LEC: Learning-Driven Data-path Equivalence Checking

Jiang Long*, Robert K. Brayton*, Michael Case†
*EECS Department, UC-Berkeley
{jlong, brayton}@eecs.berkeley.edu
†Calypto Design Systems
{mcase}@calypto.com

***Abstract*— In the LEC system, we employ a learning-driven approach for solving combinational data-path equivalence checking problems. The data-path logic is specified using Boolean and word-level operators in VHDL/Verilog. The targeted application area are C-to-RTL equivalence checking problems found in an industrial setting. These are difficult because of the algebraic transformations done on the data-path logic for highly optimized implementations. Without high level knowledge, existing techniques in bit-level equivalence checking and QF_BV SMT solving are unable to solve these problems effectively. It is crucial to reverse engineer such transformations to bring more similarity between the two sides of the logic. However, it is difficult to extract algebraic logic embedded in a cloud of Boolean and word-level arithmetic operators. To address this, LEC uses a compositional proof methodology and analysis beyond the bit and word level by incorporating algebraic reasoning through polynomial reconstruction. LEC's open architecture allows new solver techniques to be integrated progressively. It builds sub-model trees, recursively transformating the sub-problems to simplify and expose the actual bottleneck arithmetic logic. In addition to rewriting rules that normalize the arithmetic operators, LEC supports conditional rewriting, where the application of a rule is dependent on the existence of invariants in the design itself. LEC utilizes both functional and structural information of the data-path logic to recognize and reconstruct algebraic transformations. A case-study illustrates the steps used to extract the arithmetic embedded in a data-path design as a linear sum of signed integers, and shows the procedures that collaboratively led to a successful compositional proof.**

I. Introduction

With the increasing popularity of high-level design methodologies there is renewed interest in data-path equivalence checking [3][13][18][20]. In such an application, a design prototype is first implemented and validated in C/C++, and then used as the golden specification. A corresponding Verilog/VHDL design is implemented either manually or automatically through high-level synthesis tool [2][4][15]. In both cases, a miter logic for equivalence checking is formed to prove the correctness of the generated RTL model by comparing it against the original C/C++ implementation.

The data-path logic targeted in this paper is specified using Verilog/VHDL. The bit and word-level operators in Verilog/VHDL have the same semantic expressiveness as SMT QF_BV theory[5]. Table I gives a one-to-one correspondence between Verilog and QF_BV unsigned operators. Signed arithmetic operators are also supported. The complexity of such an equivalence problem is NP-complete. However, on the extreme end, the complexity becomes $O(1)$ of the size of the network if the two designs are structurally the same. An NP-complete problem can be tackled by using SAT-solvers as a general procedure. To counter the capacity limitation of SAT-solving, it is crucial to reduce the complexity by identifying internal match points and by conducting transformations to bring in more structural similarity between the two sides of the miter logic.

	Verilog operators	**SMT QF_BV operators**
Boolean	$\&\&, \|\|, !, \oplus, mux$	and, or, not, xor, ite
bit-wise	$\&, \|, \sim, \oplus, mux$	$bvand, bvor, bvnot, bvxor, bvite$
arithmetic	$+, -, *, /, \%$	$bvadd, bvsub, bvmul, bvdiv, bvmod$
extract	$[]$	$extract$
concat	$\{\}$	$concat$
comparator	$<, >, \leq, \geq$	$bvugt, bvult, bvuge, bvule$
shifter	$\ll, \gg$	$bvshl, bvshr$

TABLE I
SUPPORTED OPERATORS (UNSIGNED)

A. Motivation

The differences between the two data-path logics under equivalence checking are introduced by various arithmetic transformations for timing, area and power optimizations. These optimizations are domain specific and can be very specialized towards a particular data-path design and underlying technology. They have the following characteristics:

- The two sides of the miter logic are architecturally different and have no internal match points.
- Many expensive operators such as adders and multipliers are converted to cheaper but more complex implementations and the order of computations are changed. It is not a scalable solution to rely on SAT solving on the bit-blasted model.
- The parts of the transformed portion are embedded in a cloud of bit and word level operators. Algebraic extraction [8][26] of arithmetic logic based on structural patterns is generally too restrictive to handle real-world post-optimization data-path logic.
- Word-level rewriting uses local transformation. Without high-level information, local rewriting is not able to make the two sides of the miter logic structurally more similar.

Lacking high-level knowledge of the data-path logic, the equivalence problems can be very difficult for gate-level equiv-

alence checking and general QF_BV SMT solvers. Strategically, LEC views the bottleneck of such problems as having been introduced by high-level optimizations and employs a collaborative approach to isolate, recognize and reconstruct the high-level transformations to simplify the miter model by bringing in more structural similarities.

B. Contributions

The LEC system incorporates compositional proof strategies, uses rewriting to normalize arithmetic operators, and conducts analysis beyond bit and word level. The collaborating procedures help to expose the actual bottleneck in a proof of equivalence. The novel aspects of this system are:

1) It uses global algebraic reasoning through polynomial reconstruction. In the case-study, it uses the functional information of the design to reverse engineer the arithmetic expression as a linear sum and also uses a structural skeleton of the original data-path to achieve the equivalence proof.
2) It supports conditional rewriting and proves required invariants as pre-conditions.
3) It uses recursive transformations that target making both sides of the miter logic structurally more similar and hence more amenable to bit-level SAT sweeping.
4) It has an open architecture, allowing new solver techniques to be integrated progressively.

Through a case study, we demonstrate the steps that were used to reconstruct the arithmetic embedded in a data-path design as a linear sum of signed integers, as well as all the procedures that compositionally led to a successful equivalence proof. The experimental results demonstrate the effectiveness of these collaborating procedures.

C. Overview

The overall tool flow is described in Section II. Learning techniques and system integration are presented in Section III and IV. A case study is presented in Section V. Experimental results is presented in Section VI followed by a comparison with related work and conclusion.

II. Tool flow

LEC takes Verilog/VHDL as the input language for the data-path logic under comparison. Internally, a miter network, as in Figure 2(a), is constructed comparing combinational logic functions F and G. Figure 1 illustrates the overall tool flow.

First, the Verific RTL parser front-end[6] is used to compile input RTL into the Verific Netlist Database. VeriABC[23] processes the Verific netlist, flattens the hierarchy and produces an intermediate DAG representation in static single assignment (SSA) form, consisting of Boolean and word-level operators as shown in Table I. Except for the hierarchical information, the SSA is a close-to-verbatim representation of the original RTL description. From SSA, a bit-blasting procedure generates a corresponding bit-level network as an AIG (And-inverter graph). Word-level simulation models can be created at the SSA level. ABC[1] equivalence checking solvers are integrated as external solvers.

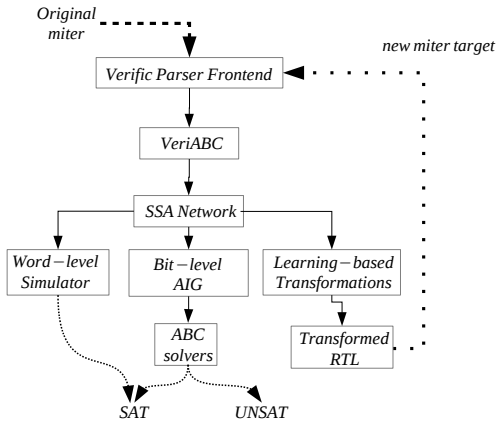

Fig. 1. Overall tool flow

LEC tries to solve the miter directly using random simulation on the word-level simulator or by ABC[1]'s equivalence checking procedure $dcec$, which is a re-implementation of $iprove$[24]. If unresolved, LEC applies transformations to the SSA and produces sub-models in Verilog miter format from which LEC can be recursively applied. The overall system integration is described in Section IV.

III. Learning techniques

In this section, we present the techniques implemented in LEC. Even though some are simple and intuitive, they are powerful when integrated together as demonstrated in the experimental results. All techniques are essential bdecause LEC may not achieve a final proof if any one is omitted. Their interactions are illustrated in the case-study in Section V.

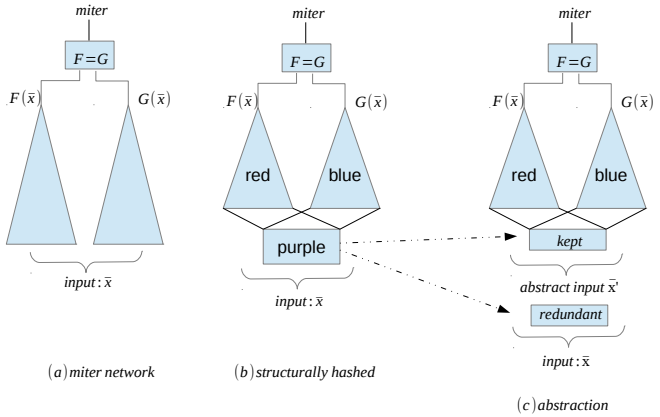

Fig. 2. Miter network

A. Structural information

An SSA netlist is a DAG of bit and word-level operators annotated with bit-width and sign information. In the tool flow, both Verific and VeriABC perform simple structural hashing at the SSA level, merging common sub-expressions. After merging, the miter logic is divided into three colored regions using cone of influence (COI) relations, as in Figure 2 (b).

- Red: if the node is in the COI of F only

- Blue: if the node is in the COI of G only
- Purple: the node is in the COI of both sides of the miter i.e. common logic

The purple region is the portion of the miter logic that has been proved equivalent already, while the red and blue regions are the unresolved ones. LEC makes progress by reducing the red/blue regions and increasing the purple region. The common logic constrains the logic for the red and blue regions, which may be abstracted (see Section III-E) to reduce redundancy and possibly expose the real bottleneck in a proof.

B. Simulation model

Two word-level simulators are generated from the SSA network. One is a native interpreted model. The other uses the open-source Verilator[29] for compiled simulation. From the SSA network, LEC automatically generates C++ code for pseudo-random input drivers and for monitoring design behavior. Verilator compiles the Verilog miter logic, links in the generated C++ code and produces a simulator as a standalone executable. Efficient and effective simulation is crucial in our current flow in capturing potential constants and potential internal equivalent points at the SSA level. Simulation is also used to reduce common logic in the abstraction computation procedure.

C. Bit-level model

As shown in Figure 1, an AIG is created from the SSA network by bit-blasting. LEC calls ABC[1]'s SAT sweeping procedure $dcec$ to perform direct solving at the bit level. Using the AIG model, the native SAT solver embedded in LEC can be used to obtain a formal proof for a particular query. Typical queries are for extracting constant nodes, proving suspected equivalent pairs of points or conducting particular learning for rewriting. Book-keeping information between the SSA nodes and the AIG nodes allows queries to be constructed at the word-level and verified at the bit-level. The result is then used to simplify the SSA network.

D. Constants and Potential Equivalent Points (PEPs)

At the word-level, candidates for constants and PEPs are identified through simulation and SAT queries are posed. Each such SAT query is constructed and checked at the bit-level. SAT-solving is configured at a low-effort level (run for a few seconds) for these types of queries. Proven constants and PEPs are used immediately to simplify the SSA network, leading to a new sub-model of less complexity. LEC then continues to process the sub-model. In the presence of unproven PEPs, LEC can choose one as the next miter target, normally the smallest in terms of the number of nodes in its COI. The proof progresses as constants and PEPs are identified and used to simplify the miter model.

E. Abstraction

As illustrated in Figure 2 (c), in computing an abstraction, LEC computes a cut in the purple region (common logic), and removes the logic between the cut and the inputs. An abstract model is formed by replacing the cut signals with free inputs $\bar{x'}$. If this abstracted miter is UNSAT, then the original miter is UNSAT. In our current implementation, LEC traverses the SSA network in topological order from the inputs. As each node is tentatively replaced with new PIs, simulation is used to validate the replacement. If successful, the node is omitted and replaced with the new PIs and the next node is processed similarly.

A successful abstraction step removes irrelevant logic and exposes a smaller unresolved region of the miter logic, allowing LEC to continue using other procedures. In addition, as seen from experimental results, the reduction of common logic can reduce significantly the amount of complexity for downstream SAT-solving, e.g. when common multipliers being removed from the miter logic. An unsuccessful abstraction when the abstract miter becomes SAT, indicates the existence of a rare event not being captured during random simulations. Often, this gives hints for selecting case-splitting candidates.

F. Rewriting

Similar to [20], word-level rewriting transforms an SSA network into a structurally different but functionally equivalent one. Through rewriting, certain equivalence checking problems can become much simpler. In our experience, a multiplier is often a source of difficulty in data-path equivalence checking. If two multipliers from opposite sides of the miter are matched exactly, LEC can simplify the miter through structural hashing and treat them as common logic. This is most effective when combined with the abstraction procedure as the common multiplier can now be totally removed.

In LEC, a few rules are hard-coded through pattern matching applied to the SSA network. The goal is to process multiplications so that they can be matched exactly. This rewriting is implementation specific; for illustration purposes, we list a few rewriting rules in Table II using Verilog notation and the semantics of the operators.

The first rule is the normalization of multiplier operands. If a multiplier uses a partial product generator and a compressor tree, switching the operands of the multiplication becomes a very hard SAT problem because at the bit level the implementation is not symmetrical. It is almost imperative to apply this rule whenever possible. The second and third rules use the distributive laws of multiplication and multiplexing. Rules 4 and 5 remove the shift operator $\gg$ when it is used with $extract$ and $concat$ because it is hard for multiplication to be restructured through the $\gg$ operator. Rule 6 distributes multiplication through the $concat$ of two bit vectors using $+$. It uses the fact that the concatenation $\{a, b[n-1:0]\}$ is equivalent to $a * 2^n + b[n-1:0]$.

The following is a more complex rule that distributes $+$ over the $extract$ operator. The right hand side is corrected with a third term, which is the carry bit from adding the lower n bits of a and b.

$$\begin{aligned}(a+b)[m:n] = \\ a[m:n] + b[m:n] + (a[n-1:0] + b[n-1:0])[n]\end{aligned} \quad (1)$$

	Before	After
1	$a * b$	$b * a$
2	$mux(cond, d0, d1) * c$	$mux(cond, d0 * c, d1 * c)$
3	$mux(cond, d0, d1)[m:n]$	$mux(cond, d0[m:n], d1[m:n])$
4	$a[m:0] \gg n$	{ (m-n)'b0, a[m:n] }
5	$(a[m:0] \gg n)[m-n:0]$	$a[m:n]$
6	$\{a, b[n-1:0]\} * c$	$a * c \ll n + b[n-1:0] * c$

TABLE II
REWRITING RULES

Repeatedly applying the above rules, LEC transforms the SSA network and keeps only the $*$ and $+$ operators, enhancing the possibility of multipliers to be matched. Note that the above rule (1) and Rule 4-6 in Table II are correct for unsigned operators. Currently, for signed operators, due to sign extension and the two's complement representation of the operands, we have not implemented a good set of rewriting rules.

1) Conditional rewriting: The following equation

$$(a \ll c) * b = (a * b) \ll c \tag{2}$$

reduces the bit-width of a multiplier on the left hand side to a smaller one on the right. It is correct if a, b, c are integers but incorrect in Verilog semantics, which uses modulo integer arithmetic. However, if the following is true within the miter model in modulo integer semantics

$$(\,(a \ll c) \gg c) \;== a \tag{3}$$

then equation (2) is valid. In such a situation, LEC identifies the pattern on the left hand side of (2) in the SSA network and executes a SAT query concerning (3) using the AIG model through bit-level solvers. The transformation to the left hand side of (2) is carried out only if the query is proven to be an invariant. Such a transformation may produce an exact match of $a*b$ afterwards, which can be crucial for achieving the final proof.

G. Case-split

Case-splitting on a binary signal, cofactors the original model into two sub-models. The miter is proven if both sub-models are proven, or falsified if any sub-model is falsified. Although exponential in nature, if many signals are chosen, case-splitting can simplify the underlying bit-level SAT solving significantly. For example, it is difficult to prove the following miter structure directly through bit-blasting and SAT solving at the AIG level

$$(x + y) * (x + y) == x * x + 2 * x * y + y * y \tag{4}$$

where x is a 32-bit integer and y a single binary signal. However, it can be proven easily if case-splitting is done on $y = 0$ and $y = 1$. After constant propagation, the bit-level solver can prove both sub-models easily.

The current case-splitting mechanism supports cofactoring on an input bit or input bit-vector. In verifying the test cases experienced so far, the case splits are conducted on a bit, a bit-vector equal to zero or not, or on the lsb or msb of a bit-vector equals to zero or not. A heuristic procedure can be implemented to trace back from the sel port of a mux node through its Boolean fanins and choose the candidates that have the highest controllability.

Another advantage of case-splitting is that the co-factored sub-models contain new candidates for constants and PEPs, which lead to other down-stream transformations not possible before. Case-splitting also reduces the amount of Boolean logic in the SSA network and exposes the data-path logic to high-level learning such as polynomial construction.

H. Polynomial construction

Reasoning at the word-level, rewriting rules are based on the arithmetic properties of the corresponding operators such as the commutative law of integer multiplication. However, rewriting applies only local transformations and does not have a global view. In situations when the miter logic is constructed from arithmetic optimization at the polynomial level, local rewriting is not able to bring similarity into the miter for further simplification. In such a situation, LEC tries to reconstruct the polynomial of the whole miter model to establish equivalence through arithmetic or algebraic equivalences and then use high level transformations to prove the equivalence of the original miter.

As a generic procedure, LEC follows four steps to prove a miter network $F(\bar{x}) = G(\bar{x})$ where F and G are the top level signals being compared, and $\bar{x}$ is the vector of input variables (bit-vectors):

1) Conjecture (possibly by design knowledge) about the algebraic domain of the polynomial, e.g. signed vs. unsigned integer, modulo integer arithmetic, the order of the polynomial etc. These conjectures set up the framework and semantics for polynomial reconstruction as illustrated in the case-study of Section V.
2) Determine a polynomial f and create a logic network F' such that the following can be proved formally.

$$F' \; implements \; f \tag{5}$$
$$miter \quad F' = F \tag{6}$$

   How f is constructed is domain and test-case dependent. In the case-study of Section V, we use simulation patterns to probe for the coefficients of a linear function.
3) Determine a polynomial g and create a logic network G' such that the following can be proved formally.

$$G' \; implements \; g \tag{7}$$
$$miter \quad G' = G \tag{8}$$

4) Establish the following equivalence formally at the algebraic level.

$$f = g \tag{9}$$

The combination of Items 2, 3, and 4 establishes the equivalence proof of the original miter model $F = G$. In constructing F' and G', we try to make them as structurally similar to F and G as possible. Details are given in Section V.

IV. System Integration

The above learning techniques are integrated in LEC as a set of logically independent procedures. Each procedure produces one or more sub-models, illustrated as a tree in Figure 3. The root node is the current targeted Verilog miter model. It has eight children. The $simulator$ and AIG models are the ones described in Figure 1. The $simplified$ sub-model is generated by constant propagation and merging proven PEPs. The $abstraction$ and $rewrite$ sub-models are created by the abstraction and rewrite procedures in the previous section. The $case\text{-}split$ sub-model consists of a set of sub-models, corresponding to the cofactoring variables selected. In the current implementation, the user needs to input the set of signals to case-split on; eventually they will be selected by heuristics. The $linear\text{-}construction$ node has two sub-models which will be explained in detail in Section V. When PEPs are identified through simulation, a PEP node is create with the set of unproven-PEPs as sub-models.

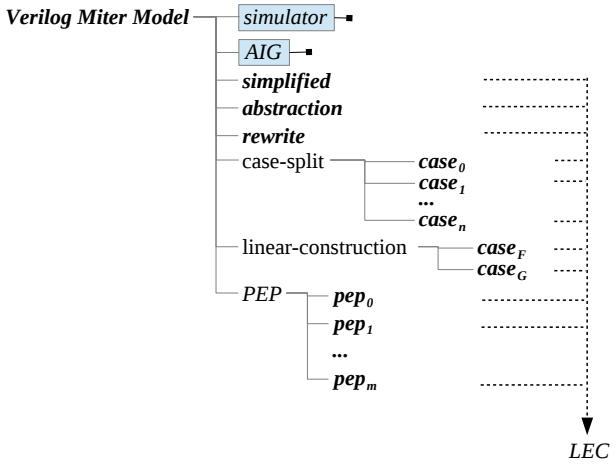

Fig. 3. Branching sub-model tree

Two nodes in the sub-model tree are terminal. One is the simulator model which can falsify the miter through random simulation. The other is the AIG model where ABC's bit-level $dcec$ procedure is applied. The rest of the leaf models (in bold font) are generated as Verilog miter models, which have the same format as the root node. LEC procedures can be applied recursively to these leaf nodes to extend the sub-model trees to simpler ones. The LEC proof process progresses by expanding the sub-model tree. A sub-model is resolved as SAT or UNSAT from its sub-models' proof results.

Since there are no logical dependencies between sibling sub-models, any branch can be chosen to continue the proof process. Sibling sub-models can be forked in parallel from a parent process. A node in the sub-model tree determines its proof result from its children. Table III gives the possible return values from the first level sub-models. SIMPLIFY is returned by a PEP node to its parent model when at least one of its sub-models, pep_i, is proven UNSAT, notifying the parent node to simplify further with the newly proved pep_i.

Depending on the logical relationships between a parent and its immediate sub-models, a node is either disjunctive or conjunctive in semantics. In Figure 3, a Verilog miter model node is disjunctive, which includes the root and all the leaf nodes in bold font. The $case\text{-}split$ and $linear\ construction$ nodes are conjunctive; a PEP node is disjunctive. The semantics, shown in the following tables, are used to resolve the proof result of the parent model from its immediate sub-models. To complete the calculus, we introduced two values: CON and BOT, where CON stands for an internal conflict indicating a potential LEC software bug and BOT is the bottom of the value lattice and acts like an uninitialized value.

Sub-model	Return
simulator	SAT
AIG	SAT/UNSAT
simplified	SAT/UNSAT
abstraction	UNSAT
rewrite	SAT/UNSAT
case-split	SAT/UNSAT
linear construction	SAT/UNSAT
PEP	SIMPLIFY

TABLE III
Sub model return value

‖	SAT	UNS	UNK	SMP	CON	BOT
SAT	SAT	CON	SAT	SAT	CON	SAT
UNS	CON	UNS	UNS	UNS	CON	UNS
UNK	SAT	UNS	UNK	SMP	CON	UNK
SMP	SAT	UNS	SMP	SMP	CON	SMP
CON	CON	CON	CON	CON	CON	CON
BOT	SAT	UNS	UNK	SMP	CON	BOT

TABLE IV
Disjunctions of models

&	SAT	UNS	UNK	SMP	CON	BOT
SAT	SAT	SAT	SAT	n/a	CON	SAT
UNS	SAT	UNS	UNK	n/a	CON	UNS
UNK	SAT	UNK	UNK	n/a	CON	UNK
SMP	n/a	n/a	n/a	n/a	n/a	n/a
CON	CON	CON	CON	n/a	CON	CON
BOT	SAT	UNS	UNK	n/a	CON	BOT

TABLE V
Conjunction of models

Tables IV and V are the truth tables for the disjunction and conjunction semantics of the return values, in which UNS, UNK, SMP stand for UNSAT, UNKNOWN, and SIMPLIFY. Assuming a bug free situation, at a disjunctive node, if either SAT or UNSAT is returned from a sub-model, this is the final proof result for the parent. In conjunction, the parent must wait until all sub-models are resolved as UNSAT before deciding that its result is UNSAT, while any SAT sub-model implies the current model is SAT. A PEP node returns SIMPLIFY to its parent if one of its sub-models, say pep_i, is proven UNSAT. In this case, the parent model can apply another round of simplification to obtain a new $simplified$ sub-model by merging the node pair in the just-proved pep_i. The proof log in Figure VI is a sample sub-model tree where only the branches that contributed to the final proof are shown. Indentation indicates the parent-child relationship. Recursively, the proof result of the top level target is evaluated as UNSAT.

```
{
  "case split": {
    "case_0": "UNSAT by AIG"
    "case_1": {
      "simplified": {
        "abstraction": {
          "case split": {
            "case_00": "UNSAT by AIG",
            "case_01": "UNSAT by AIG",
            "case_10": "UNSAT by AIG",
            "case_11": "UNSAT by AIG"
          },
        },
      },
    },
  },
}
-------------------------------------
Miter proof result: [Resolved: UNSAT]
-------------------------------------
```

Fig. 4. Illustration of proof log

Using this sub-model tree infrastructure, any new procedures discovered in the future can be plugged into the system easily. Also, the system is fully parallelizable in that siblings can be executed at the same time. The proof process can be retrieved from the expanded sub-model tree.

V. Case Study

The design in this case-study is an industrial example taken from the image processing domain. We verify specification = implementation where the "specification" is a manually-specified high-level description of the design. "Implementation" is a machine-generated and highly optimized RTL implementation of the same design using[2]. The miter logic is obtained through SLEC[3]. Therefore, the miter problem is verifying that the high-level synthesis (HLS) tool did not modify the design behavior.

This miter is sequential in nature, but here we examine a bounded model checking (BMC) problem which checks the correctness of the implementation at cycle N. This renders the problem combinational. This is industrially relevant because the sequential problem is too hard to solve in general, and even the BMC problem at cycle N becomes too difficult for industrial tools.

The original design (specification) consists of 150 lines of C++. It went through the Calypto frontend[3] and was synthesized into a word-level netlist in Verilog. The generated miter model has 1090 lines of structural Verilog code with 36 input ports: 29 of which are 7 bits wide, 2 are 9 bits, 4 are 28 bits and one is a single-bit wire. The miter is comparing two 28-bit values. We do not have knowledge about what the design does except through structural statistics: no multipliers, many adders, subtractors, comparators, shifters etc., together with Boolean logic. From a schematic produced from the Verilog, there seems to be a sorting network implemented using comparators, but we can not tell anything further.

Figure 5 illustrates the compositional proof produced by the LEC system by showing the sub-model tree created during the proof process. Indentations indicate parent and sub-model relations and are listed in the order they were created. The three numbers on the right are the node counts in the red, blue and purple regions (common logic) of the SSA network as distinguished in Figure 2(a). Only those sub-models that contributed to the final proof are shown in the figure. Others are ignored. As seen in Figure 5, the case-split procedure is

```
1.original model              :  366  332 776
2.  case-split
3.    case_0                  :  366  331 844
4.      AIG                   :  UNSAT
6.    case_1                  :  366  332 776
7.      simplified            :  344  289 675
8.        abstraction         :  344  289  29
9.          case-split
10.           case_0          :  344  289  31
11.             AIG           :  UNSAT
12.           case_1          :  344  289  31
13.             simplified    :  343  288  27
14.               PEP
15.                 pep_0     :  335  280  27
16.                   linear construction
17.                     case_F
18.                       AIG :  UNSAT
19.                     case_G
20.                       AIG :  UNSAT
21.               simplified  :  10  10  305
22.                 AIG       :  UNSAT
```

Fig. 5. Sub-model proof tree

applied twice, at lines 2 and 9. Both models have a single-bit input port, which was selected for cofactoring. ABC[1] immediately proved the first cofactored case, case_0 (3 and 10) , using the AIG model at 4 and 11. The time-out for the $dcec$ run was set to two seconds. Abstraction was applied at 8, significantly reducing the common logic from 675 to 29 SSA nodes, and effectively removing all the comparator logic. We tried abstraction on the original model without the $case\text{-}split$ procedure and it failed to produce any result. The $case\text{-}split$ at 2 removed enough Boolean logic and eliminated some corner cases such that the abstraction procedure was able to produce an abstract model successfully.

Model 15 is the smallest unproved PEP from model 13. It is proved using the linear construction procedure at 16, which we shall describe in detail in Section V-A. Model 21 is the simplified model of model 13 after merging the just-proved pep_0. After simplification, most of the logic in model 21 became common logic through structural hashing, leaving only 10 nodes in each of the blue and red regions. Model 21 was proved quickly by ABC which concludes the proof of the original miter. In this case, the linear-construction procedure was crucial in attaining the proof. However, the case-split, simplification, abstraction, and PEP models also are very important because they collaborate in removing Boolean, mux and comparator logic etc, but keeping only the part of the original miter logic which constitutes a linear function. Only at this point, can a proof by the linear construction procedure succeed.

A. Linear construction

For model 15 in Figure 5, the SSA network contains many $+$,$-$, $\ll$ and $\gg$ operators along with $extract$ and $concat$ oper-

ators, but contains no Boolean operators or muxes. The input ports consist of twenty-five 7-bit or 12-bit wide ports. The miter is comparing two 15-bit wide output ports. At this point, simplification and abstraction can not simplify the model further. Also, there are no good candidates for case-splitting. The local rewriting rules can not be applied effectively without having some global information to help converge the two sides of the miter logic. High-level information must be extracted and applied to prove this miter model.

After the linear construction procedure through LEC, the miter logic is found to be implementing the following linear sum in the signed integer domain using two's complement representation:

$$\begin{aligned} -16 * x_0 + 2 * x_1 + 2 * x_2 + 2 * x_3 + 2 * x_4 + 2 * x_5 \\ +2 * x_6 + 2 * x_7 + 2 * x_7 + 2 * x_8 + 2 * x_9 \\ +2 * x_{10} + x_{11} + x_{12} + 2 * x_{13} + 2 * x_{14} + 2 * x_{15} \\ +2 * x_{16} + 2 * x_{17} + 2 * x_{18} + 2 * x_{19} - 2 * x_{20} + 2 * x_{21} \\ +2 * x_{22} + 2 * x_{23} + 2 * x_{24} + 14 \end{aligned}$$

One side of the miter implements the above sum as a plain linear adder chain (Figure 6(a)), the other side is a highly optimized implementation using a balanced binary tree structure (Figure 6(b)) and optimization tricks, which we don't fully understand. This is a hard problem for bit-level engines because

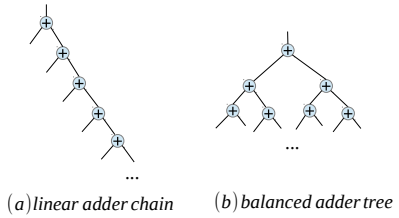

Fig. 6. Addition implementation

there are no internal match points to utilize. Therefore, LEC resorts to trying a high-level method to establish equivalence at the polynomial level. The following are the detailed steps for this specific case.

1) The conjecture: Assume the miter logic is $F(\bar{x}) = G(\bar{x})$ as in Figure 2(a). LEC conjectures the following for the arithmetic domain.

- Signed integer arithmetic. The numbers are in 2's complement representation.
- Assume $F(\bar{x})$ and $G(\bar{x})$ are implementing the linear sums f and g of the forms

$$f(\bar{x}) = \sum a_i \cdot x_i + b \tag{10}$$

$$g(\bar{x}) = \sum a'_i \cdot x_i + b' \tag{11}$$

2) Determining the coefficients of f, g and proving $f = g$ algebraically: Given the data-path logic $F(\bar{x})$ and the linear sum formula (10), it takes $n+1$ simulation patterns on the n input variables to compute the coefficients:

$$\begin{aligned} b &= F(0, 0, ..., 0) \\ a_0 &= F(1, 0, ..., 0) - b \\ a_1 &= F(0, 1, ..., 0) - b \\ &... \\ a_{n-1} &= F(0, 0, ..., 1) - b \end{aligned}$$

Another round of random simulation on both the logic and the polynomial can be done to increase the likelihood of the conjecture. The same is repeated for $G(\bar{x})$ to obtain $g(\bar{x})$.

In integer arithmetic, f is equal to g if and only if the coefficients match exactly for each term:

$$f = g <=> \forall\, i\ a_i = a'_i \quad and \quad b = b' \tag{12}$$

So checking of $f = g$ is trivial in this case. In other algebraic domains, domain specific reasoning may have to be applied to derive algebraic equivalence e.g. in [28].

3) Synthesizing implementations F'/G' for f (f=g), structurally similar to F/G: We want to find a Verilog implementation $F'(\bar{x})$ of f such that

1) F' implements f
2) F' is structurally similar to F

To do this, all nodes in the SSA network with arithmetic operators $+$, $-$ are marked, and edges connecting single bits are removed. A reduced graph is then created from the marked nodes in the remaining graph maintaining the input/output relations between marked nodes. This graph is a skeleton of the implementation structure of F. For each of its nodes, we annotate it with a conjectured linear sum computed in the same way as in the above steps. The root node F is annotated with f and internal nodes annotated with linear sums f_s, f_t, etc. For illustration purposes, Figure 7(a) shows such an annotated reduced graph for node w. For an arbitrary node w

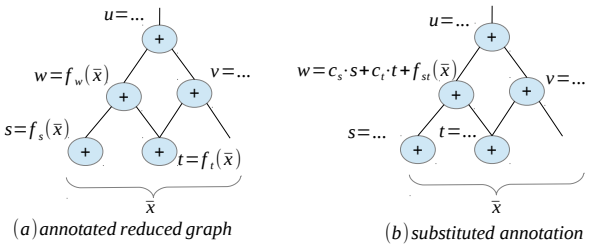

Fig. 7. Annotated reduced graph

in the reduced graph with inputs from nodes s and t, from the annotation we have the following:

$$\begin{aligned} s &= f_s(\bar{x}) \\ t &= f_t(\bar{x}) \\ w &= f_w(\bar{x}) \end{aligned}$$

We would like to substitute f_w with variable s and t, such that w is a function of s and t in order to follow the structure of the skeleton reduced graph. Because all the functions are linear sums, we can compute, using algebraic division, two constants c_s and c_t such that the following holds:

$$w = c_s \cdot s + c_t \cdot t + f_{st}(\bar{x})$$

c_s is the quotient of f_w/f_s and $c_t = (f_w - c_s \cdot f_s)/f_t$, while f_{st} is the remainder of the previous division. The substitution is conducted in one topological traversal from the inputs to the miter output. After substitution, the annotated reduced graph is essentially a multi-level implementation of the above linear sum. Because the linear sum annotated at each node in the reduced graph is only a conjecture, it may not be exactly the same function as in the original miter logic. However, in re-implementing f using this structure, certain similarities are still captured through the construction process.

This multi-level circuit is implemented by traversing the substituted reduced graph, and creating a corresponding Verilog file for F'. It is generated by allocating a bit-vector at each internal node with its bit-width equivalent to output port of F. The same can be done for G to obtain G'. The reason why LEC goes through so much trouble to obtain separate RTL implementations for F' and G' (even though they are both implementing f), is that we need to prove $F = F'$ and $G = G'$ separately next. Without the structural similarities created through this procedure, proving equivalence with an arbitrary F' and G' would be as difficult as proving the original $F = G$. Generally, only with these extra similarities injected, can the miter model be simplified enough to allow SAT sweeping to succeed for $F = F'$ and $G = G'$.

4) Proving $F = F'$ and $G = G'$: We construct two miter models $F = F'$ and $G = G'$ as $case_F$ and $case_G$ in Figure 3, and apply LEC separately to each. By construction, each miter should be simpler than the original $F = G$ because of the increased structural similarity between the two sides of the miter. Another round of LEC might reduce this miter logic, if not prove it directly through bit-level solvers. In the present case, $F = F'$ was proven instantly because F is a simple linear adder chain, and so is F'. Proving $G = G'$ takes more time using ABCs $dcec$ because G is a highly optimized implementation of f and the multi-level implementation from the annotated reduced graph only captures part of the similarity. But, the injected similarity was sufficient enough to reduce the complexity to be within $dcec$'s capacity.

5) Proving that F'/G' implements f/g : To complete the proof, we still have the proof obligation that F' and G' implement f and g respectively. By construction from the reduced graph, the generated Verilog is a verbatim translation from the multi-level form of f. However, we need to bridge the gap between Verilog's bit-vector arithmetic vs. the integer arithmetic of the linear sum. To do so, we created SVA assertions to check that every Verilog statement captures the integer value in full without losing precision due to underflows or overflows.

$$\begin{aligned} c[n:0] =& a[n-1:0] + b[n-1:0]; \\ & assert\ (a[n-1]\,\&\,b[n-1] \implies c[n]) \\ & assert\ (!a[n-1]\,\&\,!b[n-1] \implies !c[n]) \end{aligned}$$

$$\begin{aligned} c[n:0] =& a[n:0]/b[m:0]; \\ & assert\ (a[n:0] == (c[n:0] * b[m:0])[n:0]) \end{aligned}$$

$$\begin{aligned} c[m:0] =& \{a[n:0]\}[m:0]; \\ & assert\ a[n:0] == \{(n-m) * \{c[m-1]\}, c[m:0]\} \end{aligned}$$

The first two sets of assertions ensure there is no overflow of signed integer add and no non-zero remainder of division. The third one ensures that extraction does not change the value in two's complement representation. The SVA checkers are formally verified separately using the VeriABC[23] flow, which in turn uses ABCs model checker.

From the above procedures, we established the following:

$$f(\bar{x}) = g(\bar{x}) \tag{13}$$

$$F'(\bar{x})\ implements\ f(\bar{x}) \tag{14}$$

$$G'(\bar{x})\ implements\ g(\bar{x}) \tag{15}$$

$$F = F' \tag{16}$$

$$G = G' \tag{17}$$

Altogether they establish the proof for $F = G$.

Combining the above procedures into a single run, LEC took about 10 minutes on an i7 processor to complete the full proof. Roughly 80% of the time is spent on compiling and running random simulation, the rest are used for SAT solving. In table VI, we also compare this run-time against Boolector[11], z3 [16] and ABC' iprove [24] solver, all run on the same server. It is clear that LEC expedites the proof significantly by using the knowledge of the linear sum formulation inside the miter logic.

Miter	Boolector	Z3	iprove	LEC
model 1	time-out	time-out	time-out	10min

TABLE VI
COMPARISON WITH OTHER SOLVES (TIME-OUT IN 24 HOURS)

In summary, polynomial reconstruction was a key technique to prove the underlying miter problem. The case-study illustrates the major steps and the proof obligations encountered during the process. The actual techniques used for different domains of the underlying arithmetic would differ. Each algebraic domain would require special purpose heuristics and automated proof procedures to guarantee the correctness of the reconstructions and transformations used in the method. The goal of the linear construction procedure was to inject increased structural similarity by using global algebraic transformations.

VI. EXPERIMENTAL RESULTS

Table VI shows the experimental results comparing Boolector[11], Z3[16] and iprove[24] using a 24-hour time-out limit on an 2.6Ghz Intel Xeon processor. These models are generated directly using SLEC[3] for checking C-to-RTL equivalence or extracted as a sub-target from PEPs. The first column is the miter design name. The second column is the number of lines of Verilog for the miter model specification. Run-time or time-out results are reported for each solver in columns 3 to 6. Although the miter models are not big in terms of lines of Verilog, they are quite challenging for Boolector, Z3 and iprove. The run-time of LEC is the total CPU time

including Verilog compilation. It was expected that iprove would not prove any of them because it works on the bit-blasted model without any high-level information that the other solvers have.

Design	Lines	Boolector	z3	iprove	LEC
mul_64_64	125	20 sec	200 sec	timeout	10 sec
d1	24	time-out	time-out	time-out	15 sec
d2	507	time-out	time-out	time-out	2 min
d3	191	time-out	time-out	time-out	15 min
d4	473	time-out	time-out	time-out	60 sec
d5_pep_0	674	time-out	9 hour	time-out	4 min

TABLE VII
BENCHMARK COMPARISON (TIMEOUT 24 HOURS)

The miter, mul_64_64, is comparing a 64x64 multiplier with an implementation using four 32x32 multipliers as the following:

$$\{a_H, a_L\} * \{b_H, b_L\} = (a_H * b_H) << 64 + (a_H * b_L + a_L * b_H) << 32 + a_L * b_L$$

where a_H, a_L, b_H, b_L are the 32-bit slices of the original 64-bit bit-vectors. Both Boolector and Z3 are able to prove it. LEC proves it by first utilizing rewriting rules to transform the 64x64 multiplier into four 32x32 multipliers, matching the other four in the RHS of the miter. As they are matched exactly, they become common logic in the miter model. LEC then produces an abstraction and obtains a reduced model with all the multipliers removed: the outputs of the multipliers become free inputs in the abstract model. The abstract model is then proven instantly by ABC's dcec on the AIG model.

The miter d1, extracted from a PEP sub-model, is a demonstration of rewrite rule 6 in Table II using 32-bit multiplication. As both Boolector and Z3 fail to prove equivalence within the time-limit, they likely do not have this rewriting rule implemented.

To prove d2, LEC conducts conditional rewriting using rule (2) by first statically proving an invariant in the form of (3). After the transformation, the multipliers are matched exactly on both sides of the miter and removed in the subsequent abstract model. The final miter model is proved instantly by ABC on the bit level AIG.

The miter model d3 has part of its logic similar to mul_64_64 embedded inside. LEC proves d3 by first applying rewriting rules repeatedly until no more rewriting is possible. Then, LEC computes a reduced model through abstraction. In the reduced model, LEC conducts a case-split on a one-bit input. The case-0 AIG model is proven instantly, while case-1 is proven in about 10 minutes by ABC.

The miter d4 is proven by first conducting a case-split of two bit-vector inputs: cofactoring on whether the bit-vector equals zero or not. Three of the four cofactored cases are proven instantly. The one unresolved goes through a round of simplification and abstraction. On the then obtained sub-model, three one-bit inputs are identified and cofactored through case-split procedures. LEC prove all eight cases quickly within a few seconds.

Miter d5 is extracted from model 15 in Figure 5 which contains the purely linear sum miter described in the case-study section. For this simpler miter, Z3 is able to prove it in 9 hours while both iprove and Boolector time out. This shows that LEC's transformations through collaborating procedures successfully reduce the surrounding logic in the original model, which was preventing Z3 to prove it in 24 hours.

The above experiments demonstrate the effectiveness of LEC's collaborating procedures of simplification, rewriting, case-splitting and abstraction computations. The LEC architecture allows these procedures to be applied recursively through a sub-model tree: the model obtained by one procedure introduces new opportunities for applying other procedures in the next iteration. As exemplified in miter d4, the initial case-split gives rise to new opportunities for simplification as new constants are introduced by cofactoring. Then a new round of abstraction is able to remove enough common logic and expose three one-bit inputs as case-split candidates in the reduced model, which in turn gives rise to another case-split transformation that leads to the final proof. None of this is possible without the transformations being applied in sequence.

VII. COMPARISON WITH RELATED WORK

In bit-level equivalence-checking procedures [24][25], simulation, SAT-sweeping, AIG rewriting and internal equivalence identification are all relevant to data-path equivalence-checking. In LEC, these types of procedures are conducted at the word-level. Word-level rewriting is difficult if only a bit-level model is available. For example, with no knowledge of the boundary of a multiplier, normalizing its operands is impractical at the bit-level . Although abstraction and case-split techniques in LEC can be applied at the bit-level in theory, these are not used due to the difficulty of computing an abstraction boundary or of finding good cofactoring candidates.

SMT solving is relevant because a data-path is a subset of QF_BV theory. Methods such as [7][11][16][14][17][19], are state-of-art QF_BV solvers. These employ different implementations of word-level techniques in rewriting, abstraction, case-splitting, and simplification, and interleave Boolean and word-level reasoning via a generalized DPLL framework or through abstraction refinements of various forms. Hector[20] is closest to LEC in terms of technology and targeted application domains, and has a rich set of word-level rewriting rules along with some theorem prover [7] procedures to validate every rewriting applied. Hector also has an orchestration of a set of bit-level solvers using SAT and BDD engines to employ once the bit-level miter model is constructed. Strategically, LEC relies less on the capacity of SAT solver; instead it builds a compositional proof infrastructure and employs iterative transformations to finally obtain a proof through sub-model trees. The goal of these LEC learning procedures is to reverse engineer the embedded high-level algebraic transformations and bring more similarity between both sides of the miter model.

The techniques in [26] [31][33] also try to reconstruct an algebraic model from the underlying logic, but they employ a bottom up approach and their primitive element is a half-adder.

The method in [8] simplifies the algebraic construction by solving an integer linear programming problem. The limitation of these approaches is that they rely on the structural pattern of the underlying logic to reconstruct the algebraic model. On the other hand, the linear construction case-study in Section V-A constructs the polynomial through probing with simulation patterns. This is more general as it uses only the functional information of the data-path logic. For different domains, other techniques may well be more applicable such as the bottom-up approach. The use of vanishing polynomials and Grobner bases in [27][28] to prove equivalence between polynomials in the modulo integer domain can be utilized once a polynomial form is reconstructed in LEC. In many data-path miter models, such a polynomial in a certain domain or theory is likely embedded in other control and data-path logic. Direct application of algebraic techniques is often not practical. Thus the collaborating procedures in LEC are designed to bridge this gap and isolate such polynomials so that these high level theories can then be applied.

In conducting consistency checking between C and Verilog RTL, the work [21] focuses on how to process a C program to generate formal models. The tool relies on SMT solvers [11][16][14] as the back-end solving engines.

In terms of tool architecture, [9] [10] [22], all employ a sophisticated set of transformations to simplify the target model during verification. These are done at the bit-level. The LEC infrastructure allows future extension to take advantage of multi-core parallelization as demonstrated in [30]. [12] [32], use a dedicated data-structures to represent the proof-obligations, while LEC relies on the sub-model tree to track the compositional proof strategy used at each node.

VIII. Conclusion

In LEC, we build a system of collaborating procedures for data-path equivalence-checking problems found from an industrial setting. The strategy is to utilize Boolean level solvers, conduct the transformations at the word-level and to synthesize internal similarities by lifting the reasoning to the algebraic level . Using a real industrial case-study, we demonstrated the applicability of the sub-tree infrastructure for integrating a compositional proof methodology using LEC.

References

[1] ABC - a system for sequential synthesis and verification. Berkeley Verification and Synthesis Research Center, http://www.bvsrc.org.

[2] Calypto® Catapult Design Product. http://www.calypto.com.

[3] Calypto® SLEC. http://www.calypto.com.

[4] Forte design systems. http://www.forteds.com.

[5] smtlib. http://www.smt-lib.org.

[6] Verific Design Automation: http://www.verific.com.

[7] C. Barrett, C. L. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. Cvc4. In *Computer Aided Verification*, pages 171–177. Springer, 2011.

[8] M. A. Basith, T. Ahmad, A. Rossi, and M. Ciesielski. Algebraic approach to arithmetic design verification. In *Formal Methods in Computer-Aided Design (FMCAD), 2011*, pages 67–71. IEEE, 2011.

[9] J. Baumgartner, H. Mony, V. Paruthi, R. Kanzelman, and G. Janssen. Scalable sequential equivalence checking across arbitrary design transformations. In *Computer Design, 2006. ICCD 2006. International Conference on*, pages 259–266. IEEE, 2007.

[10] R. Brayton and A. Mishchenko. Abc: An academic industrial-strength verification tool. In *Computer Aided Verification*, pages 24–40. Springer, 2010.

[11] R. Brummayer and A. Biere. Boolector: An efficient smt solver for bit-vectors and arrays. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 174–177. Springer, 2009.

[12] M. L. Case, A. Mishchenko, and R. K. Brayton. Automated extraction of inductive invariants to aid model checking. In *Formal Methods in Computer Aided Design, 2007. FMCAD'07*, pages 165–172. IEEE, 2007.

[13] P. Chauhan, D. Goyal, G. Hasteer, A. Mathur, and N. Sharma. Non-cycle-accurate sequential equivalence checking. In *Proceedings of the 46th Annual Design Automation Conference*, pages 460–465. ACM, 2009.

[14] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani. The mathsat5 smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 93–107. Springer, 2013.

[15] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-level synthesis for fpgas: From prototyping to deployment. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 30(4):473–491, 2011.

[16] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[17] B. Dutertre and L. De Moura. The yices smt solver. *Tool paper at http://yices. csl. sri. com/tool-paper. pdf*, 2:2, 2006.

[18] M. Fujita. Equivalence checking between behavioral and rtl descriptions with virtual controllers and datapaths. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 10(4):610–626, 2005.

[19] S. Jha, R. Limaye, and S. A. Seshia. Beaver: Engineering an efficient smt solver for bit-vector arithmetic. In *Computer Aided Verification*, pages 668–674. Springer, 2009.

[20] A. Koelbl, R. Jacoby, H. Jain, and C. Pixley. Solver technology for system-level to rtl equivalence checking. In *Design, Automation & Test in Europe Conference & Exhibition, 2009. DATE'09.*, pages 196–201. IEEE, 2009.

[21] D. Kroening, E. Clarke, and K. Yorav. Behavioral consistency of C and Verilog programs using bounded model checking. In *Proceedings of DAC 2003*, pages 368–371. ACM Press, 2003.

[22] A. Kuehlmann and J. Baumgartner. Transformation-based verification using generalized retiming. In *Computer Aided Verification*, pages 104–117. Springer, 2001.

[23] J. Long, S. Ray, B. Sterin, A. Mishchenko, and R. Brayton. Enhancing abc for ltl stabilization verification of systemverilog/vhdl models. *Ganesh Gopalakrishnan University of Utah USA*, page 38, 2011.

[24] A. Mishchenko, S. Chatterjee, R. Brayton, and N. Een. Improvements to combinational equivalence checking. In *Computer-Aided Design, 2006. ICCAD '06. IEEE/ACM International Conference on*, pages 836–843, 2006.

[25] V. Paruthi and A. Kuehlmann. Equivalence checking combining a structural sat-solver, bdds, and simulation. In *Computer Design, 2000. Proceedings. 2000 International Conference on*, pages 459–464, 2000.

[26] E. Pavlenko, M. Wedler, D. Stoffel, W. Kunz, A. Dreyer, F. Seelisch, and G. Greuel. Stable: A new qf-bv smt solver for hard verification problems combining boolean reasoning with computer algebra. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2011*, pages 1–6. IEEE, 2011.

[27] N. Shekhar, P. Kalla, and F. Enescu. Equivalence verification of polynomial datapaths using ideal membership testing. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 26(7):1320–1330, 2007.

[28] N. Shekhar, P. Kalla, F. Enescu, and S. Gopalakrishnan. Equivalence verification of polynomial datapaths with fixed-size bit-vectors using finite ring algebra. In *Computer-Aided Design, 2005. ICCAD-2005. IEEE/ACM International Conference on*, pages 291–296, 2005.

[29] W. Snyder, P. Wasson, and D. Galbi. Verilator: Convert verilog code to c++/systemc, 2012.

[30] B. Sterin, N. Een, A. Mishchenko, and R. Brayton. The benefit of concurrency in model checking. IWLS, 2011.

[31] D. Stoffel and W. Kunz. Equivalence checking of arithmetic circuits on the arithmetic bit level. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 23(5):586–597, 2004.

[32] D. Wang and J. Levitt. Automatic assume guarantee analysis for assertion-based formal verification. In *Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, pages 561–566. ACM, 2005.

[33] M. Wedler, D. Stoffel, R. Brinkmann, and W. Kunz. A normalization method for arithmetic data-path verification. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 26(11):1909–1922, 2007.