Using High Performance SAT and QBF Solvers

Armin Biere

Institute for Formal Models and Verification
Johannes Kepler University
Linz, Austria

Theorem Proving Tools for Program Analysis
Tutorial co-located with POPL 2011
Austin, Texas, USA

Monday, January 24, 2011
Results of the SAT competition/race winners on the SAT 2009 application benchmarks, 20mn timeout

- Limmat 02
- Zchaff 02
- Berkmin 561 02
- Forklift 03
- Siege 03
- Zchaff 04
- SatELite 05
- Minisat 2.0 06
- Picosat 07
- Rsat 07
- Minisat 2.1 08
- Precosat 09
- Glucose 09
- Clasp 09
- Cryptominisat 10
- Lingeling 10
- Minisat 2.2 10

CPU Time (in seconds)
Number of problems solved

[Le Berre'10]
SAT as Core Technology

Formal Specification

Formal Synthesis

Synchronous Languages

Compiler

Theorem Proving

Model Checking

Equivalence Checking

UML

ASM

VDM

Z

B-Method

SAT

SMT

Using High Performance SAT and QBF Solvers

Armin Biere – FMV – JKU Linz
• propositional logic:
  – variables \( \text{tie} \hspace{1em} \text{shirt} \)
  – negation \( \neg \) (not)
  – disjunction \( \lor \) disjunction (or)
  – conjunction \( \land \) conjunction (and)

• three conditions / clauses:
  – clearly one should not wear a \( \text{tie} \) without a \( \text{shirt} \)  \( \neg \text{tie} \lor \text{shirt} \)
  – not wearing a \( \text{tie} \) nor a \( \text{shirt} \) is impolite  \( \text{tie} \lor \text{shirt} \)
  – wearing a \( \text{tie} \) and a \( \text{shirt} \) is overkill  \( \neg (\text{tie} \land \text{shirt}) \equiv \neg \text{tie} \lor \neg \text{shirt} \)

• is the formula  \((\neg \text{tie} \lor \text{shirt}) \land (\text{tie} \lor \text{shirt}) \land (\neg \text{tie} \lor \neg \text{shirt})\) satisfiable?
What is SAT?

- a class of rather low-level kind of problems:
  - propositional variables only, e.g. either hold (true) or not (false)
  - logic operators $\neg$, $\lor$, $\land$, actually restricted to conjunctive normal form (CNF)
  - but no quantifiers such as “for all such things”, or “there is one such thing”
  - can we find an assignment of the variables to true or false, such that a set of clauses is satisfied simultaneously

- theory: it is the standard NP complete problem [Cook’70]

- encoding: how to get your problem into CNF

- simplifying: how can the problem or the CNF be simplified (preprocessing)

- solving: how to implement fast solvers
• Davis and Putnam procedure
  – **DP**: elimination procedure [DavisPutnam’60]
  – **DPLL**: splitting [DavisLogemannLoveland’62]

• modern SAT solvers are mostly based on **DPLL**, actually **CDCL**
  CDCL = Conflict Driven Clause Learning
  – learning: GRASP [MarquesSilvaSakallah’96], RelSAT [BayardoSchrag’97]
  – watched literals, VSIDS: [mz]Chaff [MoskewiczMadiganZhaoZhangMalik-DAC’01]
  – improved heuristics: MiniSAT [EénSörensson-SAT’03] actually version from 2005

• preprocessing is still a hot topic:
  – most practical solvers use SatELite style preprocessing [EénBiere’05] **DP**
  – *inprocessing* in fastest available solvers PrecoSAT, Lingeling, CryptoMiniSAT, …
What is Satisfiability Modulo Theory (SMT)?

- satisfiability solving for first order formulae
  - extension of SAT but interpreted over fixed theories
  - originally without quantifiers but quantifiers are important
  - fully automatic decision procedures which also can provide models

- theories of interest
  - equality, uninterpreted functions
  - real / integer arithmetic
  - bit-vectors, arrays

- particularly important are bit-vectors and arrays for HW/SW verification
  - our SMT solver Boolector ranked #1 in this category (SMT 2008 + 2009)
Applications of SAT and SMT

• **bounded model checking** in electronic design automation (EDA)
  – routinely used for falsification in all major design houses
  – unbounded extensions also use SAT, e.g. *sequential* equivalence checking

• SAT as working horse in *static software verification*

• static device driver verification at Microsoft (SLAM, SDV)
  – predicate abstraction with SMT solvers
  – spurious counter example checking

• *software configuration*, e.g. Eclipse IDE ships with SAT4J

• cryptanalysis and other *combinatorial problems* (bio-informatics)
QBF can be seen as extension to SAT:

- existentially quantified variables as in SAT
- but some variables can be universally quantified

QBF is the *the classical* PSPACE complete problem

- as SAT is *the* NP-complete problem

- two other important PSPACE complete problems:
  * (Propositional) Linear Temporal Logic (LTL) satisfiability
  * symbolic model checking / symbolic reachability
• semantics given as **expansion** of quantifiers

\[
\exists x[f] \equiv f[0/x] \lor f[1/x] \quad \forall x[f] \equiv f[0/x] \land f[1/x]
\]

• expansion as translation from SAT to QBF is exponential
  
  – SAT problems have only existential quantifiers
  
  – expansion of universal quantifier can double formula size

• large number of different approaches to solve QBF versus “mono-culture” in SAT
  
  – scalability for practically interesting problem still an issue
  
  – nevertheless first real applications appear, e.g. black-box equivalence checking
  
  – steady progress: currently fastest solvers **DepQBF** and **Qube**
Bounded Model Checking (BMC)

• look only for counter example made of $k$ states  
  \[ "k" = \text{the bound} \]

\[ I(s_0) \land T(s_0, s_1) \land \cdots \land T(s_{k-1}, s_k) \land \bigvee_{i=0}^{k} \neg p(s_i) \]

• simple for safety properties  \[ p \] is invariantly true  
  \[ (e.g. p = \neg B) \]

\[ I(s_0) \land T(s_0, s_1) \land \cdots \land T(s_{k-1}, s_k) \land \bigwedge_{i=0}^{k} \neg p(s_i) \land \exists l \ T(s_k, s_l) \]

• harder for liveness properties  \[ p \] is eventually true
Bounded Model Checking (BMC)

- look only for counter example made of \( k \) states  
  \[ \text{“} k \text{” = the bound} \]

- simple for safety properties  
  \( p \) is invariantly true  
  \[ I(s_0) \land T(s_0, s_1) \land \cdots \land T(s_{k-1}, s_k) \land \bigvee_{i=0}^{k} \neg p(s_i) \]  

- harder for liveness properties  
  \( p \) is eventually true  
  \[ I(s_0) \land T(s_0, s_1) \land \cdots \land T(s_{k-1}, s_k) \land \bigwedge_{i=0}^{k} \neg p(s_i) \land \bigvee_{l=0}^{k} T(s_k, s_l) \]
\[ \forall s_0 [\text{empty}(s_0) \to \\
\exists x_1 [\text{circle}(s_0, x_1, s_1) \land \\
\forall y_2 [\text{cross}(s_1, y_2, s_2) \to \\
\exists x_3 [\text{circle}(s_2, x_3, s_3) \land \\
\forall y_4 [\text{cross}(s_3, y_4, s_4) \to \\
\exists x_5 [\text{circle}(s_4, x_5, s_5) \land \\
\forall y_6 [\text{cross}(s_5, y_6, s_6) \to \\
\exists x_7 [\text{circle}(s_6, x_7, s_7) \land \\
(9 \times 3 \text{ bits each}) \\
\forall y_8 [\text{cross}(s_7, y_8, s_8) \to \\
\exists x_9 [\text{circle}(s_8, x_9, s_9) \land \text{win}_{\text{circle}}(s_9)]]]]]]] \]
Example: Equivalence Checking If-then-else Chains

<table>
<thead>
<tr>
<th>original code</th>
<th>optimized code</th>
</tr>
</thead>
<tbody>
<tr>
<td>if(!a &amp;&amp; !b) h(); else if(!a) g(); else f();</td>
<td>if(a) f(); else if(b) g(); else h();</td>
</tr>
<tr>
<td>↓</td>
<td>↑</td>
</tr>
<tr>
<td>if(!a) { if(!b) h(); else g(); } else f();</td>
<td>if(a) f(); else { if(!b) h(); else g(); }</td>
</tr>
</tbody>
</table>

How to check that these two versions are equivalent?
1. represent procedures as *independent* boolean variables

\[
\text{original} := \begin{align*}
\text{if } & \neg a \land \neg b \text{ then } h \\
\text{else if } & \neg a \text{ then } g \\
\text{else } & f
\end{align*}
\]

\[
\text{optimized} := \begin{align*}
\text{if } & a \text{ then } f \\
\text{else if } & b \text{ then } g \\
\text{else } & h
\end{align*}
\]

2. compile if-then-else chains into boolean formulae

\[
\text{compile}\left(\text{if } x \text{ then } y \text{ else } z\right) \equiv (x \land y) \lor (\neg x \land z)
\]

3. check equivalence of boolean formulae

\[
\text{compile}(\text{original}) \iff \text{compile}(\text{optimized})
\]
original $\equiv$ if $\neg a \land \neg b$ then $h$ else if $\neg a$ then $g$ else $f$
$\equiv$ $(\neg a \land \neg b) \land h \lor \neg(\neg a \land \neg b) \land$ if $\neg a$ then $g$ else $f$
$\equiv$ $(\neg a \land \neg b) \land h \lor \neg(\neg a \land \neg b) \land (\neg a \land g \lor a \land f)$

optimized $\equiv$ if $a$ then $f$ else if $b$ then $g$ else $h$
$\equiv$ $a \land f \lor \neg a \land$ if $b$ then $g$ else $h$
$\equiv$ $a \land f \lor \neg a \land (b \land g \lor \neg b \land h)$

$$(\neg a \land \neg b) \land h \lor \neg(\neg a \land \neg b) \land (\neg a \land g \lor a \land f) \iff a \land f \lor \neg a \land (b \land g \lor \neg b \land h)$$
How to Check (In)Equivalence?

Reformulate it as a satisfiability (SAT) problem:

Is there an assignment to $a, b, f, g, h,$
which results in different evaluations of original and optimized?

or equivalently:

Is the boolean formula $\text{compile(original)} \not\leftrightarrow \text{compile(optimized)}$ satisfiable?

such an assignment would provide an easy to understand counterexample
\[ b \lor a \land c \quad \Leftrightarrow \quad (a \lor b) \land (b \lor c) \]
**Definition** formula in *Conjunctive Normal Form* (CNF) is a conjunction of clauses 

\[ C_1 \land C_2 \land \ldots \land C_n \]

each *clause* \( C \) is a disjunction of literals 

\[ C = L_1 \lor \ldots \lor L_m \]

and each *literal* is either a plain variable \( x \) or a negated variable \( \overline{x} \).

**Example**  

\[ (a \lor b \lor c) \land (\overline{a} \lor \overline{b}) \land (\overline{a} \lor \overline{c}) \]

**Note 1:** two notions for negation: in \( \overline{x} \) and \( \neg \) as in \( \neg x \) for denoting negation.

**Note 2:** original SAT problem is actually formulated for CNF

**Note 3:** solvers (mostly) expect CNF as input
• common ASCII file format of SAT solvers, used by SAT competitions

• variables are represented as natural numbers, literals as integers

• header “p cnf <vars> <clauses>”, comment lines start with “c”

In order to show the validity of
\[ b \lor a \land c \iff (a \lor b) \land (b \lor c) \]
	negate,
\[ \overline{(b \lor a \land c)} \land (a \lor b) \land (b \lor c) \]
simplify and show unsatisfiability of
\[ \overline{b} \land (\overline{a} \lor \overline{c}) \land (a \lor b) \land (b \lor c) \]

c the first two lines are comments
c ex1.cnf: a=1, b=2, c=3
p cnf 3 4
-2 0
-1 -3 0
1 2 0
2 3 0
// compile with: gcc -o ex1 ex1.c picosat.o
#include "picosat.h"
#include <stdio.h>
int main () {
    int res;
    picosat_init ();

    picosat_add (-2); picosat_add (0);
    picosat_add (-1); picosat_add (-3); picosat_add (0);
    picosat_add (1); picosat_add (2); picosat_add (0);
    picosat_add (2); picosat_add (3); picosat_add (0);

    res = picosat_sat (-1);

    if (res == 10) printf ("s SATISFIABLE\n");
    else if (res == 20) printf ("s UNSATISFIABLE\n");
    else printf ("s UNKNOWN\n");

    picosat_reset ();
    return res;
}
assume invalid equivalence resp. implication: \[(a \lor b) \Rightarrow (a \text{ xor } b)\]

its negation

as CNF

\[(a \lor b) \land (a = b)\]

\[(a \lor b) \land (\neg a \lor b) \land (\neg b \lor a)\]

c ex2.cnf: a=1,b=2

p cnf 2 3
1 2 0
-1 2 0
-2 1 0

SAT solver then allows to extract one satisfying assignment:

$\$ picosat ex2.cnf
s SATISFIABLE
v 1 2 0

this is the only one since “assuming” the opposite values individually is UNSAT

$\$ picosat ex2.cnf -a -1; picosat ex2.cnf -a -2
s UNSATISFIABLE
s UNSATISFIABLE
Example of Tseitin Transformation: Circuit to CNF

CNF

\[ o \land (x \leftrightarrow a \land c) \land (y \leftrightarrow b \lor x) \land (u \leftrightarrow a \lor b) \land (v \leftrightarrow b \lor c) \land (w \leftrightarrow u \land v) \land (o \leftrightarrow y \oplus w) \]

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

\[ o \land (\overline{x} \lor a) \land (\overline{x} \lor c) \land (x \lor \overline{a} \lor \overline{c}) \land \ldots \]
1. generate a new variable $x_s$ for each non input circuit signal $s$

2. for each gate produce complete input / output constraints as clauses

3. collect all constraints in a big conjunction

the transformation is *satisfiability equivalent*:

the result is satisfiable iff and only the original formula is satisfiable

not equivalent to the original formula: it has new variables

just *project* satisfying assignment onto the original variables
Tseitin Transformation: Input / Output Constraints

Negation:  \( x \leftrightarrow \bar{y} \iff (x \rightarrow \bar{y}) \land (\bar{y} \rightarrow x) \)
\[ \iff (\bar{x} \lor \bar{y}) \land (y \lor x) \]

Disjunction:  \( x \leftrightarrow (y \lor z) \iff (y \rightarrow x) \land (z \rightarrow x) \land (x \rightarrow (y \lor z)) \)
\[ \iff (\bar{y} \lor x) \land (\bar{z} \lor x) \land (\bar{x} \lor y \lor z) \]

Conjunction:  \( x \leftrightarrow (y \land z) \iff (x \rightarrow y) \land (x \rightarrow z) \land ((y \land z) \rightarrow x) \)
\[ \iff (\bar{x} \lor y) \land (\bar{x} \lor z) \land ((y \land z) \lor x) \]
\[ \iff (\bar{x} \lor y) \land (\bar{x} \lor z) \land (\bar{y} \lor \bar{z} \lor x) \]

Equivalence:  \( x \leftrightarrow (y \leftrightarrow z) \iff (x \rightarrow (y \leftrightarrow z)) \land ((y \leftrightarrow z) \rightarrow x) \)
\[ \iff (x \rightarrow ((y \rightarrow z) \lor (z \rightarrow y)) \land ((y \leftrightarrow z) \lor (y \land z)) \rightarrow x) \]
\[ \iff (x \rightarrow ((y \rightarrow z) \lor (z \rightarrow y)) \land ((y \leftrightarrow z) \lor (y \land z)) \rightarrow x) \]
\[ \iff (\bar{x} \lor y \lor z) \land (\bar{x} \lor z \lor y) \land (y \land z \lor \bar{y} \lor \bar{z}) \rightarrow x) \]
\[ \iff (\bar{x} \lor y \lor z) \land (\bar{x} \lor z \lor y) \land (y \land z \lor \bar{y} \lor \bar{z}) \rightarrow x) \]
\[ \iff (\bar{x} \lor y \lor z) \land (\bar{x} \lor z \lor y) \land (y \land z \lor \bar{y} \lor \bar{z}) \rightarrow x) \]
\[ \iff (\bar{x} \lor y \lor z) \land (\bar{x} \lor z \lor y) \land (y \land z \lor \bar{y} \lor \bar{z}) \rightarrow x) \]
• goal is smaller CNF
  less variables, less clauses, so easier to solve (?!)

• extract multi argument operands to remove variables for intermediate nodes

• half of AND, OR node constraints/ clauses can be removed for *unnegated* nodes
  [PlaistedGreenbaum'86]
  – node occurs negated if it has an ancestor which is a negation
  – half of the constraints determine parent assignment from child assignment
  – those are unnecessary if node is not used negated
  – those have to be carefully applied to DAG structure

• further structural circuit optimizations …
CNF Blocked Clause Elimination simulates many encoding / circuit optimizations

[JarvisaloBiereHeule-TACAS'10]
Intermediate Representations

- encoding directly into CNF is hard, so we use intermediate levels:

  1. application level

  2. bit-precise semantics world-level operations: bit-vector theory

  3. bit-level representations such as AIGs or vectors of AIGs

  4. CNF

- encoding application level formulas into word-level: as generating machine code

- word-level to bit-level: bit-blasting similar to hardware synthesis

- encoding “logical” constraints is another story
addition of 4-bit numbers $x, y$ with result $s$ also 4-bit: \[ s = x + y \]

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

\[
[s_3, \cdot]_2 = \text{FullAdder}(x_3, y_3, c_2) \\
[s_2, c_2]_2 = \text{FullAdder}(x_2, y_2, c_1) \\
[s_1, c_1]_2 = \text{FullAdder}(x_1, y_1, c_0) \\
[s_0, c_0]_2 = \text{FullAdder}(x_0, y_0, \text{false})
\]

where

\[
[s, o]_2 = \text{FullAdder}(x, y, i) \quad \text{with}
\]

\[
s = x \text{ xor } y \text{ xor } i \\
o = (x \land y) \lor (x \land i) \lor (y \land i) = ((x + y + i) \geq 2)
\]
And-Inverter-Graphs (AIG)

- widely adopted bit-level intermediate representation
  - see for instance our AIGER format [http://fmv.jku.at/aiger](http://fmv.jku.at/aiger)
  - used in Hardware Model Checking Competition (HWMCC)
  - also used in the *structural track* in 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 …

Using High Performance SAT and QBF Solvers Armin Biere — FMV — JKU Linz
XOR as AIG

negation/sign are edge attributes
not part of node

\[ x \text{ xor } y \equiv (\overline{x} \land y) \lor (x \land \overline{y}) \equiv (\overline{x} \land y) \land (x \land \overline{y}) \]
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
Encoding Logical Constraints

• Tseitin’s construction suitable for most kinds of “model constraints”
  – assuming simple operational semantics: encode an interpreter
  – small domains: one-hot encoding  large domains: binary encoding

• harder to encode properties or additional constraints
  – temporal logic / fix-points
  – environment constraints

• example for fix-points / recursive equations: $x = (a \lor y)$, $y = (b \lor x)$
  – has unique least fix-point $x = y = (a \lor b)$
  – and unique largest fix-point $x = y = true$ but unfortunately
  – only largest fix-point can be (directly) encoded in SAT otherwise need ASP
Example of Logical Constraints:  Cardinality Constraints

- given a set of literals \( \{l_1, \ldots, l_n\} \)
  - constraint the \textit{number} of literals assigned to \textit{true}
    - \(|\{l_1, \ldots, l_n\}| \geq k\) or \(|\{l_1, \ldots, l_n\}| \leq k\) or \(|\{l_1, \ldots, l_n\}| = k\)

- multiple encodings of cardinality constraints
  - naïve encoding exponential: \textit{at-most-two} quadratic, \textit{at-most-three} cubic, etc.
  - quadratic \(O(k \cdot n)\) encoding goes back to Shannon
  - linear \(O(n)\) parallel counter encoding [Sinz’05]
  - for an \(O(n \cdot \log n)\) encoding see Prestwich’s chapter in our Handbook of SAT

- generalization \textit{Pseudo-Boolean} constraints (PB), e.g.
  \[2 \cdot \overline{a} + \overline{b} + c + \overline{d} + 2 \cdot e \geq 3\]
  actually used to handle MaxSAT in SAT4J for configuration in Eclipse
2 \leq |\{l_1, \ldots, l_9\}| \leq 3

\begin{align*}
&l_1 \quad - \quad - \quad - \quad - \quad l_3 \quad - \quad - \quad - \quad - \quad l_4 \quad - \quad - \quad - \quad - \quad l_5 \quad - \quad - \quad - \quad - \quad l_6 \quad - \quad - \quad - \quad - \quad l_7 \quad - \quad - \quad - \quad - \quad l_8 \quad - \quad - \quad - \quad - \quad l_9 \quad - \quad - \quad 0 \\
&l_2 \quad - \quad - \quad - \quad - \quad l_3 \quad - \quad - \quad - \quad - \quad l_4 \quad - \quad - \quad - \quad - \quad l_5 \quad - \quad - \quad - \quad - \quad l_6 \quad - \quad - \quad - \quad - \quad l_7 \quad - \quad - \quad - \quad - \quad l_8 \quad - \quad - \quad - \quad - \quad l_9 \quad - \quad - \quad 0 \\
&l_3 \quad - \quad - \quad - \quad - \quad l_4 \quad - \quad - \quad - \quad - \quad l_5 \quad - \quad - \quad - \quad - \quad l_6 \quad - \quad - \quad - \quad - \quad l_7 \quad - \quad - \quad - \quad - \quad l_8 \quad - \quad - \quad - \quad - \quad l_9 \quad - \quad - \quad 1 \\
&l_4 \quad - \quad - \quad - \quad - \quad l_5 \quad - \quad - \quad - \quad - \quad l_6 \quad - \quad - \quad - \quad - \quad l_7 \quad - \quad - \quad - \quad - \quad l_8 \quad - \quad - \quad - \quad - \quad l_9 \quad - \quad - \quad 1 \\
&0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0 \quad 0
\end{align*}

“then” edge downward, “else” edge to the right
// compile with: gcc -o ex2 ex2.c picosat.o
#include "picosat.h"
#include <stdio.h>
#include <assert.h>

int main () {
    int res, a, b;
    picosat_init ();
    picosat_add (1); picosat_add (2); picosat_add (0);
    picosat_add (-1); picosat_add (2); picosat_add (0);
    picosat_add (-2); picosat_add (1); picosat_add (0);
    assert (picosat_sat (-1) == 10); // SATISFIABLE
    a = picosat_deref (1); b = picosat_deref (2);
    printf ("v %d %d\n", a*1, b*2);
    picosat_assume (-a*1); assert (picosat_sat (-1) == 20); // UNSAT
    picosat_assume (-b*2); assert (picosat_sat (-1) == 20); // UNSAT
    return res;
}
static void block_current_solution (void) {
  int max_idx = picosat_variables (), i;

  // since 'picosat_add' resets solutions
  // need to store it first:
  signed char * sol = malloc (max_idx + 1);
  memset (sol, 0, max_idx + 1);

  for (i = 1; i <= max_idx; i++)
    sol[i] = (picosat_deref (i) > 0) ? 1 : -1;

  for (i = 1; i <= max_idx; i++)
    picosat_add ((sol[i] < 0) ? i : -i);
  picosat_add (0);

  free (sol);
}
Using High Performance SAT and QBF Solvers

Armin Biere – FMV – JKU Linz
Failed Assumptions

- two ways to implement incremental SAT solvers
  - push / pop as in SMT solvers  partial support in SATIRE, zChaff, PicoSAT
    * clauses associated with context and pushed / popped in a stack like manner
    * pop discards clauses of current context
  - most common: assumptions  [ClaessenSörensson’03]  [EénSörensson’03]
    * allows to use set of literals as assumptions
    * force SAT solver to first pick assumption as decisions
    * more flexible, since assumptions can be reused
    * assumptions are only valid for the next SAT call

- failed assumptions: sub set of assumptions inconsistent with CNF
Example: Bit-Vector Under-Approximation

- goal: reduce size of bit-vector constants in satisfying assignments

- refinement approach: for each bit-vector variable only use an “effective width”
  - example: 4-bit vector \([x_3, x_2, x_1, x_0]\) and effective width 2 use \([x_1, x_1, x_1, x_0]\)
  - either encode from scratch with \(x_3\) and \(x_2\) replaced by \(x_1\) (1)
  - or add \(x_3 = x_1\) and \(x_2 = x_1\) after push (2)
  - or add \(a_x^2 \rightarrow x_3 = x_1\) and \(a_x^2 \rightarrow x_2 = x_1\) and assume fresh literal \(a_x^2\) (3)

- if satisfiable then a solution with small constants has been found
  otherwise increase eff. width of bit-vectors where it was used to derive UNSAT
  under-approximations not used then formula UNSAT "used" = "failed assumption"

- in (3) constraints are removed by forcing assumptions to the opposite value
  by adding a unit clause, e.g. \(\neg a_x^2\) in next iteration
Clausal Cores

- clausal core (or unsatisfiable sub set) of an unsatisfiable formula
  - clauses used to derive the empty clause
  - may include not only original but also learned clauses
  - similar application as in previous under-approximation example
  - but also useful for diagnosis of inconsistencies

- variable core
  - sub set of variables occurring in clauses of a clausal core

- these cores are not unique and not necessary minimimal

- minimal unsatisfiable sub set (MUS) = clausal core where no clause can be removed
• PicoMUS is a MUS extractor based on PicoSAT
  – uses several rounds of clausal core extraction for preprocessing
  – then switches to assumption based core minimization using
    `picosat_failed_assumptions`
  – source code serves as a good example on how to use cores / assumptions

• new MUS track in this year’s SAT 2011 competition
  – with high- and low-level MUS sub tracks
Examples for Core and MUS Extraction

```
c ex3.cnf $ picosat ex3.cnf -c core
p cnf 6 10 s UNSATISFIABLE
1 2 3 0 $ cat core
1 2 -3 0 p cnf 6 9
1 -2 3 0 2 3 1 0 c ex4.cnf $ picomus ex4.cnf mus
p cnf 6 11 s UNSATISFIABLE
4 5 6 0 -2 3 1 0 1 2 3 0 $ cat mus
4 5 -6 0 -2 -3 1 0 1 2 -3 0 p cnf 6 6
4 -5 6 0 6 5 4 0 1 -2 3 0 1 2 3 0
4 -5 -6 0 5 -6 4 0 1 -2 -3 0 1 2 -3 0
-1 -4 0 6 4 -5 0 4 5 6 0 1 -2 3 0
1 4 0 4 -6 -5 0 4 5 -6 0 1 -2 3 0
-1 -4 0 4 -5 6 0 -1 4 0 4 -5 -6 0 -1 4 0
-1 -4 0 -1 4 0 -1 -4 0
```

Using High Performance SAT and QBF Solvers Armin Biere – FMV – JKU Linz
Proof-Traces and TraceCheck

• core extraction in PicoSAT is based on tracing proofs
  – enabled by `picosat_enable_trace_generation`
  – maintains “dependency graph” of learned clauses
  – kept in memory, so fast core generation

• traces can also written to disk in various formats
  – RUP format by Allen Van Gelder (SAT competition)
  – or format of TraceCheck tool

• TraceCheck can check traces for correctness
  – orders clauses and antecedents to generate and check resolution proof
  – (binary) resolution proofs can be dumped
same as DIMACS except that we have additional quantifiers:

```
c SAT
p cnf 3 4
a 1 0
e 2 3 0
-1 -2 3 0
-1 2 -3 0
1 2 3 0
1 -2 -3 0
c UNSAT
p cnf 4 8
a 1 2 0
e 3 4 0
-1 -3 4 0
-1 3 -4 0
1 3 4 0
1 -3 -4 0
-2 -3 4 0
-2 3 -4 0
2 3 4 0
2 -3 -4 0
```
/* Create and initialize solver instance. */
QDPLL *qdpl1_create (void);

/* Delete and release all memory of solver instance. */
void qdpl1_delete (QDPLL * qdpl1);

/* Ensure var table size to be at least 'num'. */
void qdpl1_adjust_vars (QDPLL * qdpl1, VarID num);

/* Open a new scope, where variables can be added by 'qdpl1_add'.
   Returns nesting of new scope.
   Opened scope can be closed by adding '0' via 'qdpl1_add'.
   NOTE: will fail if there is an opened scope already. */
unsigned int qdpl1_new_scope (QDPLL * qdpl1, QDPLLQuantifierType qtype);

/* Add variables or literals to clause or opened scope.
   If scope is opened, then 'id' is interpreted as a variable ID,
   otherwise 'id' is interpreted as a literal.
   NOTE: will fail if a scope is opened and 'id' is negative. */
void qdpl1_add (QDPLL * qdpl1, LitID id);

/* Decide formula. */
QDPLLResult qdpl1_sat (QDPLL * qdpl1);
Wish List for QBF Solvers

- extraction of “certificates”
  - satisfying assignment to outer-most existential variables
  - resp. in general skolem-functions for satisfiable instances
  - falsifying assignment to outer-most universal variables

- incremental QBF solving

- API for preprocessing / inprocessing

- beside steady progress more scalability
• SAT and QBF

• conjunctive normal form (CNF) and (Q)DIMACS format

• encoding of models and logical constraints

• PicoSAT, PicoMUS, TraceCheck, DepQBF

• examples and use cases

• APIs
What is missing?

- how SAT can handle millions of variables routinely
- whether there will still be progress in SAT
- AIGER format, proof (trace) formats
- the full API of PicoSAT (see `picosat.h` for more details)
- skipped API of other Solvers, in particular (Crypto)MiniSAT, SAT4J
- tricky issues with incremental pre/in-processing such as “freezing variables”
- compact QBF encodings [JussilaBiere’06]
The increase in power of modern SAT solvers has been phenomenal. It has become the key enabling technology of both computer hardware and software. Bounded Model Checking software is now probably the most widely used model checking technique. As it finds the satisfying instances of a Boolean formula obtained by depth a sequential circuit and its specification in linear temporal logic. Software verification is a much more difficult problem on the frontier. A promising approach for languages like C with finite word-length integers is in BMC but with a decision procedure for the theory of bit-vectors and procedures for bit-vectors that I am familiar with, I am confident that ultimately make use in complex formulas.

More complicated theories, like linear real and integer arithmetic, are verification. Most of them use powerful SAT solvers in an essential way.

Solving is a key technology for 21st century computer science. I expect all theoretical and practical aspects of SAT solving will be extremely interesting to researchers and will lead to many further advances in the field.

Edmund Clarke

Armin Biere
Marijn Heule
Hans van Maaren
Toby Walsh
Blocked Clauses

- blocked clause \( C \in F \)
- all clauses in \( F \) with \( \bar{l} \)

fix a CNF \( F \)

\[
\begin{align*}
(a \lor b \lor l) \quad & (l \lor \bar{a} \lor c) \\
& (l \lor \bar{b} \lor d)
\end{align*}
\]

since all resolvents of \( C \) on \( l \) are tautological \( C \) can be removed

Proof

assignment \( \sigma \) satisfying \( F \setminus C \) but not \( C \)

can be extended to a satisfying assignment of \( F \) by flipping value of \( l \)
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>COI</strong></td>
<td>Cone-of-Influence reduction</td>
</tr>
<tr>
<td><strong>MIR</strong></td>
<td>Monontone-Input-Reduction</td>
</tr>
<tr>
<td><strong>NSI</strong></td>
<td>Non-Shared Inputs reduction</td>
</tr>
<tr>
<td><strong>PG</strong></td>
<td>polarity based encoding</td>
</tr>
<tr>
<td><strong>TST</strong></td>
<td>standard Tseitin encoding</td>
</tr>
<tr>
<td><strong>VE</strong></td>
<td>Variable-Elimination as in DP / Quantor / SATeLite</td>
</tr>
<tr>
<td><strong>BCE</strong></td>
<td>Blocked-Clause-Elimination</td>
</tr>
</tbody>
</table>

Using High Performance SAT and QBF Solvers

Armin Biere – FMV – JKU Linz
Boolector: Lemmas-on-Demand + Underapproximation

Appendix A3/A3

[BrummayerBiere'09]

---

1. **Array formula**
   - **Over-approximate**
     - **Encode to CNF**
     - **Add under-approx. clauses C**
     - **Refine under-approx.**
       - **Under-approx. clauses C used?**
         - **NO**
           - **Call SAT solver**
             - **SAT?**
               - **NO**
                 - **C used?**
                   - **YES**
                     - **Refine over-approx.**
                   - **NO**
                     - **Formula is unsatisfiable**
               - **YES**
                 - **Call SAT solver**
                   - **spurious?**
                     - **YES**
                       - **Add lemma**
                     - **NO**
                       - **Refine over-approx.**

2. **Formula is satisfiable**
   - **Call SAT solver**
     - **SAT?**
       - **YES**
         - **Add lemma**
       - **NO**
         - **Refine over-approx.**