 2011.
[17] C. Stangier, U. Holtmann. Applying formal verification with Protocol Compiler. In *Proc. Euromicro Symp. Digital Systems Design*, pp. 165–169, 2001.
[18] S. Tasiran, K. Keutzer. Coverage metrics for functional validation of hardware designs. *IEEE Des. Test*, 18(4), pp. 36–45, 2001.
[19] C. Wang, G. D. Hachtel, F. Somenzi. *Abstraction refinement for large scale model checking*. Springer, 2006.
[20] P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Proc. POPL '86*, pp. 184–193, 1986.
[21] P. Yeung. How to instrument your design with simple SystemVerilog assertions. *EE Times DesignLine*, January 26, 2011.