Challenges in Bit-Precise Reasoning

Armin Biere

Johannes Kepler University

Linz, Austria

based on joined work with

Aina Niemetz, Andreas Fröhlich, Gergely Kovásznai, Mathias Preiner

FMCAD 2014

EPFL, Lausanne, Switzerland

Tuesday, 21 October, 2014


VSL 2014

SMT-COMP - Mozilla Firefox

SMT-COMP

www.smtcomp.org

QF_BV

Competition results for the QF_BV division as of Fri Jun 27 16:49:23 EDT 2014

Competition benchmarks = 2488 (total = 32500, unknown status = 28138, trivial = 546)

Division COMPLETE: The winner is Boolector - BRONZE medal winner

Solver	Errors	Solved	Not Solved	Remaining	CPU Time (on solved instances)	Weighted medal score weight = 3.396
Boolector	0	2361	127	0	138077.59	3.058
STP-CryptoMiniSat4	0	2283	205	0	190660.82	2.859
[CVC4-with-bugfix]	0	2237	251	0	139205.24	2.745
[MathSAT]	0	2199	289	0	262349.39	2.653
[Z3]	0	2180	308	0	214087.66	2.607
CVC4	0	2166	322	0	87954.62	2.574
4Simp	0	2121	367	0	187966.86	2.468
SONOLAR	0	2026	462	0	174134.49	2.252
Yices2	0	1770	718	0	159991.55	1.719
abziz_min_features	9	2155	324	0	134385.22	2.548
abziz_all_features	9	2093	386	0	122540.04	2.403

Home • Intro • Tools • Specs • Thanks • SMT-LIB • Previous

SMT-COMP - Mozilla Firefox

SMT-COMP

www.smtcomp.org

QF_ABV

Competition results for the QF_ABV division as of Fri Jun 27 16:49:23 EDT 2014

Competition benchmarks = 6457 (total = 15091, unknown status = 4190, trivial = 4423)

Division COMPLETE: The winner is Boolector (justification)

Solver	Errors	Solved	Not Solved	Remaining	CPU Time (on solved instances)	Weighted medal score weight = 3.8
Boolector (justification)	0	6413	44	0	53176.27	3
Boolector (dual propagation)	0	6410	47	0	69040.03	3
[MathSAT]	0	6394	63	0	73535.00	3
SONOLAR	0	6386	71	0	53248.38	3
CVC4	0	6352	105	0	78865.09	3
[Z3]	0	6351	106	0	53957.15	3
Yices2	1	6410	46	0	37112.15	3
Kleaver-STP	56	5827	574	0	1120.08	3
Kleaver-portfolio	91	5799	567	0	3403.29	3

Home • Intro • Tools • Specs • Thanks • SMT-LIB • Pr

Last modified: Wed 16 Jul 2014 19:3

W3C XHTML 1.0 W3C CS

Almost All Binary Search Implementations are Broken

```
int bsearch (int * a, int n, int e) {
  int l = 0, r = n;
  if (!n) return 0;
  while (l + 1 < r) {
    printf ("l=%d r=%d\n", l, r);
    int m = (l + r) / 2;
    if (e < a[m]) r = m;
    else l = m;
  }
  return a[l] == e;
}
```

```
int main (void) {
  int n = INT_MAX;
  int * a = calloc (n, 4);
  (void) bsearch (a, n, 1);
}
```

```
$ ./bsearch
l=0 r=2147483647
l=1073741823 r=2147483647
Segmentation fault
```

- common "word-level" operators QF_BV standard SMTLIB2 format
  - constants: `0x7fffffff`, variables: fixed size bit vectors `bool x[32]`
  - predicates: *equality* "$x = y$", *inequality* "$x \leq y$" (signed & unsigned)
  - bit-wise logical ops: *negation*, *conjunction*, *xor* `~x` `x & y` `x ^ y`
  - word operators: *slicing* "$x[l : r]$", *concatenation* "$x \circ y$"
  - *conditional* operator or *if-then-else* operator "$c\ ?\ t : e$"
  - *zero extension* and *sign extension*
  - shift operators: *left shift*, *arithmetic/logical right shift*, *rotation*
  - basic arithmetic operators: *negation* (1-complement), *addition*, *multiplication*
  - overflow checking for addition and multiplication
  - derived arith. ops: *unary minus* (2-complement), *substraction*, *division*, *modulo*
- extended word-level operators (QF_)[A][UF]BV
  - uninterpreted functions "UF", arrays "A" with *read* / *write* operators
  - with quantifiers (no "QF_")

Modelling with Bit-Vectors

- allows to capture bit-precise semantics precisely
  - RTL-level / word-level for HW
  - assembler or C level for SW
    but beware: `int` in Java has 2-complement semantics
- *arrays* used to model memories in HW or pointers in SW
  - low-level (flat) memory model
  - "writable" extension of uninterpreted functions (UF $\subseteq$ A)
  - extensional arrays:
    - check satisfiability assuming equality of (updated) arrays
    - a = write (b, j, v) $\wedge$ read (a, j) $\neq$ v
      in this example extensionality could be removed by substitution
- quantifiers (and lambdas) are even more powerful than arrays
- typical scenario
  - symbolic execution of a program
  - bounded model checking of an RTL model

Bit-Blasting of 4-Bit Addition

addition of 4-bit numbers x, y with result s also 4-bit: $s = x + y$

$$[s_3, s_2, s_1, s_0]_4 = [x_3, x_2, x_1, x_0]_4 + [y_3, y_2, y_1, y_0]_4$$

$$\begin{aligned}
[s_3, \cdot\,]_2 &= \mathsf{FullAdder}(x_3, y_3, c_2) \\
[s_2, c_2]_2 &= \mathsf{FullAdder}(x_2, y_2, c_1) \\
[s_1, c_1]_2 &= \mathsf{FullAdder}(x_1, y_1, c_0) \\
[s_0, c_0]_2 &= \mathsf{FullAdder}(x_0, y_0, 0)
\end{aligned}$$

where

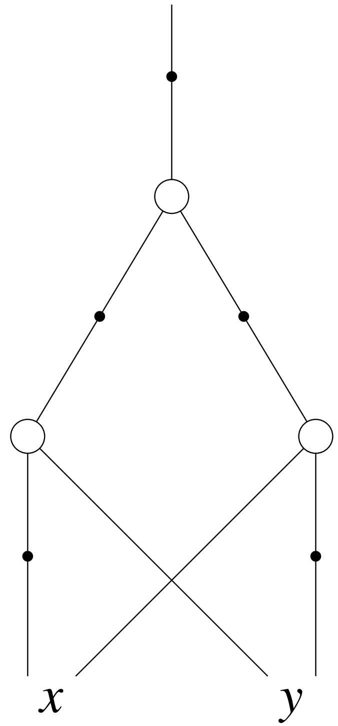
$$\begin{aligned}
[\, s \,,\, o \,]_2 &= \mathsf{FullAdder}(x, y, i) \quad \text{with} \\
s &= x \mathbin{\hat{}} y \mathbin{\hat{}} i \\
o &= (x \wedge y) \vee (x \wedge i) \vee (y \wedge i) = ((x + y + i) \geq 2)
\end{aligned}$$

And-Inverter-Graphs (AIG)

- widely adopted bit-level intermediate representation
  - see for instance our AIGER format http://fmv.jku.at/aiger
  - used in Hardware Model Checking Competition (HWMCC)
  - also used in the *structural track* in (ancient) SAT competitions
  - many companies use similar techniques
- basic logical operators: *conjunction* and *negation*
- DAGs: nodes are conjunctions, negation/sign as *edge attribute*

  bit stuffing: signs are compactly stored as LSB in pointer
- automatic sharing of isomorphic graphs, constant time (peep hole) simplifications
- *or even* SAT sweeping, full reduction, etc ... see ABC system from Berkeley

XOR as AIG



negation/sign are edge attributes

not part of node

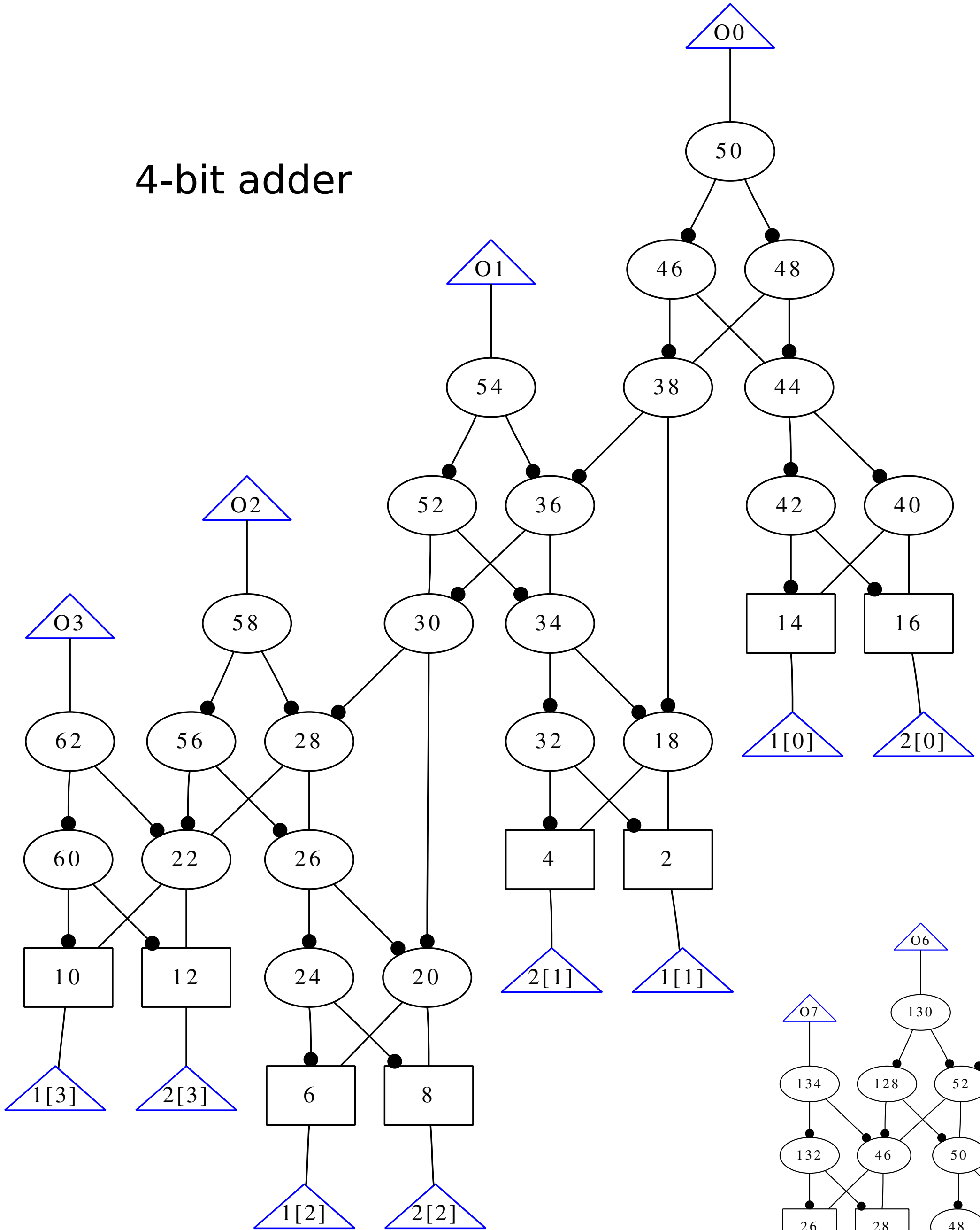
$$x \mathbin{\hat{}} y \equiv (\bar{x} \wedge y) \vee (x \wedge \bar{y}) \equiv \overline{\overline{(\bar{x} \wedge y)} \wedge \overline{(x \wedge \bar{y})}}$$

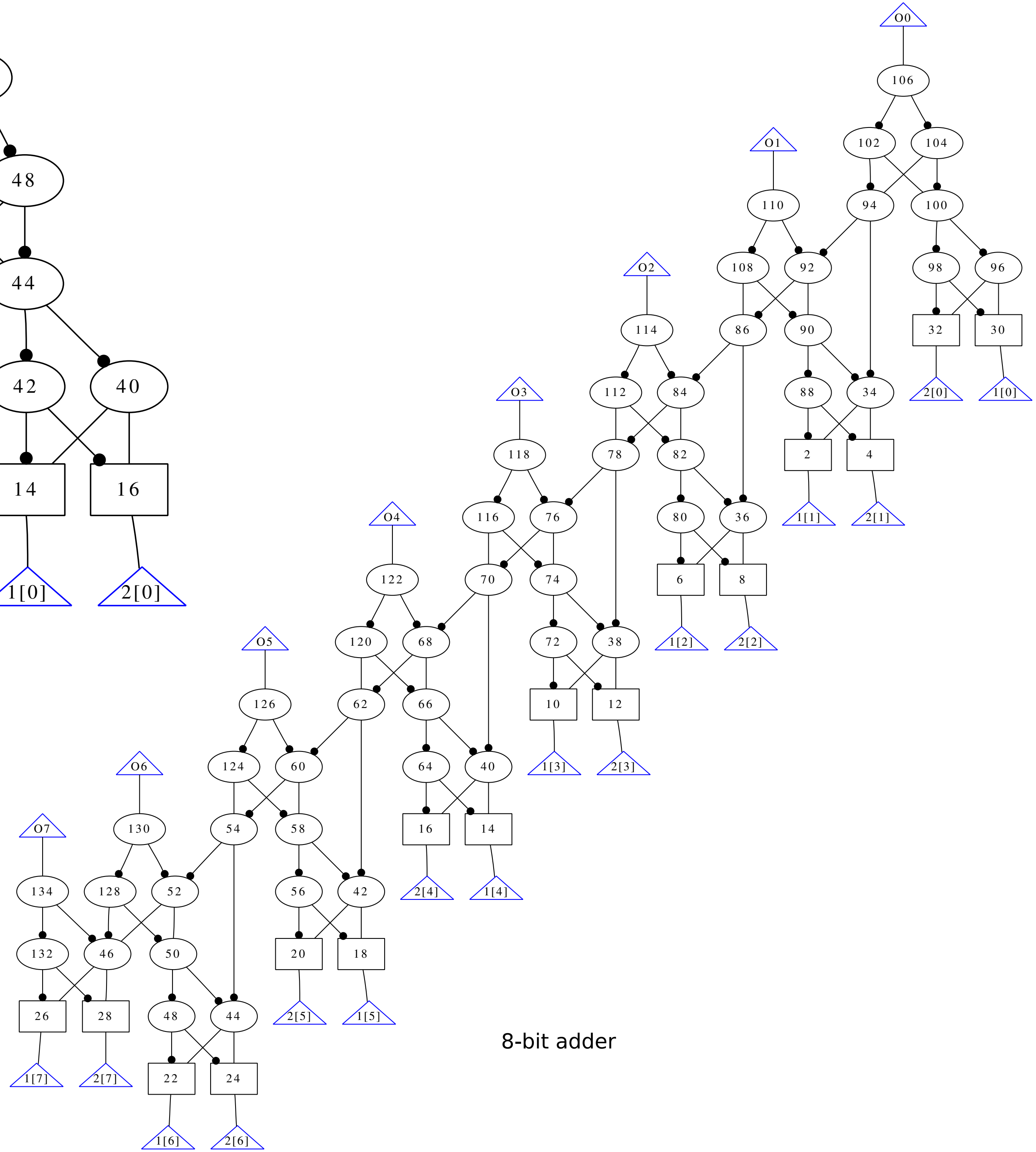
```c
typedef struct AIG AIG;

struct AIG
{
  enum Tag tag;                   /* AND, VAR */
  void *data[2];
  int mark, level;                /* traversal */
  AIG *next;                      /* hash collision chain */
};

#define sign_aig(aig) (1 & (unsigned) aig)
#define not_aig(aig) ((AIG*)(1 ^ (unsigned) aig))
#define strip_aig(aig) ((AIG*)(~1 & (unsigned) aig))
#define false_aig ((AIG*) 0)
#define true_aig ((AIG*) 1)
```

assumption for correctness:

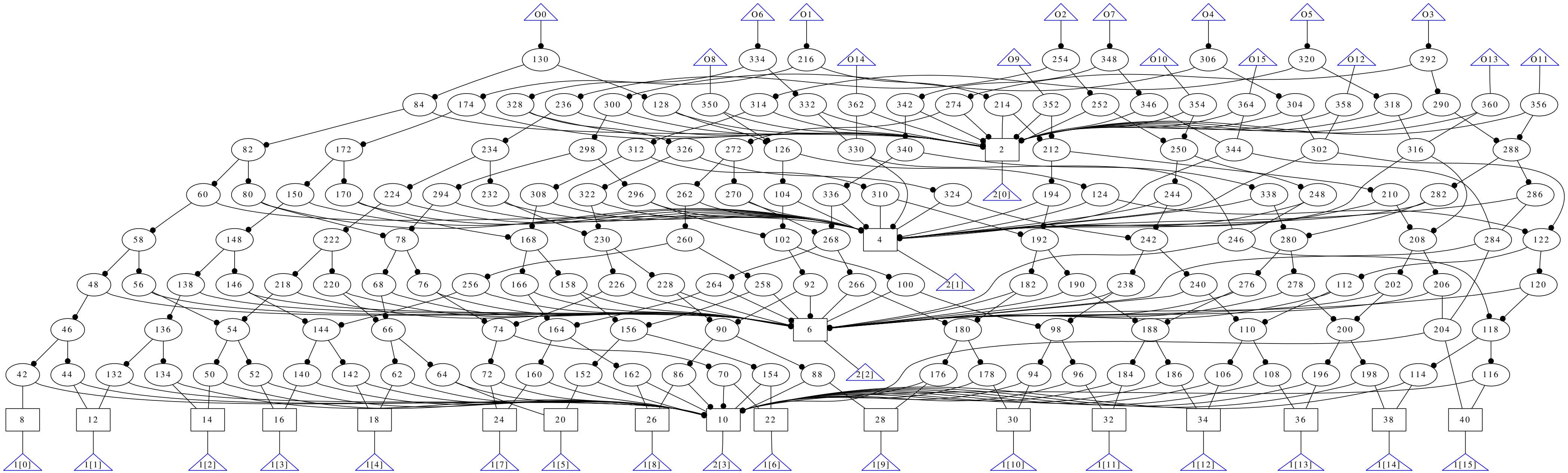
`sizeof(unsigned) == sizeof(void*)`

4-bit adder

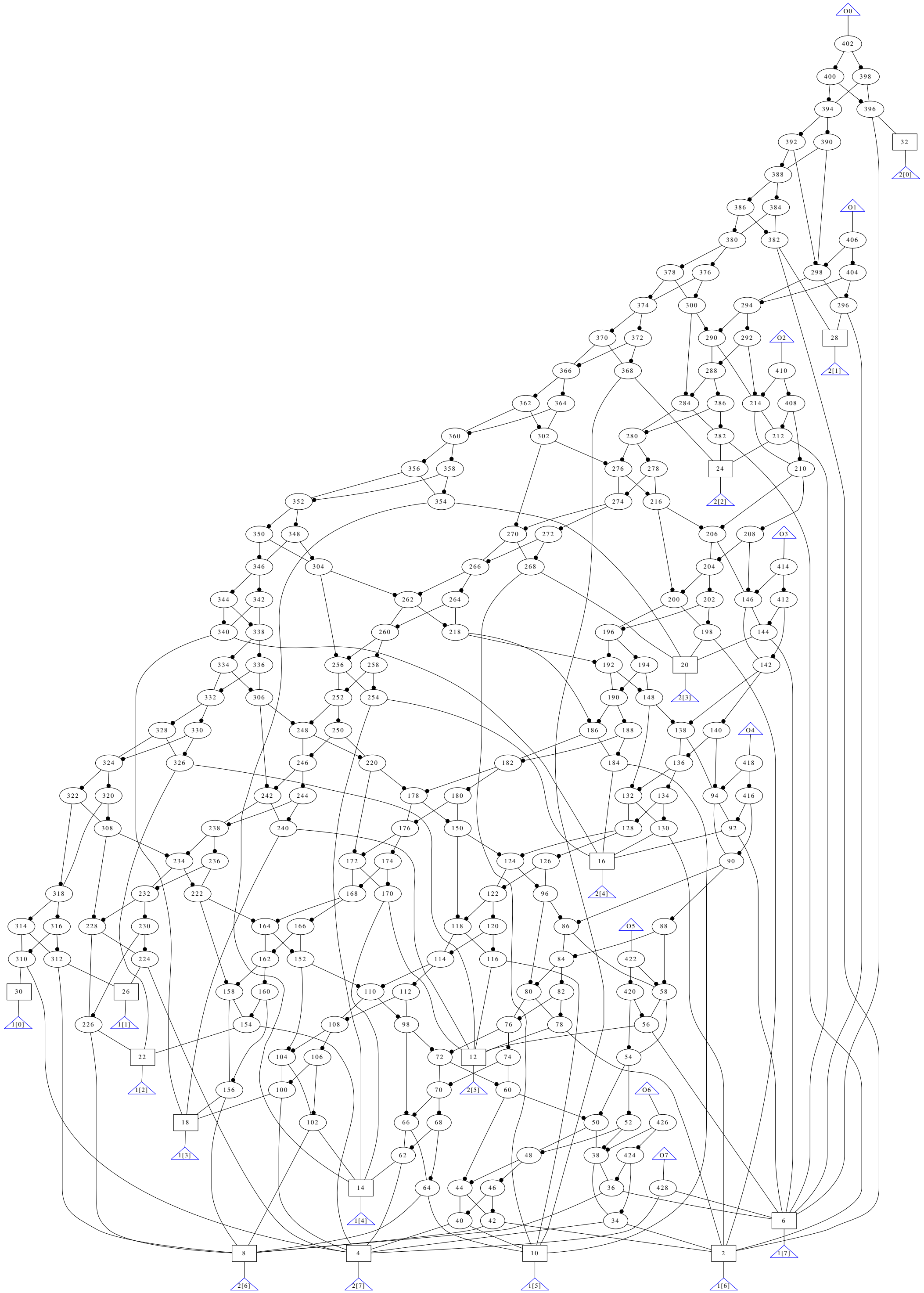




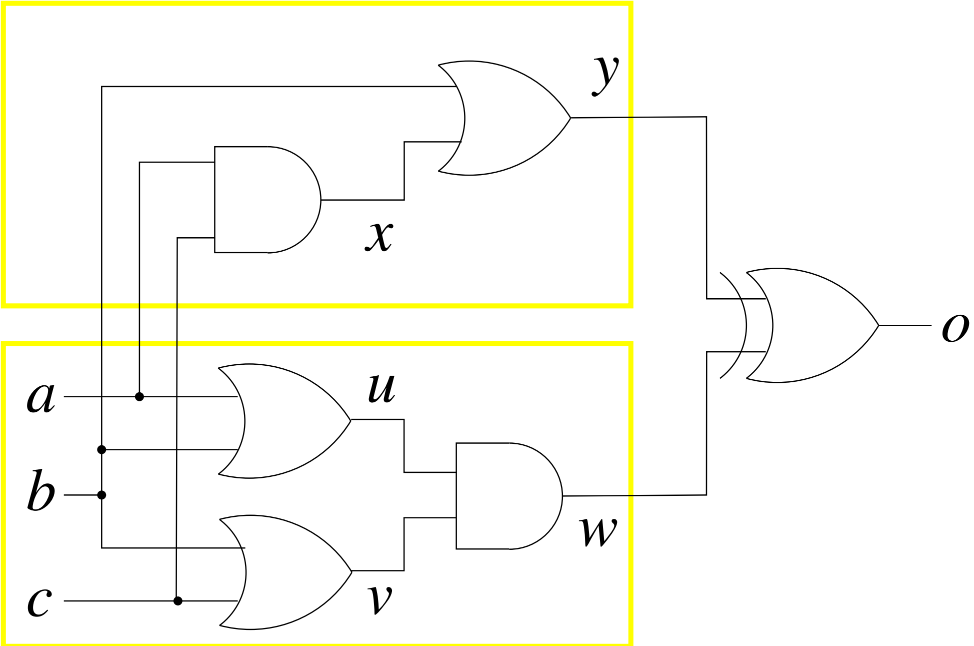
8-bit adder



bit-vector of length 16 shifted by bit-vector of length 4


O0
O1
O2
O3
O4
O5
O6
O7
2[0]
2[1]
2[2]
2[3]
2[4]
2[5]
2[6]
2[7]
1[0]
1[1]
1[2]
1[3]
1[4]
1[5]
1[6]
1[7]

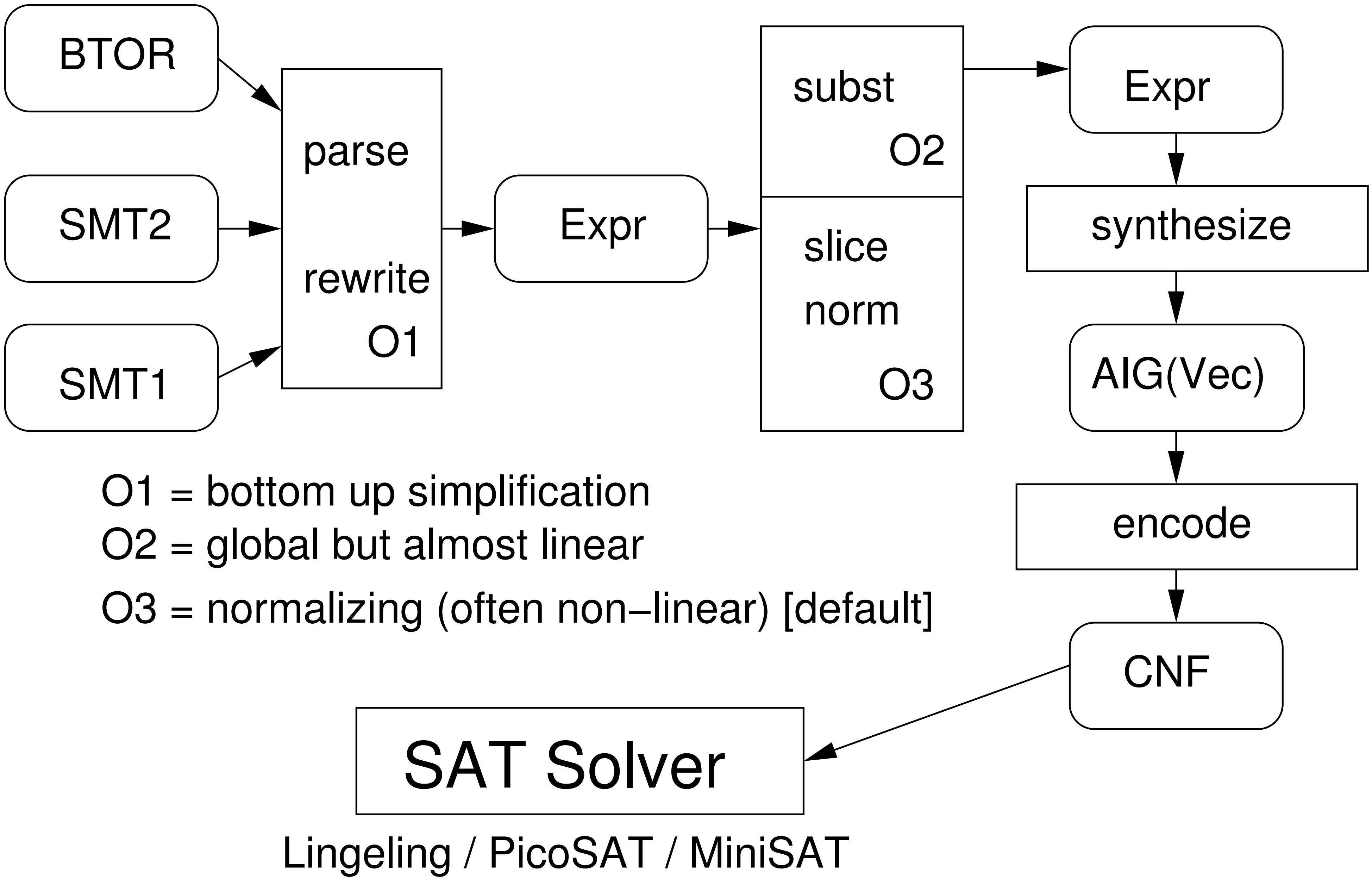
CNF



$$\begin{aligned} & o \wedge \\ (x & \leftrightarrow a \wedge c) \wedge \\ (y & \leftrightarrow b \vee x) \wedge \\ (u & \leftrightarrow a \vee b) \wedge \\ (v & \leftrightarrow b \vee c) \wedge \\ (w & \leftrightarrow u \wedge v) \wedge \\ (o & \leftrightarrow y \oplus w) \end{aligned}$$

$$o \wedge (x \rightarrow a) \wedge (x \rightarrow c) \wedge (x \leftarrow a \wedge c) \wedge \ \ldots$$

$$o \wedge (\overline{x} \vee a) \wedge (\overline{x} \vee c) \wedge (x \vee \overline{a} \vee \overline{c}) \wedge \ \ldots$$


BTOR
SMT2
SMT1
parse
rewrite
O1
Expr
subst
O2
slice
norm
O3
Expr
synthesize
AIG(Vec)
encode
CNF
O1 = bottom up simplification
O2 = global but almost linear
O3 = normalizing (often non−linear) [default]
SAT Solver
Lingeling / PicoSAT / MiniSAT

```
enum BtorNodeKind
{
  BTOR_BV_CONST_NODE  =  1,
  BTOR_BV_VAR_NODE    =  2,
  BTOR_PARAM_NODE     =  3,
  BTOR_SLICE_NODE     =  4,
  BTOR_AND_NODE       =  5,
  BTOR_BEQ_NODE       =  6,
  BTOR_FEQ_NODE       =  7,
  BTOR_ADD_NODE       =  8,
  BTOR_MUL_NODE       =  9,
  BTOR_ULT_NODE       = 10,
  BTOR_SLL_NODE       = 11,
  BTOR_SRL_NODE       = 12,
  BTOR_UDIV_NODE      = 13,
  BTOR_UREM_NODE      = 14,
  BTOR_CONCAT_NODE    = 15,
  BTOR_APPLY_NODE     = 16,
  BTOR_LAMBDA_NODE    = 17,
  BTOR_BCOND_NODE     = 18,
  BTOR_ARGS_NODE      = 19,
  BTOR_UF_NODE        = 20,
  BTOR_PROXY_NODE     = 21
};
```

Further Boolector Rewriting Internals

- fast parallel substitution
  - collects top-level variable assignments (equalities)
  - collects boolean (bit-width 1) top-level constraints (embedded constraints)
  - normalize arithmetic equalities and try to isolate variables (Gauss)
  - one pass substitution restricted to output-cone of substituted variables
  - needs occurrence check, equalities between non-variable terms not used
  - so only partially simulates congruence closure
  - but works nice for typical SSA form encodings
- boolean skeleton preprocessing
  - encode boolean (bit-width 1) part into SAT solver
  - use SAT preprocessing to extract forced units (backbone)
- replace sliced variables by new variables
- eliminate unconstrained sub-expressions
- optionally perform full beta reduction
- these expensive global rewriting steps iterated until completion

Inprocessing

- preprocessing interleaved with search or between incremental calls
  - Boolector inprocessing only in each incremental SAT call
  - Lingeling explicitly interleaves preprocessing with CDCL search
- incremental word-level solving
  - through Boolector API only (currently)
  - requires user to specify incremental usage initially
  - disables unconstrained optimization and slice elimination
- preprocessing/inprocessing in SAT solver
  - quite powerful
  - need to maintain mapping of AIG nodes to CNF variables
  - CNF variables eliminated by SAT solver can not be reused

- don't do it
- our solution: **clone** SAT solver
  - triggered after (fixed) conflict limit is reached
  - cloned SAT solver can make full use of preprocessing
  - except that it can not propagate back learned clauses to parent
- various papers by Nadel, Ryvchin, Strichman SAT'12, SAT'14:
  - bring back clauses with eliminated but reused variables
  - only works for bounded variable elimination (DP, BVE, SateLite)
  - needs support from SAT solver (best version requires to maintain proofs)
- actually cloning useful for many other things: Treengeling

Bit-Blasting Explodes

- show commutativity of bit-vector addition for bit-width 1 million:

```
(set-logic QF_BV)
(declare-fun x () (_ BitVec 1000000))
(declare-fun y () (_ BitVec 1000000))
(assert (distinct (bvadd x y) (bvadd y x)))
```

- size of SMT2 file: 138 bytes
- bit-blasting with our SMT solver Boolector
  - rewriting turned off
  - except structural hashing
  - produces AIGER circuits of file size 103 MB
- Tseitin transformation leads to CNF in DIMACS format of size 1 GB

Complexity of Bit-Vector Logics

- SMT2 bit-vector logic QF_BV
  - quantifier free bit-vector logic
  - all common operators (incl. multiplication, division etc.)
  - without uninterpreted functions nor arrays nor with macros (`define-fun`)
- classical *bogus* argument
  - bit-blast formula (polynomially in bit-width)
  - check with SAT solver, thus in NP
  - any CNF is a bit-vector formula, thus NP hard
- *however* bit-blasting is really exponential
  - since bit-width is encoded logarithmically:

    `(declare-fun x () (_ BitVec 1000000))`
  - same for constants: `0x7fffffff`
- we claim this is a fundamental difference: word-level vs. bit-level

from our SMT'12 paper (extended journal version submitted):

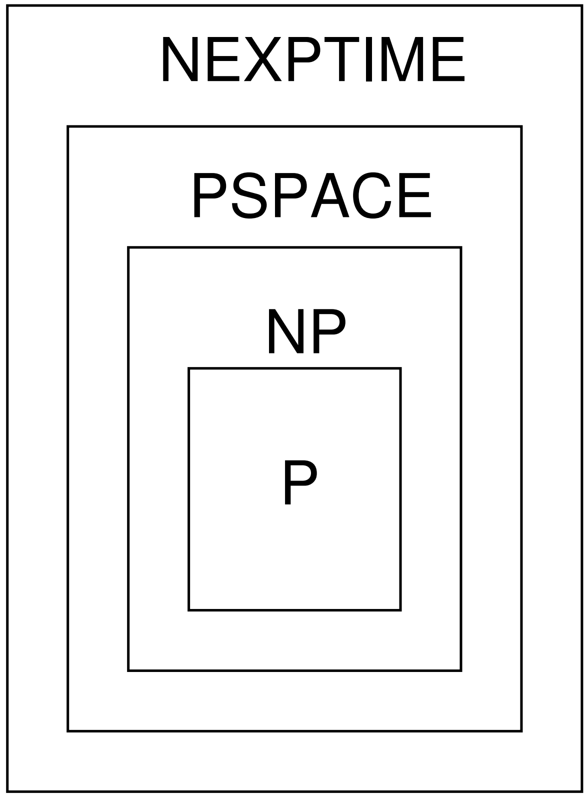
		quantifiers			
		no		*yes*	
		uninterpreted functions		uninterpreted functions	
		no	*yes*	*no*	*yes*
encoding	*unary*	NP QF_BV1 obvious	NP QF_UFBV1 Ackermann	PSPACE BV1 [TACAS'10]	NEXPTIME UFB1 [FMCAD'10]
	binary	**NEXPTIME** **QF_BV2** **[SMT'12]**	NEXPTIME QF_UFBV2 [SMT'12]	?	2NEXPTIME UFBV2 [SMT'12]

QF = "quantifier free" UF = "uninterpreted functions" BV = "bit-vector logic"

BV1 = "unary encoded bit-vectors" BV2 = "binary encoded bit-vectors"

Some Complexity Classes

- P
  - problems with polynonmially **time**-bounded algorithms
  - bounds measured in terms of input (file) size
- NP
  - same as P but with non-determininistic choice
  - needs a SAT solver
- PSPACE
  - as P but **space**-bounded
  - QBF falls in this class, but also model checking (bit-level)
- NEXPTIME
  - same as NP but with exponential time
- P $\subseteq$ NP $\subseteq$ PSPACE $\subseteq$ NEXPTIME
  - usually it is assumed: P $\neq$ NP
  - it is further known: NP $\neq$ NEXPTIME



Complexity Concretely

- NP problems
  - *anything* which can be (polynomially) encoded into SAT
  - combinational equivalence checking, bounded model checking
- PSPACE problems
  - *anything* which can be encoded (polynomially) into QBF
  - or into (bit-level) symbolic model checking
  - sequential equivalence checking, combinational synthesis or bounded games
- NEXPTIME problems
  - *anything* which can be encoded **exponentially** into SAT
  - first-order logic Bernays-Schönfinkel class (EPR): no functions, $\exists^*\forall^*$ prefix
  - QBF with explicit dependencies (Henkin Quantifiers): DQBF
  - partial observation games, black-box bounded model checking
  - bit-vector logics: QF_BV2

NEXPTIME Completeness

- QF_BV2 contained in NEXPTIME
  - bit-blast (single exponentially)
  - give resulting formula to SAT solver
- show QF_B2 NEXPTIME hardness by reducing DQBF to QF_BV2

$$\forall x_0, x_1, x_2, x_3, x_4 \, \exists e_0(x_0, x_1, x_2, x_3), e_1(x_1, x_2, x_3, x_4) \; \varphi$$

1. replace DQBF variables by 32 bit-vector variables $X_i^{[32]}, E_j^{[32]}$
2. replace conjunction, disjunction, negation, by bit-wise operations
3. add independence constraints, e.g., e_0 independent from x_4: “$e_0|_{x_4} = e_0|_{\overline{x_4}}$”
4. enumerate all combinations of universal variables (function-table):
   - these combinations are called binary magic numbers $M_i^{[32]} = X_i^{[32]}$
   - used for “cofactoring” too: $(E_0^{[32]} \mathbin{\&} M_4^{[32]}) = (E_0^{[32]} \mathbin{\&} \sim M_4^{[32]}) >> 1$
   - binary magic numbers can be generated polynomially

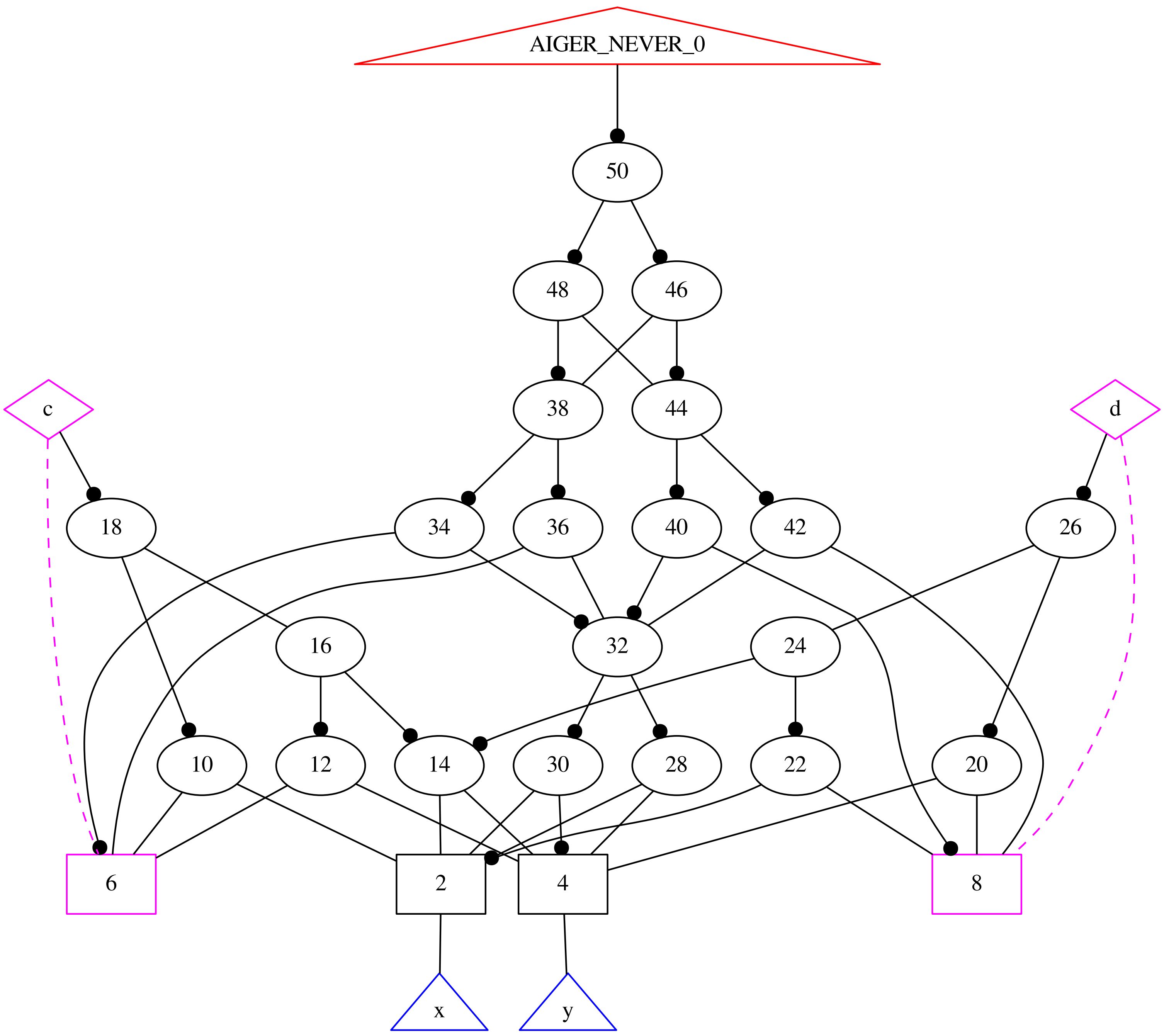
- NP complete: QF_BV2$_{bw}$
  - equality and all bit-wise operators
  - similar to well-known Ackermann reduction:
    - domain can be restricted to be the same size as the number of variables
    - thus bit-vector sizes can be reduced to logarithm of number of variables
  - adapted from Johannsen [PhD Thesis ’02] to binary encoding
- PSPACE complete: QF_BV2$_{bw,\ll 1}$
  - only allow operators which relate neighbouring bits:
    - base operators: equality, inequality, bit-wise ops, shift-by-one
    - extended operators: addition, multiplication by constants, single-bit-slices etc.
  - encode in symbolic model checking logarithmically in bit-width
  - adapted from Spielmann, Kuncak [IJCAR’12] to fixed size bit-vectors

    related to early work by Bernard Boigelot
  - extensions to a larger sub-set
- see our CSR’12, SMT’13 papers (as well as our journal draft)

Commutativity of Bit-Vector Addition in SMV


```
MODULE main
VAR
  c : boolean;     -- carry 'bvadd x y'
  d : boolean;     -- carry 'bvadd y x'
  x : boolean;     -- x0, x1, ...
  y : boolean;     -- y0, y1, ...
ASSIGN
  init (c) := FALSE;
  init (d) := FALSE;
ASSIGN
  next (c) := c & x | c & y | x & y;
  next (d) := d & y | d & x | y & x;
DEFINE
  o := c != (x != y);
  p := d != (y != x);
SPEC
  AG (o = p)
```

Commutativity of Bit-Vector Addition in AIGER



- companies reluctant to publish word-level models
  - thus we do not really have benchmarks
  - also need properties
- no publically available flow from HDL to word-level models
- front-ends do not give us proper word-level models
  - originally designed with bit-blasting in mind
  - much more choices on word-level modelling languages
- sequential extension of BTOR (see our BPR’08 paper)
  - we are working on a new sequential version of BTOR
  - AIGER style

Lambdas

- lambda's can be used to represent array updates (e.g. UCLID)
- our DIFTS'13 paper: lemmas-on-demand for lambdas
- various applications:
  - $write(a,i,e)$:
    $\lambda j\ .\ ite(i = j, e, read(a,j))$
  - $memset(a,i,n,e)$:
    $\lambda j\ .\ ite(i \le j \wedge j < i+n, e, read(a,j))$
  - $memcpy(a,b,i,k,n)$:
    $\lambda j\ .\ ite(k \le j \wedge j < k+n, read(a, i+j-k), read(b,j))$
  - equivalence checking of different address logic in HW
  - ...

- lemmas-on-demand
  - originally proposed by [DeMoura'03]
  - implements a CEGAR loop: extremely lazy CDCL(T) / DPLL (T)
  - checks model guessed by SAT solver for theory consistency
  - used in Boolector for *arrays* and *lambdas*
- use dont'care reasoning to obtain **partial models**
  - shorter lemmas
  - related to generalization in IC3
  - future work: online version
- see our FMCAD'14 paper

- new 2.0 release for FMCAD'14: http://fmv.jku.at/boolector
- support for *lambdas* [DIFTS'13] and *uninterpreted functions*
- had to remove support for extensional arrays
- way *faster model generation*
- C and Python interface
- model based tester
- latest Lingeling
- cloning

FMCAD'14, Thursday, 16:15 - 16:45

Aina Niemetz, Mathias Preiner and Armin Biere.

Turbo-Charging Lemmas on Demand with Don't Care Reasoning.

- new 2.0 release for FMCAD'14: http://fmv.jku.at/boolector
- support for *lambdas* [DIFTS'13] and *uninterpreted functions*
- had to remove support for extensional arrays
- way *faster model generation*
- C and Python interface
- model based tester
- latest Lingeling
- cloning

Thank You!

FMCAD'14, Thursday, 16:15 - 16:45

Aina Niemetz, Mathias Preiner and Armin Biere.

Turbo-Charging Lemmas on Demand with Don't Care Reasoning.