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
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

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

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)

<table>
<thead>
<tr>
<th>Solver</th>
<th>Errors</th>
<th>Solved</th>
<th>Not Solved</th>
<th>Remaining</th>
<th>CPU Time on solved instances</th>
<th>Weighted medal score weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>Boolector (justification)</td>
<td>0</td>
<td>6413</td>
<td>44</td>
<td>0</td>
<td>53176.27</td>
<td>3</td>
</tr>
<tr>
<td>Boolector (dual propagation)</td>
<td>0</td>
<td>6410</td>
<td>47</td>
<td>0</td>
<td>69040.03</td>
<td>3</td>
</tr>
<tr>
<td>[MathSAT]</td>
<td>0</td>
<td>6394</td>
<td>63</td>
<td>0</td>
<td>73535.00</td>
<td>3</td>
</tr>
<tr>
<td>SONOLAR</td>
<td>0</td>
<td>6386</td>
<td>71</td>
<td>0</td>
<td>53248.38</td>
<td>3</td>
</tr>
<tr>
<td>CVC4</td>
<td>0</td>
<td>6352</td>
<td>105</td>
<td>0</td>
<td>78865.09</td>
<td>3</td>
</tr>
<tr>
<td>[Z3]</td>
<td>0</td>
<td>6351</td>
<td>106</td>
<td>0</td>
<td>53957.15</td>
<td>3</td>
</tr>
<tr>
<td>Yices2</td>
<td>1</td>
<td>6410</td>
<td>46</td>
<td>0</td>
<td>37112.15</td>
<td>3</td>
</tr>
<tr>
<td>Kleaver-STP</td>
<td>56</td>
<td>5827</td>
<td>574</td>
<td>0</td>
<td>1120.08</td>
<td>3</td>
</tr>
<tr>
<td>Kleaver-portfolio</td>
<td>91</td>
<td>5799</td>
<td>567</td>
<td>0</td>
<td>3403.29</td>
<td>3</td>
</tr>
</tbody>
</table>
Almost All Binary Search Implementations are Broken

```c
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
```

Challenges in Bit-Precise Reasoning @ FMCAD'14
- common “word-level” operators \( \text{QF\_BV} \) standard SMTLIB2 format
  - constants: 0x7fffffff, variables: fixed size bit vectors \( \text{bool} \ x[32] \)
  - predicates: equality “\( x = y \)”, inequality “\( x \leq y \)” (signed & unsigned)
  - bit-wise logical ops: negation, conjunction, xor \( \sim x \ x \ & \ y \ x \ ^\sim \ y \)
  - 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 \( \text{(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 ⊆ A)
  - extensional arrays:
    - check satisfiability assuming equality of (updated) arrays
    - `a = write (b, j, v) ∧ read (a, j) ≠ 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
addition of 4-bit numbers $x, y$ with result $s$ also 4-bit: \[ s = x + y \]

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

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

where

\[
\begin{align*}
[s, o]_2 &= \text{FullAdder}(x, y, i) \quad \text{with} \\
 s &= x \lor y \lor i \\
o &= (x \land y) \lor (x \lor i) \lor (y \land i) = ((x + y + i) \geq 2)
\end{align*}
\]
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

\[ x \oplus y \equiv (\overline{x} \land y) \lor (x \land \overline{y}) \equiv (\overline{x} \land y) \land (x \land \overline{y}) \]

negation/sign are edge attributes
not part of node
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
Tseitin Transformation: Encode Circuit to CNF

CNF

\[ o \land (x \rightarrow a) \land (x \rightarrow c) \land (x \leftarrow a \land c) \land \ldots \]

\[ o \land (\bar{x} \lor a) \land (\bar{x} \lor c) \land (x \lor \bar{a} \lor \bar{c}) \land \ldots \]
Boolector Architecture

**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_SLL_NODE = 11,
    BTOR_BV_VAR_NODE = 2,
    BTOR_SRL_NODE = 12,
    BTOR_PARAM_NODE = 3,
    BTOR_UDIV_NODE = 13,
    BTOR_SLICE_NODE = 4,
    BTOR_UREM_NODE = 14,
    BTOR_AND_NODE = 5,
    BTOR_CONCAT_NODE = 15,
    BTOR_BEQ_NODE = 6,
    BTOR_APPLY_NODE = 16,
    BTOR_FEQ_NODE = 7,
    BTOR_LAMBDA_NODE = 17,
    BTOR_ADD_NODE = 8,
    BTOR_BCOND_NODE = 18,
    BTOR_MUL_NODE = 9,
    BTOR_ARGS_NODE = 19,
    BTOR_ULT_NODE = 10,
    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
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: \textbf{138 bytes}

bit-blasting with our SMT solver Boolector
\begin{itemize}
  \item rewriting turned off
  \item except structural hashing
  \item produces AIGER circuits of file size \textbf{103 MB}
\end{itemize}

Tseitin transformation leads to CNF in DIMACS format of size \textbf{1 GB}
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):

<table>
<thead>
<tr>
<th>encoding</th>
<th>quantifiers</th>
<th>no</th>
<th>yes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>uninterpreted functions</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
<tr>
<td>unary</td>
<td>NP</td>
<td>NP</td>
<td>PSPACE</td>
</tr>
<tr>
<td></td>
<td>QF_BV1</td>
<td>QF_UFBV1</td>
<td>BV1</td>
</tr>
<tr>
<td></td>
<td>obvious</td>
<td>Ackermann</td>
<td>[TACAS’10]</td>
</tr>
<tr>
<td>binary</td>
<td>NEXPTIME</td>
<td>NEXPTIME</td>
<td>?</td>
</tr>
<tr>
<td></td>
<td>QF_BV2</td>
<td>QF_UFBV2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>[SMT’12]</td>
<td>[SMT’12]</td>
<td></td>
</tr>
<tr>
<td>binary</td>
<td>NEXPTIME</td>
<td>NEXPTIME</td>
<td>2NEXPTIME</td>
</tr>
<tr>
<td></td>
<td>UFBV2</td>
<td></td>
<td>UFBV2</td>
</tr>
<tr>
<td></td>
<td>[SMT’12]</td>
<td></td>
<td>[SMT’12]</td>
</tr>
</tbody>
</table>

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 polynomially **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 ⊆ NP ⊆ PSPACE ⊆ NEXPTIME**
  - usually it is assumed:  $P \neq NP$
  - it is further known:  $NP \neq NEXPTIME$
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_BV2 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: \( (E_0^{[32]} \& M_4^{[32]}) = (E_0^{[32]} \& \overline{M}_4^{[32]}) \gg 1 \)
   - binary magic numbers can be generated polynomially
Bit-Wise Operators and Shifting Neighbouring Bits Only

- **NP complete:** $\text{QF\_BV2}_{bw}$
  - 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:** $\text{QF\_BV2}_{bw,<<1}$
  - 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)
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:
  
  - \texttt{write}(a, i, e):
    \begin{align*}
    \lambda j . \text{ite}(i = j, e, \text{read}(a, j))
    \end{align*}
  
  - \texttt{memset}(a, i, n, e):
    \begin{align*}
    \lambda j . \text{ite}(i \leq j \land j < i + n, e, \text{read}(a, j))
    \end{align*}
  
  - \texttt{memcpy}(a, b, i, k, n):
    \begin{align*}
    \lambda j . \text{ite}(k \leq j \land j < k + n, \text{read}(a, i + j - k), \text{read}(b, j))
    \end{align*}

- equivalence checking of different address logic in HW
Dual Propagation

- 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 don’t’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
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.