LEC : Learning-Driven Equivalence Checking of Data-Path Arithmetic

Jiang Long, Robert. K. Brayton, Mike Case

Oct 19th, 2013

1 Background

- Data-path arithmetic
- Related work

2 LEC: design and implementation

- Tool architecture and sub-model tree
- Learning techniques
  - Generic Techniques
  - Specialized techniques
- Integration

3 Experiments

4 Conclusion

Background

- Data-path optimization
  - RTL synthesis
  - High-level synthesis
    - Catapult C
    - Forte
    - Synfora
    - AutoESL
    - ...
- Data-path equivalence checking
  - Important component of the design flow
- The challenge
  - Differences are introduced from high-level arithmetic/algebraic optimizations
  - Need to reconstruct high-level transformation from a sea of logic

Data-path logic

- Expressed by synthesizable VHDL/Verilog operators
- Expressed by QF_BV logic in SMT 2.0
- No memory/array constructs

	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*

Related work

Bit-level equivalence checking procedures

- Structural hashing
- SAT-sweep
- BDD/SAT solvers

High-level approach

- MBDD
- STABLE
- Interval arithmetic
- Integer Linear Programming
- Vanishing polynomial, Grobner base

C/RTL equivalence checking

- C/RTL equivalence checking: Hector, SLEC, CBMC

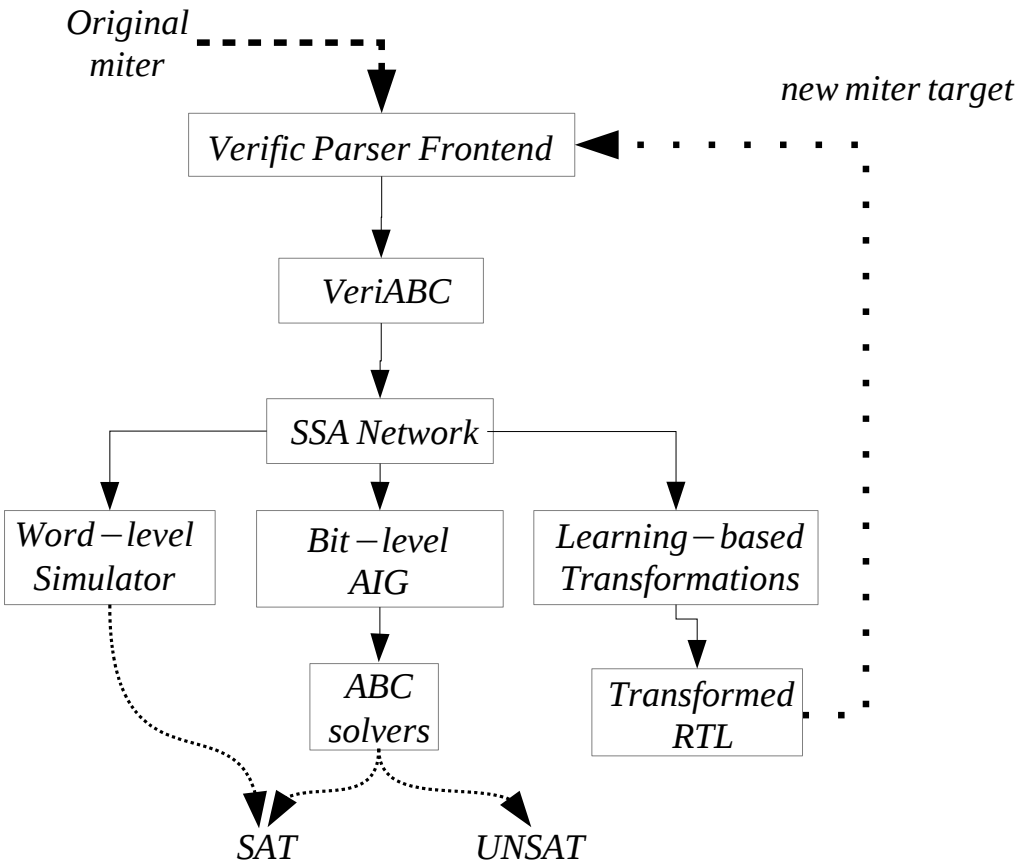
Comparison with QF_BV SMT solvers

- General DPLL
  - Lazy/eager
  - Tight integration with SAT-solver
  - Use conflict clauses to communicate between theory solver and SAT solver
- Word-level transformation
  - Rewriting
- Benchmarking solvers
  - Boolector
  - z3

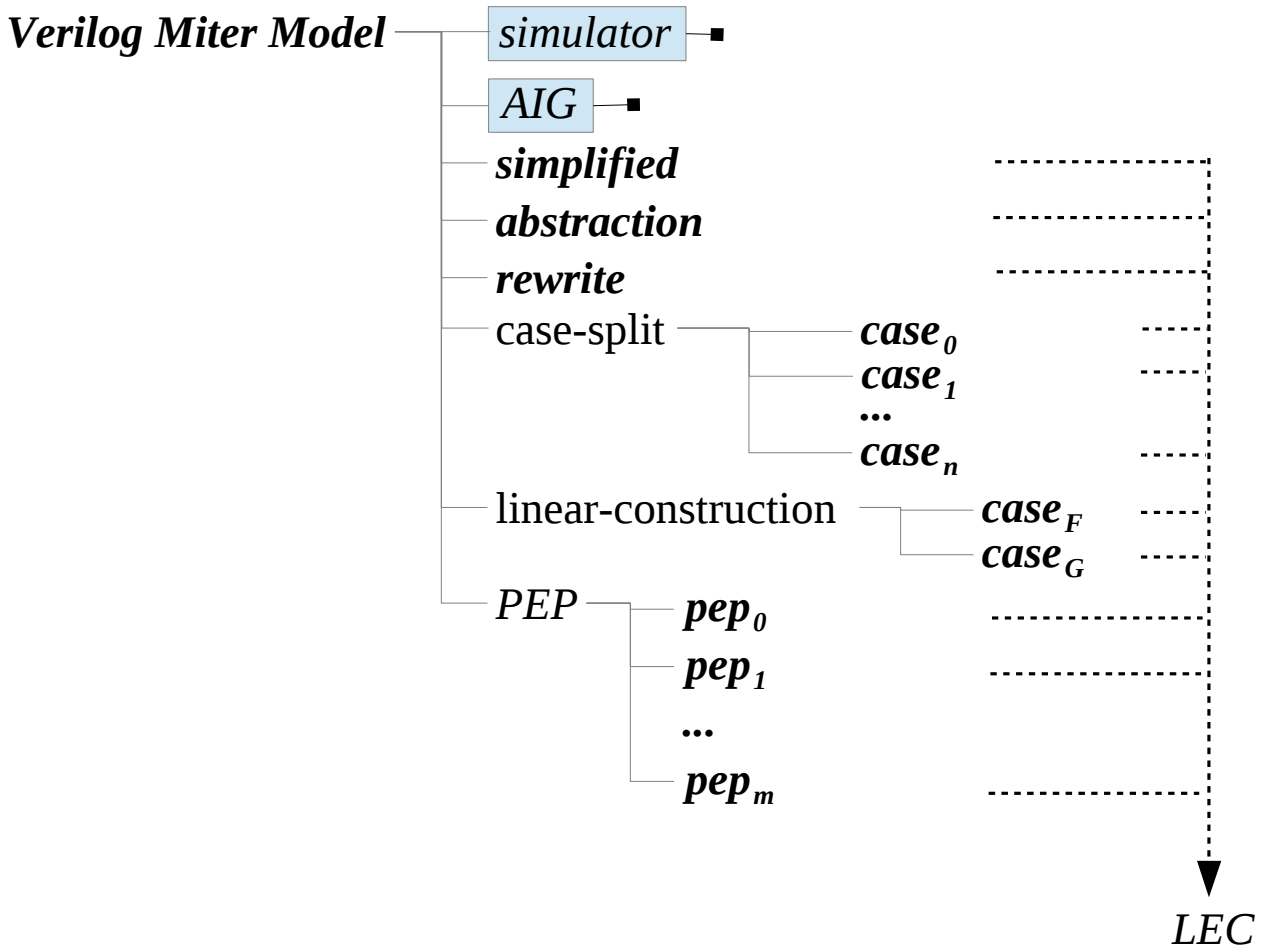
Our strategy

The two designs under comparison needs to be transformed to have more internal match points, i.e. structurally more similar.

LEC architecture



LEC sub-model tree



Example sub-model tree proof


```
{
  "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]
```

Generic Techniques

- Bit-level SAT solving
- Word-level random simulation
- Simplification
- Abstraction
- Case-split
- Rewriting

Bit-level SAT solving

- Create AIG model from SSA network
- Use ABC as external libraries
  - dcec
  - iprove
- Effort level: low , high

Word-level Random simulation

Implementation

- Compiled simulation using Verilator
- Native simulator by interpretation

Usages

- Identify constant signals
- Identify potential equivalent pairs
- Compute abstraction

Simplification

Constants:

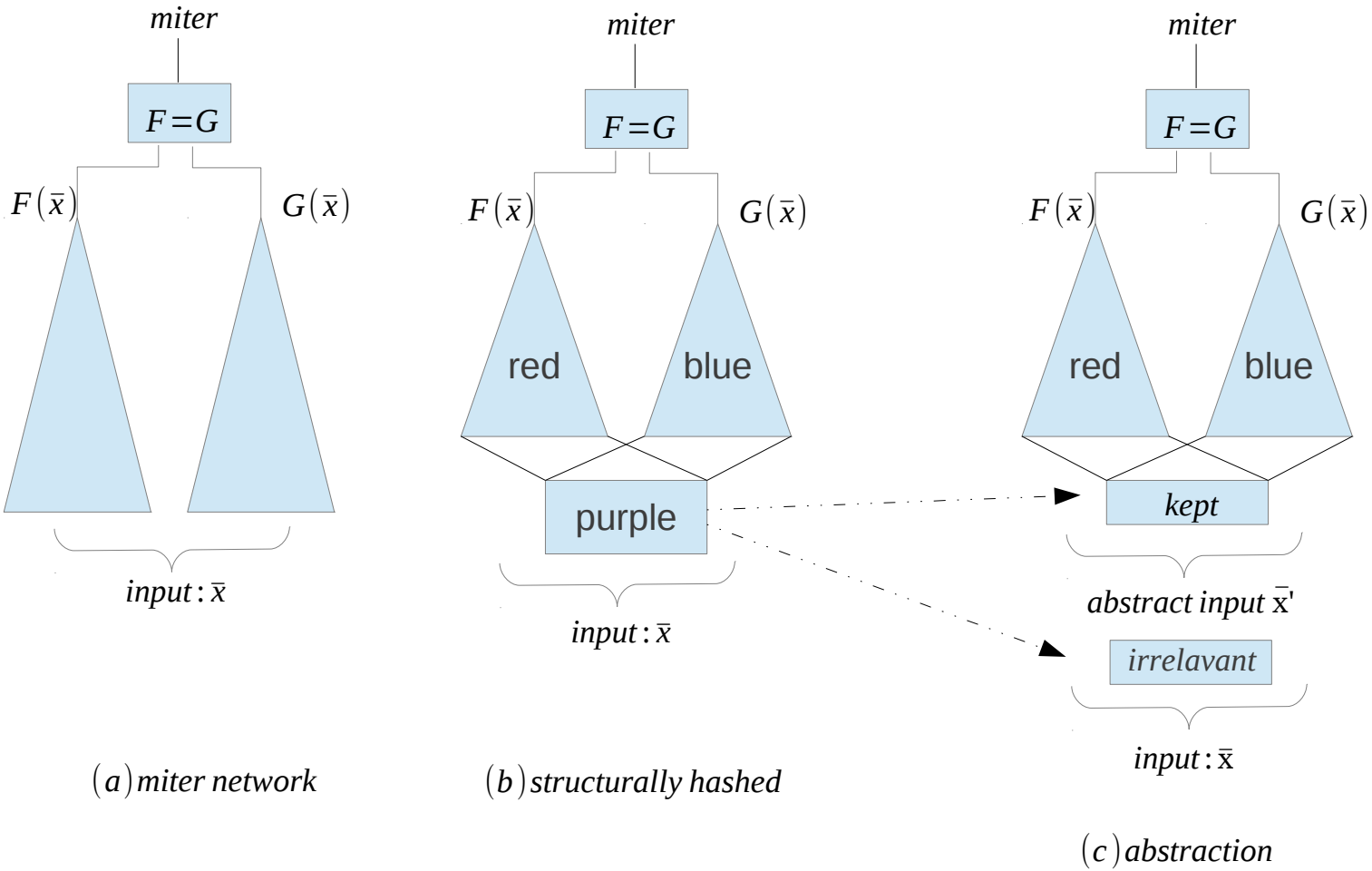
- Propagate proven constants

Potential equivalent pairs (PEP):

- Merge proven PEPs
- Create sub-model target for inconclusive PEPs

Abstraction

- Conjecture to remove common COI.



Rewriting

	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	$\{a, b[n - 1 : 0]\} * c$	$a * c \ll n + b[n - 1 : 0] * c$

Rewriting

- Distribute $+$ over $[]$

$$(a+b)[m:n] = a[m:n] + b[m:n] + (a[n-1:0] + b[n-1:0])[n] \quad (1)$$

Rewriting: Condition rewriting

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

if we can prove

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

Case split

$$F(\bar{x}) = G(\bar{x}) \leftrightarrow \begin{cases} F_{\bar{x_0}\bar{x_1}}(\bar{x}) = G_{\bar{x_0}\bar{x_1}}(\bar{x}) \\ F_{\bar{x_0}x_1}(\bar{x}) = G_{\bar{x_0}x_1}(\bar{x}) \\ F_{x_0\bar{x_1}}(\bar{x}) = G_{x_0\bar{x_1}}(\bar{x}) \\ F_{x_0x_1}(\bar{x}) = G_{x_0x_1}(\bar{x}) \end{cases}$$

- Introduce constants
- Enables other transformations after simplification

Typical use model

Collect design statistics

- Number of nodes, bit-width
- Number of each arithmetic operators, shifts, MUXs, etc.
- Size of common logic
- Visualize if possible

Run simulation and ABC's *dcec* to prove or dis-prove.

Run the following procedures :

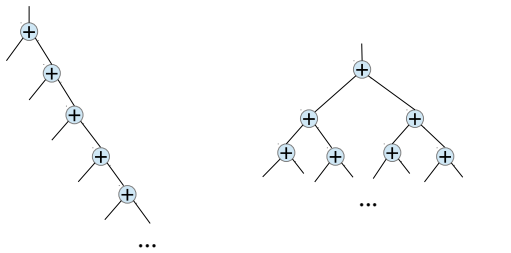
1. Simplify: each SAT call is given 2-second limit
2. Abstraction: no need for a refinement
3. Case-split: look for one-bit input, MSB/LSB, control flow
4. Rewriting (recursively applied)
5. Continue solve the smallest PEP candidates

What if we reach a dead-end, none of the techniques are making any further progress...

Specialized technique: Polynomial reconstruction

- Motivation
  - Rewriting is a local transformation
- Example

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



(a) linear adder chain (b) balanced adder tree

Polynomial construction : $F(\bar{x}) \equiv G(\bar{x})$

$$p = q$$

$$F' \text{ implements } p$$

$$G' \text{ implements } q$$

$$F' \equiv F$$

$$G' \equiv G$$

$$- - - - - - - - - - - - - - - - -$$

$$F \equiv G$$

Case study : linear sum

Prove $F(\bar{x}) = G(\bar{x})$:

Domain	signed integer arithmetic
conjecture p	$p = \sum_i a_i \cdot x_i + b$
conjecture q	$q = \sum_i a'_i \cdot x_i + b'$
$p = q$	$a_i = a'_i$ and $b = b'$
construct F'	Follow structural hints of F
construct G'	Follow structural hints of G
F' implements p	Overflow checks
G' implements q	Overflow checks
$F \equiv F'$	LEC
$G \equiv G'$	LEC

Linear sum: probe coefficients

In signed integer domain, linear sum construction for $F(x_0, x_1, ..., x_n)$:

$$p = \sum_{i}^{n} a_i \cdot x_i + b \tag{4}$$

Probe F with input vectors:

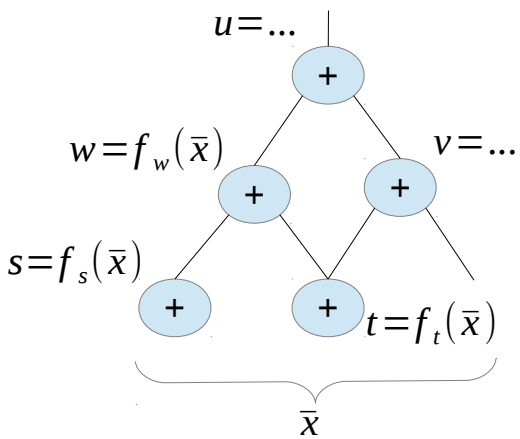
$$\begin{aligned} b &= F(0, 0, ..., 0) \\ a_0 &= F(1, 0, ..., 0) - b \\ a_1 &= F(0, 1, ..., 0) - b \\ &... \\ a_n &= F(0, 0, ..., 1) - b \end{aligned}$$

Next: Prove the F implements the linear sum p

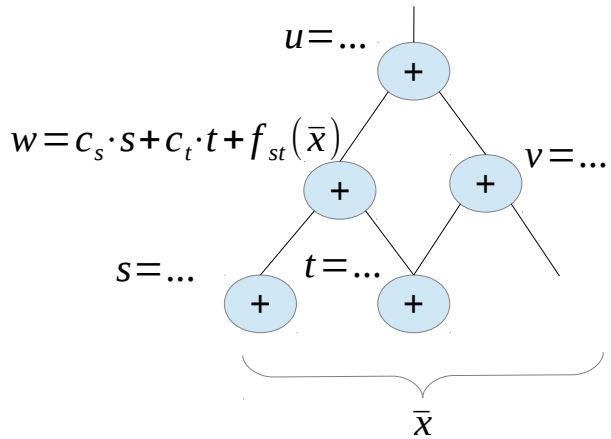
Linear sum: Prove F implements p

- Construct F' implements p
- Prove $F \equiv F'$

$$p = \sum_{i}^{n} a_i \cdot x_i + b \tag{5}$$



(a) *annotated reduced graph*



(b) *substituted annotation*

Linear sum : Prove F' implements f

The following checks ensures F' implements f:

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

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

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

The above checks ensure F' implements the linear sum p by construction.

Linear sum : Summary

$$p = q = \sum_i a_i \cdot x_i + b$$

$$F' \text{ implements } p$$

$$G' \text{ implements } q$$

$$F' \equiv F$$

$$G' \equiv G$$

$$- - - - - - - - - - - - - - - - - -$$

$$F \equiv G$$

1 Background

- Data-path arithmetic
- Related work

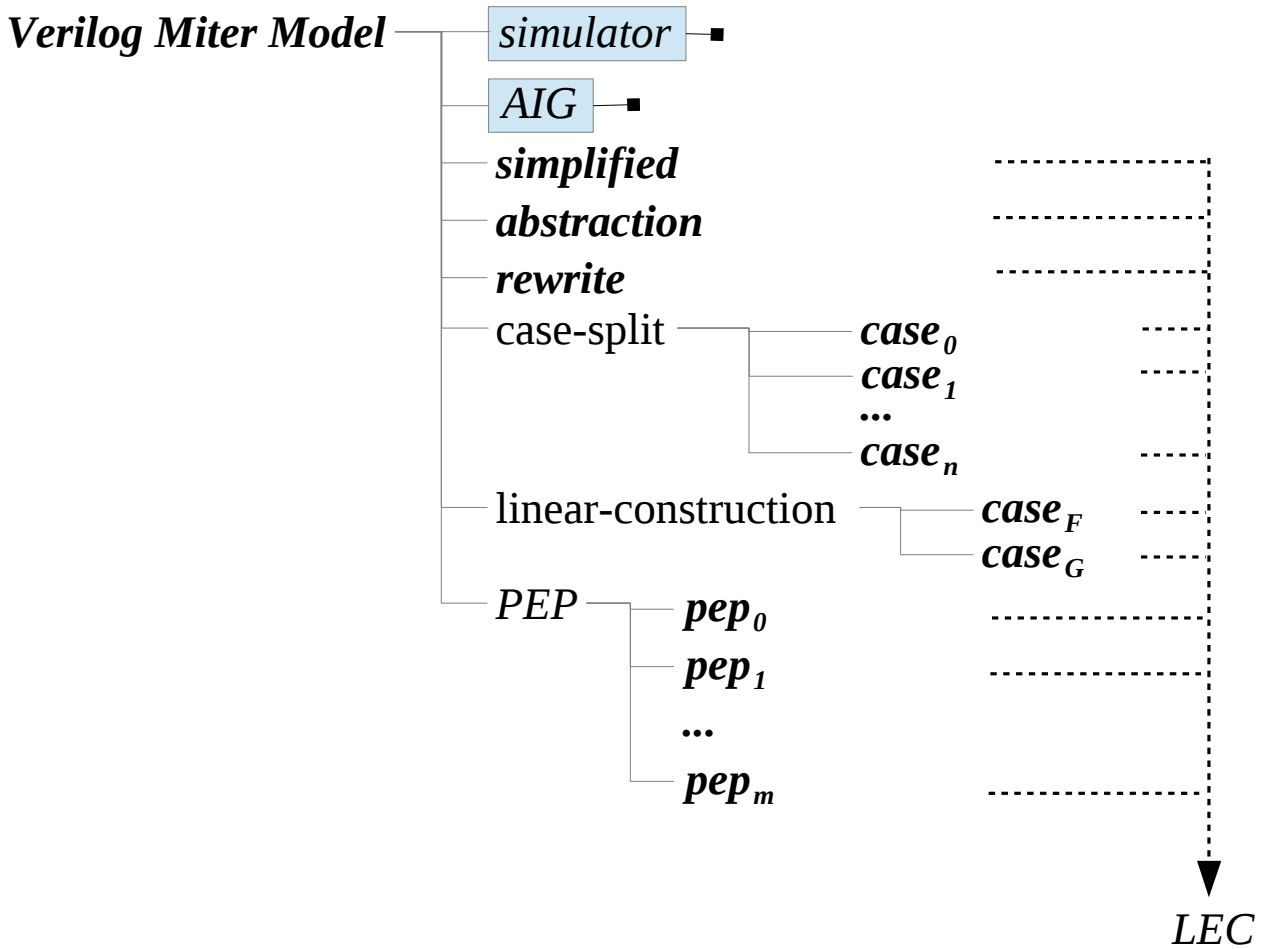
2 LEC: design and implementation

- Tool architecture and sub-model tree
- Learning techniques
  - Generic Techniques
  - Specialized techniques
- Integration

3 Experiments

4 Conclusion

LEC integration



Integration: Sub-models

Model	Result	Type
simulation	SAT_ONLY	Terminal
AIG	IFF	Terminal
simplified model	IFF	Recursive
abstraction model	UNSAT_ONLY	Recursive
case-split model	IFF	Recursive
rewrite model	IFF	Recursive
PEP sub-model	SIMPLIFY	Recursive
linear construction	IFF	Recursive

Integration: Sub-model disjunction

‖	**SAT**	**UNS**	**UNK**	**SIM**	**CON**	**BOT**
SAT	SAT	CON	SAT	SAT	CON	SAT
UNS	CON	UNS	UNS	UNS	CON	UNS
UNK	SAT	UNS	UNK	SIM	CON	UNK
SIM	SAT	UNS	SIM	SIM	CON	SIM
CON	CON	CON	CON	CON	CON	CON
BOT	SAT	UNS	UNK	SIM	CON	BOT

Integration: Sub-model conjunction

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

Integration: Example


```
{
    "bit-blast model": null,
    "random sim model": null,
    "rewrite_model": null,
    "pep miters": {
        "proven peps": [],
        "miters for unknown peps": {}
    },
    "merged_model": null,
    "reduced model": null,
    "case split procedure": {
        "case_0": "PROVEN by bitblast to aig",
        "case_1": {
            "bit-blast model": null,
            "random sim model": "PASSED",
            "rewrite_model": null,
            "pep miters": {
                "proven peps": [

                    "138 344 577 ",
                    "110 317 550 ",
                    "127 333 566 "
                ],
                "miters for unknown peps": {
                    "708_804": null,
                    "736_751": null
                }
            },
            "merged_model": {
                "bit-blast model": null,
                "random sim model": "PASSED",
                "rewrite_model": null,
                "pep miters": {
                    "proven peps": [],
                    "miters for unknown peps": {}
                },
                "merged_model": null,
                "reduced model": {
                    "bit-blast model": null,
                    "random sim model": null,
                    "rewrite_model": null,
                    "pep miters": {
                        "proven peps": [],
                        "miters for unknown peps": {}
                    },
                    "merged_model": null,
                    "reduced model": null,
                    "case split procedure": {
                        "case_00": "PROVEN by bitblast to aig",
                        "case_01": "PROVEN by bitblast to aig",
                        "case_10": "PROVEN by bitblast to aig",
                        "case_11": "PROVEN by bitblast to aig"
                    },
                    "Linear construction": null
                },
                "case split procedure": null,
                "Linear construction": null
            },
            "reduced model": null,
            "case split procedure": null,
            "Linear construction": null
        }
    },
    "Linear construction": null
}
```

Case-study : Linear sum


```
                                    red  blue purple
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
```

Experimental results

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

d1: $\{a, b[n-1:0]\} * c = a * c \ll n + b[n-1:0] * c$

d2: $(a \ll c) * b = (a * b) \ll c$

d3: embedded mul_64_64

d4: apply case-splits twice

Conclusion

- Open architecture to integrate new domain-specific techniques
- Linear-construction procedure to bring similiarities from high-level
- Empower user to conduct learning based transformation to isolate and identify the real bottleneck of the problem from a sea of logic

Future work

- Speed up the generic learning procedures
- Incorporate other learning techniques to categorize the data-path characteristics
- Create generic scripts for mature strategies
- Create domain-specific techniques
  - Constant multiplication: $181 \cdot x_1 - 181 \cdot x_2 + 284 \cdot x_3$
  - High order polynomial
  - Vanishing polynomial/Grobner base for modulo integer arithmetic
  - ...

LEC : Computer-aided Reverse Engineering of Human Intelligence

Oct 19th, 2013