Using Model Checking to Prove Constraints of Combinational Equivalence Checking

Xiushan Feng, Joseph Gutierrez*, Mel Pratt, Mark Eslinger, Noam Farkash†
*Advanced Micro Devices, Inc.
{Shaun.Feng, Joseph.Gutierrez}@amd.com
†Mentor Graphics Corp. USA
{Mel_Pratt, Mark_Eslinger, Noam_Farkash}@mentor.com

***Abstract*—RTL-to-gate logic equivalence checking is a very critical step inside circuit design flows. It is used to make sure the gate-level circuit doesn't alter functional behaviors of the RTL. Of the various commercial logic equivalence checking tools, Combinational Equivalence Checking (CEC) tools are often used to prove equivalence between RTL and gate due to their high efficiency and good scalability. However, unlike Sequential Equivalence Checking (SEC), which traverses the product Finite State Machine (FSM), combinational equivalence checking proves equivalence for combinational circuits (i.e., Equivalences are formally verified for combinational logic cones between the state points.).**

For AMD high-performance microprocessor designs, we aggressively exploit optimization techniques to achieve competitive performance, power consumption, and die size. It is very common to see false non-equivalence of CEC on state spaces that are either optimized away or infeasible based on the circuit behavior. In order to avoid such false non-equivalences, we use design constraints to remove the invalid state space. As a consequence, the CEC results now highly rely on the correctness of these constraints.

This paper presents our experiences and learning with formal verification of CEC constraints using model checking tools.

***Index Terms*—Model Checking, Formal Verification, Combinational Equivalence Checking**

I. Introduction

In current circuit design/verification flows, most of the simulation/formal-based verification is done at the Register Transfer Level (RTL). After the RTL circuit is extensively verified, we synthesize and implement the RTL into a gate-level circuit. As soon as the gate-level circuit is available, we need to prove logical equivalence between the RTL and the gate-level to make sure the gate-level circuit hasn't altered the functional behavior of the RTL. Otherwise, all of the RTL verification results become invalid and non-applicable.

A. Sequential vs. Combinational Equivalence Checking

Based on whether equivalence needs to be proven for each state point, there are two major types of logic equivalence checking tools – *combinational* and *sequential* [6, 9, 13].

- *Combinational Equivalence Checking (CEC)*: CEC tools prove equivalence for combinational circuits. CEC tools don't need to analyze logic across state points. Equivalence checking passes only if all state points are properly mapped between RTL and gate and mapped points are proven to be equivalent (Fig. 1).

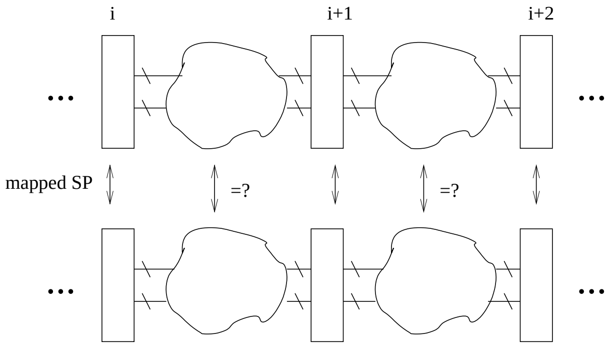

Fig. 1. Combinational Equivalence Checking (CEC). Equivalence is proven by verifying each combinational logic cone.

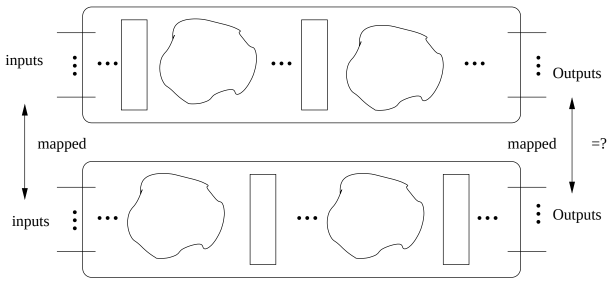

Fig. 2. Sequential Equivalence Checking (SEC). Equivalence is proven by verifying primary outputs.

- *Sequential Equivalence Checking (SEC)*: SEC tools prove equivalence for sequential circuits. Instead of proving equivalence for each combinational logic cone, SEC tools prove equivalence based on circuit inputs and previous values on state points (e.g., DFF, DLAT, etc.). SEC algorithms analyze the product finite state machine of the RTL and gate designs. Therefore, equivalence of intermediate state points inside the circuit may not be required. If we flatten the whole sequential circuit and fold all the state points into one big combinational circuit, then the SEC problem is reduced to CEC (Fig. 2).

From these descriptions, we can see that SEC tools have to deal with the extra complexities of state elements. Although both CEC and SEC tools are built upon similar formal proof engines (e.g., BDDs [2] or SAT solvers [1, 10]), the SEC tools are more sensitive to the size of the design. Of the various commercial logic equivalence checking tools, CEC tools are often used to prove equivalence between RTL and gate-level designs due to their high efficiency and good scalability.

At AMD, due to the size of our designs, SEC is not a practical option for our design flow. In addition, we have synthesis/implementation rules in place to guarantee the 1-to-1 mapping between RTL and gate state points. Therefore, CEC is our option of choice for logic equivalence checking.

The primary limitation of CEC is that it only verifies combinational logic cones between state points. In real circuits, each state point is driven by its fan-in logic cone, but such functional behavior is not taken into account by CEC tools because CEC doesn't analyze logic across state boundary points. CEC tools will use the whole exponential combination of boolean values (0 and 1) for state points when analyzing the combinational logic cones that they drive. In typical designs, this scenario is usually not valid. Possible values of a state point are constrained by the previous stages of logic and state points driving the state point in question. Without using these constraints, we could have false non-equivalences.

B. False Non-equivalence

False non-equivalence (or false negative) is a term used for the scenario in logic equivalence checking where there is a non-equivalent point found by the tool, but in the actual circuit behavior, this point is equivalent.

For AMD high-performance microprocessor designs, we aggressively exploit optimization techniques to achieve competitive performance, power consumption, and die size. It is very common to see false non-equivalence when using CEC on state spaces that are either optimized away or infeasible based on the circuit behavior.

In order to avoid false non-equivalence, we need to identify the invalid state space first. Then, we write design constraints for the CEC tools to ignore the invalid state space (i.e., we don't care whether it is equivalent or not on the invalid state space.). Most of these constraints are identified manually from the actual design or from analysis of the higher-level design block that contains the current design. Please keep in mind that CEC is a conservative approach. If CEC can prove equivalence without any constraint, then the equivalence holds for the entire state space which includes the valid states. Our designers follow a methodology of only adding constraints when they are necessary to resolve false non-equivalences. After the design passes CEC, we must then formally prove these constraints, noting that false constraints may invalidate all of the CEC results.

Here is a simple example to show why we need constraints for CEC and how a bad constraint can invalidate CEC results.

For example, we have an RTL circuit:

```
module TOY (CLK, SHIFT, G);
input CLK;
input [1:0] SHIFT;
output [3:0] G;
reg [3:0] B;
reg [3:0] G;

always @(posedge CLK)
      B[3:0] <=  4'h2 ** SHIFT;

always @ *
     case (B[3:0])
       4'b0001 : G[3:0] = B[3:0];
       4'b0010 : G[3:0] = B[3:0];
       4'b0100 : G[3:0] = B[3:0];
       4'b1000 : G[3:0] = B[3:0];
       default : G[3:0] = 4'b1111;
     endcase

endmodule
```

Given the above RTL, synthesis tools or human designers can implement the following gate-level circuit:

```
module TOY (CLK, SHIFT, G);

input CLK;
input [1:0] SHIFT;
output [3:0] G;

reg [3:0] B;

always @(posedge CLK)
      B[3:0] <= 4'b0001 << SHIFT[1:0];

assign G[3:0] = B[3:0];

endmodule
```

The equivalence of these two designs is obvious for an SEC tool. For all possible inputs with all possible values on the flops, the outputs of the two designs are equivalent. However, since CEC tools (e.g., Cadence Conformal LEC [3]) cannot evaluate logic on the other side of the flops, they will report non-equivalences for $G[3:0]$ by finding a random vector on the flops (e.g., $B[3:0]$ = 4'b1011). We can easily see that such a test vector is not possible due to the decoder logic before the flops.

In order to work around the false non-equivalence that may be found, we have to tell the tool that there is a constraint on $B[3:0]$. For example, in Conformal LEC, we can give the following constraint:

```
$constraint( $one_hot(B[3:0]));
```

This constraint will create a don't-care space for equivalence checking. Any values that don't satisfy the above one-hot constraint are don't-cares. CEC tools will ignore any non-equivalence in the don't-care space and successfully prove the equivalence of the two circuits.

We can see constraints in CEC are very useful. However, they are very dangerous if not used properly. CEC users can easily ignore any state space that they want, which will introduce false equivalence [12]. For the same RTL in the TOY example, if a designer gives the following gate-level implementation:

```
module TOY (CLK, SHIFT, G);

input CLK;
input [1:0] SHIFT;
output [3:0] G;

reg [3:0] B;

always @(posedge CLK)
      B[3:0] <= 4'b0001 << SHIFT[1:0];

assign G[3:0] = ~B[3:0];

endmodule
```

and the following Conformal LEC constraint:

```
$constraint(B[3:0] == 4'b0000);
```

Conformal LEC will still report equivalences for all state points and outputs. However, the equivalence was proven on an infeasible state ($B[3:0]$ = 4'b0000) and all valid states are ignored as don't-cares by using this constraint. Such a false CEC constraint can introduce serious design bugs that will cause huge verification efforts later inside your design flow. In the worse case scenario, you may never locate these bugs before taping out a broken chip.

In this paper, we will present our experience on how to formally prove CEC constraints by using model checking tools.[1] We start with how to maintain CEC constraints inside our design flow and how to translate them for use in CEC and model checking tools. We will also show some challenges encountered in proving such constraints. In the end, we give some suggestions to the EDA community on how to improve these tools to support a better flow.

II. Constraint Harvesting and Translations

CEC constraints can come from multiple places: CEC constraints can be written inside the RTL by using assertion specification languages, then harvested and translated for the CEC tools; CEC constraints can also be created when circuit designers run RTL versus gate-level equivalence checking. Each usage has its advantages. Harvesting RTL assertions can give us some higher-level constraints, such as architecture-level constraints. In addition, assertions in the RTL can be verified by RTL simulation tools using test vectors. Also, allowing circuit designers to provide their own CEC constraints can give them more flexibility to design or optimize their circuits. The

[1]We use Cadence Conformal LEC [3] as our CEC tool. Our model checking tool of choice is 0-In [8] from Mentor Graphics. Although we attempt to use standard assertion languages for the test cases used in this paper, for some test cases, we have used proprietary constraints (or assertions) from Conformal or 0-In.

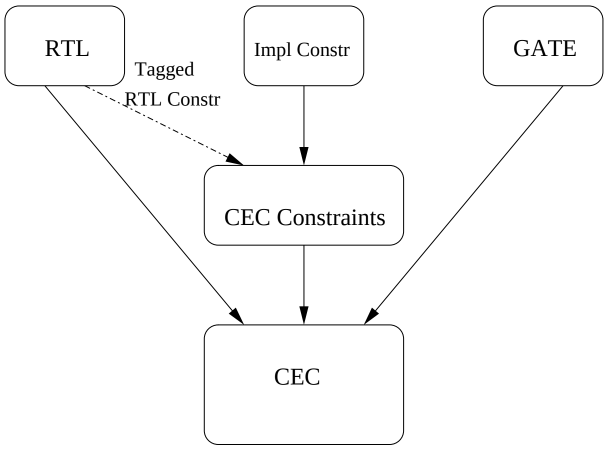

Fig. 3. Constraints in CEC. Harvested from RTL assertions or from implementation constraints

circuit designers have done the synthesis/implementation, then run the CEC tools to verify the RTL to gate-level equivalence, so they are the ones to best provide the CEC constraints when they find the false non-equivalence.

With the CEC constraints, RTL, and gate-level circuit, the CEC tool can then proceed with the equivalence checking (Fig. 3 shows how constraints are prepared for the CEC tool). A good design flow should allow constraints from these two sources and can harvest and translate constraints for both the CEC and model checking tools.

Because different tools use different assertion/constraint languages (or libraries), it is a big challenge to translate constraints from various sources for different tools. To help our tools support the harvesting/translation step, we have to put extra information (e.g., pragmas) around our assertions/constraints.

A. RTL Assertions Reused as CEC Constraints

In each design project, the RTL verification team can use different assertion languages (e.g., OVA, OVL, etc.) for their simulation. To support that, we have to use pragmas to tell our flow whether an assertion will be needed for the equivalence checking and model checking steps. We use two kinds of pragmas to guide constraint harvesting and model checking tools on how to treat these CEC constraints.

- *'//! CEC MC-ASSUME'*: this pragma tells our flow that the assertion followed by this tag is used as a constraint for both CEC and model checking. We rely on RTL simulation to verify these constraints.
- *'//! CEC MC-PROVE'*: for CEC constraints with such a tag, the model checking step must prove them.

For example, here is an OVA assertion inside RTL code:

```
//! CEC MC-ASSUME
// ova forbid_bool((~foo & bar),
      "when foo is 0, bar cannot be 1!");
```

This OVA will be harvested as a CEC constraint. It will be translated into an assumption for the model checking tool.

An equivalent OVL RTL assertion could look like this:

```
//! CEC MC-ASSUME
ovl_assert
 #(.msg("When foo is 0, bar cannot be 1!")
   OVL_TEST (
     .clock(CLK),
     .reset(RST),
     .test_expr(~(~foo & bar)),
     .fire(),
     .enable(‘EN_ABV),
)
```

By recognizing the proper pragmas, these RTL assertions are harvested for the current design under CEC check using CAD tools. After they are harvested, we need to translate them into the format that the CEC tool can understand. For Conformal LEC, we have to translate the above CEC constraint to:

```
$constraint RTL_C_1 (~(~foo & bar));
```

We give each CEC constraint a unique name, so we can track them inside the various flows.

Most model checking tools can not read OVA assertions directly due to their proprietary nature and for most OVLs there are many global references for OVL ports or parameters (e.g., clock, reset, enable, port, etc.). These global references may not be resolvable inside the current design. For these reasons, the constraints must be translated before they can be used by model checking tools.

For example, for one project, we translate an OVA assertion to 0-In CheckerWare assertion like this:

```
/* 0in assert -var (~(~foo & bar))
       -name RTL_C_1 -constraint*/
```

Then we use 0-In to automatically infer the clock and reset for the assertion. We can also use 0-In control files to specify this information.

We can also translate the OVA into an OVL assertion if needed:

```
assert_never
   #(.msg("..."),
     .property_type(‘OVL_ASSUME))
OVL_TEST (
  .clk(‘block_default_clock),
  .reset_n(1’b1),
  .test_expr(~foo & bar)
)
```

In this translation, we have to specify reset and clock explicitly.

B. Constraints Used Inside CEC Flows

As we discussed earlier, it is very helpful to allow designers to specify their CEC constraints, so they can avoid any false non-equivalence. They can also run the model checking tool to prove their CEC constraints right after they pass the CEC check.

This is the methodology we used: for design blocks at the lowest level, designers can provide the CEC constraints. They also need to provide pragmas if they want the constraint to be an assumption instead of an assertion in the model checking tool.

For example, a designer can provide the following constraint in their user input file.

```
//! MC-ASSUME
$constraint ((~foo & bar) == 1’b0);
```

The flow will pick it up, then give it a unique name. CEC will pick up the translated constraint. Based on the pragma, the flow will translate it into an assumption for the model checking tool.

We only allow users to put their CEC constraints in the lowest block level. These constraints will be automatically propagated by the CAD tools to higher levels once the lower level blocks have passed the CEC check.

Users can choose to prove their CEC constraints in the low-level blocks. However, because the low level may not have enough context logic to prove all constraints, running model checking tools at this level is not a tape out requirement. In the ideal scenario, we should run model checking at the highest level, e.g., the full chip level, but that is not possible because of state complexity considerations for model checking tools.

We define a level for our tape-out check where the CEC check must pass without any blackboxing. i.e., we don’t allow constraints hidden inside any blackboxes. Because such hidden constraints will not be used by CEC, therefore won’t be verified by model checking. We want all CEC constraints proven at this tape-out-check level.

In the next section, we will discuss how to prove CEC constraints.

III. Constraint Proving Challenges and Solutions

A. How to Choose Initial State for Model Checking?

Generally, a hardware model checking problem [4, 5, 11] can be expressed as follows: given a hardware model in any design language (e.g., VHDL, Verilog, System C) M, an initial state of M, and a specified property ϕ in a certain temporal logic (e.g., SVA, PSL, etc.), determine whether $M, s_{init} \models \phi$ (i.e., for all possible circuit executions starting from s_{init}, the assertion ϕ holds for the design model M.).

From this description, we can see that the initial state of the design is one of the factors on which the model checking result is built.

In general, the ultimate goal of hardware model checking is to verify the assertion for all ‘possible’ initial states. However, given a sufficiently complex design, such as a full CPU core, finding all the possible initial states is not always practical or even possible because of potential missing execution context.

In our CEC constraint model checking flow, we leverage simulation results created during RTL simulation. At a certain point of the simulation, we ask the simulator to dump out a state for model checking. This approach is very easy to implement inside our verification flow.

However, there are some potential issues using this approach. For example, for our modern microprocessor designs, a universal reset state for all possible executions is not always available. Dumping a state too late could lead to an incomplete proof, if a certain state is only possible early in the simulation, and a real bug can be missed. Dumping a state too early can leave a lot of states uninitialized, which may cause false firings. There is some research on how to get a more general initial state for model checking without specifying a test vector [7]. The authors use symbolic simulation to provide symbolic initial states for model checking, therefore, the model checking covers all possible valid initial states and avoids false firings. This symbolic simulation approach works well for gate-level properties that are related to decoder logic. For CEC constraint model checking, using symbolic simulation only to get good initial states is not enough. Some more advanced techniques are needed to get design constraints/assumptions to avoid false firings due to uninitialized states. This could be a future direction for us.

Knowing the limitations of using a simulation state for initialization, we always ask the RTL verification teams to provide a good initial state that covers the most execution paths of the circuit. When we see model checking disprove an assertion because of uninitialized states (model checking can freely choose 0 or 1 for such states) that take on infeasible values, we will set up the RTL simulator to run more cycles to flush out the uninitialized values. After that, we need to carefully review the constraints to make sure that there is no true firing before this new initial state.

B. How to Bind Constraints/Assertions to Original RTL?

In general, except for these harvestable RTL assertions, we don't want to modify the existing RTL to add CEC constraints. This could lead to un-manageable RTL code with different assertions/constraints.

For example, if we put CEC constraints and related 0-In assertions into the RTL, you will have something like this:

```
‘ifdef CEC
$constraint( $one_hot(B[3:0]));
‘endif CEC

‘ifdef MC
 assert_one_hot #(. )
    FOO (... .text_expr(B[3:0]), ...);
‘endif MC

// or 0in CW
// 0in 0in bits_on -var B[3:0] -max 1 ...
```

In addition to that, whenever a circuit designer identifies a CEC constraint, RTL writers need to maintain multiple equivalent assertions manually, which is very error-prone. In the AMD CAD flow, we maintain implementation-related assertions outside of the RTL. Conformal LEC constraints are stored in CEC working directories. These Conformal LEC constraints are either harvested and translated from RTL tagged assertions automatically by the CAD tools or manually provided by circuit designers. All 0-In model checking assertions/assumptions are translated automatically by the CAD tools.

Therefore, it takes very little effort to maintain these CEC constraints. For each step, the flow will save log files and results so that we can easily retrieve and reproduce any data.

After constraints for each tool are in place, we need to bind them to the design for each tool. For the CEC tool, we use a nice feature in Conformal LEC, "$append\_to$". So we can append LEC constraints to any RTL module. e.g., this will append some LEC constraints to module FOO. It works the same as putting these LEC constraints directly into the RTL source code of model FOO.

```
append_to module FOO;
...
$constraint( ...
...
endmodule
```

For Mentor 0-In, if we use 0-In CheckerWare as the assertion language, we can put assertions into 0-In user control files and specify on which module each assertion is applied.

```
module checker_control;
...
//! MC-PROVE
/* 0in assert -var ...  -name ...
           -module FOO ... */
...
//! MC-ASSUME
/* 0in assert -var ...  -name ...
           -module BAR ... -constraint */
...
endmodule
```

For projects that use OVL as the assertion specification library, binding OVLs to a design is not simple. We can use the System Verilog "bind" statement, but that will still involve a lot of effort to bind ports and signals. The best solution that we have found is to use the PSL flavor of OVL. Then, we can use the PSL vunit assertion binding solution:

```
vunit formal_assertions_foo (FOO) {
assert_... #(.property_type(‘OVL_ASSERT)
         (... .text_expr( ... ), ...);
}

vunit formal_assertions_bar (BAR) {
assert_... #(.property_type(‘OVL_ASSUME)
         (... .text_expr( ... ), ...);
}
```

This PSL code binds one assertion to module FOO and one assumption to module BAR.

Using this solution, we can always have constraints in a separate file, so we don't need to touch the RTL inside the design flow. This also makes constraint management much easier.

C. How to Reduce the Prove Space for Tough CEC Constraints?

Sometimes the model checking tool will give an inconclusive answer for an assertion. We name such assertions "aborts". After investigating what caused the aborts, we can use some techniques to attack them by reducing the prove state space.

1) Re-write Formal-friendly Models: Usually, model checking tools expect synthesizable RTL. There still may be certain structures such as glitching latch models which may cause issues. Abstraction techniques can sometimes increase the state space of the formal prove model, which can cause some assertions to abort. In these cases, formal friendly models may need to be created or advanced tool features used to resolve the aborts to a conclusive result.

2) Deal with Clock Gaters: Model checking tools prove assertions in a certain clock domain. Signals of each assertion need to be driven (controlled) by a clock. Our designs use aggressive power-save techniques, e.g., extensive use of clock gaters to enable/disable clocks to save power. Formal analysis state space may increase very fast by trying to explore the exponential state points of these gater states. Almost all of our CEC constraints don't rely on these gaters. i.e., they should be true if gaters are all on. So if we see aborts on assertions that are gated by many clock gaters, we need to locate the gating chains and turn them on. This will guide the module checking tool to explore a much smaller state space.

3) Provide Assumptions for Model Checking: Sometimes, 0-In formal will fire certain assertions due to the lack of port constraints. For example, certain signals (reset, enable, etc.) are constants (or should satisfy certain constraints) all the way through the whole formal run, but we didn't put assumptions for them. 0-In will try all possible values for these ports, which can cause assertions to abort. Common techniques for these are to identify these ports with RTL designers and set a constraint/constant on them.

If we find certain aborted assertions that are related to large logic cones, we may need to provide 0-In formal more information to quickly prove these assertions. We can put more complicated assumptions on ports to provide a smaller prove state space. Since such assumptions cannot be proven at the current design level with model checking, we ask our RTL writers to write RTL assertions and tag them as "//! MC_ASSUME". i.e., this is a harvestable RTL assertion. Then, RTL simulation will verify them.

4) Use Simulation-based Tools to Verify: There exist some assertions for which we cannot easily find port assumptions to prove them. Model checking spends days with no progress. For such assertions, we use simulation-based engines to find counter examples. e.g., 0in_confirm in the 0-In tool suite is one of our choices. We use 0in_confirm to run fast simulation with constrained random simulation. If we can find a counter example, then we disprove the assertion. If 0in_confirm still cannot disprove the assertion, and we have tried all the above solutions with no luck, then the last solution we have is to ask our RTL writers to put this assertion in RTL and let RTL simulation verify it. By using this approach, we have tried all we can do to find a counter example to cause this assertion to fire.

D. How to Handle Limitations of Model Checking Tools?

Model checking, which works on an NP-complete space, has its fundamental limitations. When the technique tries to formally prove properties by exhaustively exploring the state space, it is limited by the exponential size of the state space. In our hardware model checking, we cannot apply model checking on our whole CPU design. We have to find the best level to run model checking. This level cannot be too small, to prove anything would require huge efforts to write design assumptions, and this level cannot too big, to compile a formal model may be prohibitive. Finding a good design level is important. We run 0-In formal at our tape out check level, which requires the block to pass both CEC and model checking.

Another limitation is non-formal friendly structures. For certain RTL, we could have some arithmetic operators (e.g., multiplication), that could cause state explosion for the model checking tools. Fortunately, at the gate-level, these difficult-to-understand structures are gone. We can prove these assertions at the gate level. Since we have proven the equivalence of the gate-level to the RTL by using the CEC tool, the formal proofs at gate level should hold for RTL as well. After CEC is done, we have the complete mappings of state points between the RTL and gate design. Then, we can translate RTL assertions into their gate-level representations. Some signals of these assertions are nets instead of state points. The CEC tool doesn't have mappings for non-state points. What we do is to force these nets as primary outputs by using Conformal LEC commands. Then, ask our gate designers (or synthesis tools) to have these nets implemented at the gate level circuit. Therefore, we have mappings for these forced primary outputs by using CEC tools. In order to prove these assertions at the gate-level circuit, we also need to read in an RTL simulation dump file and get initial values for these gate state points (e.g., we start from an RTL FSDB dump file, then translate it to a VCD file). By using the CEC mappings, we parse the VCD file for state points and inputs, and translate each RTL signal to its gate-level corresponding signal with an initial value. While tedious, this process is script-able. Right now, all these steps are done automatically by our CAD tools.

IV. Suggestions to EDA Companies

The flow (harvesting CEC constraints, using them inside equivalence checking, and proving them by model checking) is extremely complicated. We do our best to automate each step to avoid any user errors. However, we believe with the help of EDA tools, there is a much simpler solution.

A. Single Assertion Language

When our flow was first built, there were many assertion languages and no industrial standard for all tools. So we have mixed usage of OVA, 0-In CW, and OVL. Now, more and more EDA tools support OVL. However, the support is not mature.

For example, the latest version of Conformal LEC only takes a small set of OVL assertions (e.g., assert_proposition, assert_always). It doesn't support system calls inside the test expressions. Our model checking tool, 0-In, supports OVL but has limited support for system calls in the test expression. Even if we can have an OVL checker that is accepted by RTL simulation, Conformal LEC, and 0-In, we still need to resolve the signals of ports such as clock and reset. In theory, the CEC tool should not care anything about clock and reset ports. The model checking tool should have a way to infer clock and reset without specifying it. (e.g., when a user writes a CEC constraint, there is no clock or reset defined. They are just propositional constraints and should be always true inside its current combinational logic cone – clock or reset should be inferred easily).

We would like to have EDA tools on the same page for OVL checkers [2] that are needed for CEC. We also want to have smarter tools to avoid any translations.

For example, in RTL or our user CEC constraint file, we can have the following OVL constraint:

```
‘ifdef EC
assert_never
   #(.msg("..."),
     .property_type(‘OVL_ASSERT))
OVL_MC_ASSUME (
  .clk(‘CLK),
  .reset_n(‘RESET),
  .test_expr($one_hot(bus_foo))
)
‘endif
```

Then, with EC defined, both RTL simulation and the CEC tool should understand it. When we run model checking, the tool should pick up that assertion as an assumption (we could force certain assertions to assumptions). If CLK or $RESET$ are not inside the current design, the tool should infer them. Therefore, there is no need to translate them for each tool inside the design flow.

B. Easy Assertion Bindings

We like the way Cadence Conformal supports assertion binding by using “append_to module”, which is very handy. There should be a similar simple standard for all tools (if such a standard is not possible in the RTL language). Right now, we are using a PSL “vunit” to bind our OVL assertions, but this is not a good approach because we lose the ability to use any SVA features (e.g., system tasks/calls) inside our assertions.

[2]At AMD, we use less than 6 OVL checkers.

Acknowledgment

The authors thank AMD engineers who use the flows and provide us feedback.

References

[1] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In *Proceedings of TACAS 1999*, pages 193–207, 1999.

[2] R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):667–691, August 1986.

[3] Cadence. Encounter Conformal Equivalence Checking User Guide, Product Version 8.1, June 2009.

[4] Edmund M. Clarke and E. Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In Dexter Kozen, editor, *Workshop on Logics of Programs*, pages 52–71, May 1981.

[5] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.

[6] Xiushan Feng. *Formal Equivalence Checking of Software Specifications vs. Hardware Implementations*. PhD thesis, University of British Columbia, January 2007.

[7] Xiushan Feng, Brian McMinn, Richard Bartolotti, and Mark Eslinger. Using backward symbolic justification to constrain initial state don't-cares in model checking. In *MTV '09: Proceeding of the 10th International Workshop on Microprocessor Test and Verification*, 2009.

[8] Mentor Graphics. Formal Verification User Guide, Software Version 2.6j, October 2009.

[9] Andreas Kuehlmann and Cornelis A. J. van Eijk. Combinational and sequential equivalence checking. In Tsutomu Saso, Soha Hassoun, editor, *Logic Synthesis and Verification*, pages 343–372. Kluwer Academic Publishers, 2002. ISBN:0-7923-7606-4.

[10] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *38th Design Automation Conference*, pages 530–535. ACM/IEEE, 2001.

[11] Jean-Pierre Queille and Joseph Sifakis. Specification and verification of concurrent systems in Cesar. In *5th International Symposium on Programming*, pages 337–351. Springer, 1981.

[12] Erik Seligman and Joonyoung Kim. FEV's greatest bloopers: False positives in formal equivalence. In *DVCon '07: Proceeding of Design Verification Conference*, February 2007.

[13] Fabio Somenzi and Andreas Kuehlmann. Equivalence checking. In Louis Scheffer, Luciano Lavagno, and Grant Martin, editor, *Electronic Design Automation For Integrated Circuits Handbook*. CRC Press, 2006. ISBN:0-8493-3096-3.