**Institute for Formal Models and Verification**
Johannes Kepler University Linz

VL Logik (LVA-Nr. 342208), Winter Semester 2014/2015

# Satisfiabiliy Modulo Theories Details

Version 2014.2

Armin Biere (biere@jku.at)
Martina Seidl (martina.seidl@jku.at)

# Propositional Skeleton

**Example (arbitrary LRA formula)**

$$x \neq y \;\wedge\; (2 * x \leq z \quad \vee \quad \neg(x - y \geq z \;\wedge\; z \leq y))$$

eliminate $\neq$ by disjunction

$$(\underbrace{x < y}_{a} \vee \underbrace{x > y}_{b}) \;\wedge\; (\underbrace{2 * x \leq z}_{c} \vee \neg(\underbrace{x - y \geq z}_{d} \;\wedge\; \underbrace{z \leq y}_{e}))$$

which is abstracted to a propositional formula called "propositional skeleton"

$$(a \vee b) \wedge (c \vee \neg(d \wedge e)) \qquad \text{with} \quad \alpha(x < y) = a, \quad \alpha(x > y) = b, \ldots$$

SAT solver enumerates solutions, e.g., $a = b = c = d = e = 1$

check solution literals with theory solver, e.g., Fourier-Motzkin

spurious solutions (disproven by theory solver) added as "lemma",
e.g. $\neg(a \wedge b \wedge c \wedge c \wedge d \wedge e)$ or just $\neg(a \wedge b)$ after minimization

continue until SAT solver says *unsatisfiable* or theory solver *satisfiable*

# Lemmas on Demand

this is an extremely "lazy" version of DPLL (T) / CDCL(T)

*LemmasOnDemand*$(\phi)$

$\psi =$ *PropositionalSkeleton*$(\phi)$

let $\alpha$ be the abstraction function, mapping theory literals to prop. literals

while $\psi$ has satisfiable assignment $\sigma$

let $l_1, \ldots, l_n$ be all the theory literals with $\sigma(\alpha(l_i)) = 1$

check conjunction $L = l_1 \wedge \cdots \wedge l_n$ with theory solver

if theory solver returns satisfying assignment $\rho$ return *satisfiable*

determine "small" sub-set $\{k_1, \ldots, k_m\} \subseteq \{l_1, \ldots, l_n\}$ where

$K = k_1 \wedge \cdots \wedge k_m$ remains unsatisfiable (by theory solver)

add lemma $\neg K$ to $\psi$, actually replace $\psi$ by $\psi \wedge \alpha(\neg K)$

return *unsatisfiable*

note that these lemmas $\neg K$ are all clauses

# Minimal Unsatisfiable Set (MUS)

motivation: the lemmas we add in “lemmas on demand” should be small

$$\overbrace{(a \lor \neg b) \land (a \lor b) \land \rlap{$\underbrace{\phantom{(\neg a \lor \neg c) \land (\neg a \lor c) \land (a \lor \neg c) \land (a \lor c)}}_{\text{MUS}}$}(\neg a \lor \neg c) \land (\neg a \lor c)}^{\text{MUS}} \land (a \lor \neg c) \land (a \lor c)$$

- given an unsatisfiable set of “constraints” $S$ (set of literals, or clauses)
- an MUS $M$ is a sub-set $M \subseteq S$ such that
  - $M$ is still unsatisfiable
  - any $M' \subset M$ (with $M' \neq M$) is satisfiable
- so an MUS is a “minimal” inconsistent subset
  - all constraints in the MUS are *necessary* for $M$ to be inconsistent
  - so one minimal way to explain inconsistency of $S$
- note that “being inconsistent” is a monotone property
  - if $A \subseteq B$ is a set of constraints
  - if $A$ is unsatisfiable then $B$ is unsatisfiable
  - essential for algorithms to compute an MUS

# Iterative Destructive Algorithm for MUS Computation

destructive = remove constraints from an over-approximation of an MUS

*IterativeDestructiveMUS*$(S)$

$M = S$
$D = S$
while $D \neq \emptyset$
  pick constraint $C \in D$
  if $M \setminus \{C\}$ unsatisfiable remove $C$ from $M$
  remove $C$ from $D$
return $M$

needs exactly $|S|$ satisfiability checks

any-time algorithm: preliminary result $M$ remains inconsistent

can stop any time

# QuickXplain Variant of MUS Computation

quickly "zoom in" on one MUS (particularly if there is a small one)

*QuickMUSRecursive*$(D)$

if $M \backslash D$ is satisfiable

if $|D| > 1$

let $D = L \cup R$ with $|L|, |R| > 0$ $\quad \ldots \geq \lfloor \frac{|D|}{2} \rfloor$

*QuickMUSRecursive*$(L)$

*QuickMUSRecursive*$(R)$

else remove $D$ from $M$

*QuickMUS*$(S)$

global variable $M = S$

*QuickMUSRecursive*$(S)$

return $M$

needs at most $2 \cdot |S|$ and at least $|M|$ satisfiability checks

# Theory of Arrays

- functions "read" and "write": $\text{read}(a, i), \text{write}(a, i, v)$
- axioms

$$\forall a, i, j \colon i = j \rightarrow \text{read}(a, i) = \text{read}(a, j) \qquad \textit{array congruence}$$

$$\forall a, v, i, j \colon i = j \rightarrow \text{read}(\text{write}(a, i, v), j) = v \qquad \textit{read over write 1}$$

$$\forall a, v, i, j \colon i \neq j \rightarrow \text{read}(\text{write}(a, i, v), j) = \text{read}(a, j) \qquad \textit{read over write 2}$$

- used to model memory (HW and SW)
- eagerly reduce arrays to uninterpreted functions by eliminating "write"

$$\text{read}(\text{write}(a, i, v), j) \quad \text{replaced by} \quad (i = j \ ? \ v : \text{read}(a, j))$$

- more sophisticated non-eager algorithms are usually faster
- such as for instance the lemmas-on-demand algorithm in Boolector

# Simple Array Example

$$i \neq j \ \wedge\ u = \mathrm{read}(\mathrm{write}(a, i, v), j) \ \wedge\ v = \mathrm{read}(a, j) \ \wedge\ u \neq v$$

eliminate “write”

$$i \neq j \ \wedge\ u = (i = j \ ?\ v : \mathrm{read}(a, j)) \ \wedge\ v = \mathrm{read}(a, j) \ \wedge\ u \neq v$$

simplify conditional by assuming “$i \neq j$”

$$i \neq j \ \wedge\ u = \mathrm{read}(a, j) \ \wedge\ v = \mathrm{read}(a, j) \ \wedge\ u \neq v$$

applying congruence for both “read”

$$i \neq j \ \wedge\ u = \mathrm{read}(a, j) = \mathrm{read}(a, j) = v \ \wedge\ u \neq v$$

which is clearly *unsatisfiable*

# More Complex Array Example for Checking Aliasing

| *original* | *optimized* |
|---|---|
| assert (i != k); | int t = a[k]; |
| a[i] = a[k]; | a[i] = t; |
| a[j] = a[k]; | a[j] = t; |

| | |
|---|---|
| $i \neq k$ | $t = \text{read}(a, k)$ |
| $b_1 = \text{write}(a, i, t)$ | $c_1 = \text{write}(a, i, t)$ |
| $b_2 = \text{write}(b_1, j, s)$ | $c_2 = \text{write}(c_1, j, t)$ |
| $s = \text{read}(b_1, k)$ | |

*original* $\neq$ *optimized* iff $b_2 \neq c_2$

$b_2 \neq c_2$ iff $\exists l$ with $\text{read}(b_2, l) \neq \text{read}(c_2, l)$

# Aliasing Example Continued 1

thus *original* $\neq$ *optimized* iff

$$
\begin{aligned}
& i \neq k \\
& t = \mathrm{read}(a, k) \\
& b_1 = \mathrm{write}(a, i, t) \\
& b_2 = \mathrm{write}(b_1, j, s) \\
& c_1 = \mathrm{write}(a, i, t) \\
& c_2 = \mathrm{write}(c_1, j, t) \\
& s = \mathrm{read}(b_1, k) \\
& \mathrm{read}(b_2, l) \neq \mathrm{read}(c_2, l)
\end{aligned}
$$

satisfiable

# Aliasing Example Continued 2

thus *original* $\neq$ *optimized* iff

$$
\begin{aligned}
& i \neq k \\
& t = \mathrm{read}(a, k) \\
& b_1 = \mathrm{write}(a, i, t) \\
& b_2 = \mathrm{write}(b_1, j, s) \\
& c_1 = \mathrm{write}(a, i, t) \\
& c_2 = \mathrm{write}(c_1, j, t) \\
& s = \mathrm{read}(b_1, k) \\
& u = \mathrm{read}(b_2, l) \\
& v = \mathrm{read}(c_2, l) \\
& u \neq v
\end{aligned}
$$

satisfiable

# Aliasing Example Continued 3

after eliminating $c_2$

$$
\begin{aligned}
& i \neq k \\
& t = \text{read}(a, k) \\
& b_1 = \text{write}(a, i, t) \\
& b_2 = \text{write}(b_1, j, s) \\
& c_1 = \text{write}(a, i, t) \\
& c_2 = \text{write}(c_1, j, t) \\
& s = \text{read}(b_1, k) \\
& u = \text{read}(b_2, l) \\
& v = (i = j \;\; ? \;\; t : \text{read}(c_1, l)) \\
& u \neq v
\end{aligned}
$$

# Aliasing Example Continued 4

after eliminating $c_2$, $c_1$

$$
\begin{aligned}
& i \neq k \\
& t = \mathrm{read}(a, k) \\
& b_1 = \mathrm{write}(a, i, t) \\
& b_2 = \mathrm{write}(b_1, j, s) \\
& c_1 = \mathrm{write}(a, i, t) \\
& c_2 = \mathrm{write}(c_1, j, t) \\
& s = \mathrm{read}(b_1, k) \\
& u = \mathrm{read}(b_2, l) \\
& v = (l = j \ ? \ t : (l = i \ ? \ t : \mathrm{read}(a, l))) \\
& u \neq v
\end{aligned}
$$

# Aliasing Example Continued 5

after eliminating $c_2$, $c_1$, $b_2$

$$
\begin{aligned}
& i \neq k \\
& t = \text{read}(a, k) \\
& b_1 = \text{write}(a, i, t) \\
& b_2 = \text{write}(b_1, j, s) \\
& c_1 = \text{write}(a, i, t) \\
& c_2 = \text{write}(c_1, j, t) \\
& s = \text{read}(b_1, k) \\
& u = (l = j \ ? \ s : \text{read}(b_1, l)) \\
& v = (l = j \ ? \ t : (l = i \ ? \ t : \text{read}(a, l))) \\
& u \neq v
\end{aligned}
$$

# Aliasing Example Continued 6

after eliminating $c_2$, $c_1$, $b_2$, $b_1$

$$
\begin{aligned}
&i \neq k \\
&t = \text{read}(a, k) \\
&b_1 = \text{write}(a, i, t) \\
&b_2 = \text{write}(b_1, j, s) \\
&c_1 = \text{write}(a, i, t) \\
&c_2 = \text{write}(c_1, j, t) \\
&s = (k = i \ ? \ t : \text{read}(a, k)) \\
&u = (l = j \ ? \ s : (l = i \ ? \ t : \text{read}(a, l))) \\
&v = (l = j \ ? \ t : (l = i \ ? \ t : \text{read}(a, l))) \\
&u \neq v
\end{aligned}
$$

# Aliasing Example Continued 7

result after "write" elimination

$$
\begin{aligned}
& i \neq k \\
& t = \text{read}(a, k) \\
& s = (k = i \ ? \ t : \text{read}(a, k)) \\
& u = (l = j \ ? \ s : (l = i \ ? \ t : \text{read}(a, l))) \\
& v = (l = j \ ? \ t : (l = i \ ? \ t : \text{read}(a, l))) \\
& u \neq v
\end{aligned}
$$

# Aliasing Example Continued 8

after eliminating conditionals (if-then-else)

$$
\begin{aligned}
& i \neq k \\
& t = \text{read}(a, k) \\
& k = i \;\rightarrow\; s = t \\
& k \neq i \;\rightarrow\; s = \text{read}(a, k) \\
& l = j \;\rightarrow\; u = s \\
& l \neq j \wedge l = i \;\rightarrow\; u = t \\
& l \neq j \wedge l \neq i \;\rightarrow\; u = \text{read}(a, l) \\
& l = j \;\rightarrow\; v = t \\
& l \neq j \wedge l = i \;\rightarrow\; v = t \\
& l \neq j \wedge l \neq i \;\rightarrow\; v = \text{read}(a, l) \\
& u \neq v
\end{aligned}
$$

now treat “read” as uninterpreted function (say $f$)

check with lemmas-on-demand and congruence closure

# Ackermann's Reduction

formula in theory of uninterpreted functions with equality and disequality:

1. flatten terms by introducing new variables as before
   - remove nested function applications
   - equalities and disequalities have at least one variable on left or right side
2. instantiate congruence axiom in all possible ways:
   - replace all function applications $f(u)$ by new variable $f^u$
   - replace all function applications $f(u, v)$ by new variable $f^{u,v}$ etc.
3. if formula contains $f^u$ and $f^v$ add $u = v \rightarrow f^u = f^v$ as lemma etc.
4. use decision procedure for theory of equality and disequality
   - if the resulting formula after the first two steps contains $n$ variables
   - then only need to consider domains with $n$ elements
   - or bit-vectors of length $\lceil \log_2 n \rceil$ bits
   - allows eager encoding into SAT

"eagerly" generates all instantiations of the congruence axioms as lemmas

# Example of Ackermann's Reduction

we start with an already flattened formula

$$x = f(y) \wedge y = f(x) \wedge x \neq y$$

after second step

$$x = f^y \wedge y = f^x \wedge x \neq y$$

after adding lemmas in second step

$$x = f^y \wedge y = f^x \wedge x \neq y \wedge (x = y \rightarrow f^x = f^y)$$

resulting formula has 4 variables thus needs bit-vectors of length 2

# Example of Ackermann's Reduction to Bit-Vectors

```
$ cat ack.smt2
(set-logic QF_BV)
(declare-fun x () (_ BitVec 2))
(declare-fun y () (_ BitVec 2))
(declare-fun fx () (_ BitVec 2))
(declare-fun fy () (_ BitVec 2))
(assert (and (= x fy) (= y fx) (distinct x y) (=> (= x y) (= fx fy))))
(check-sat)
(exit)
$ boolector ack.smt2 -m -d
sat
x 0
y 3
fx 3
fy 0
```

# Theory of Bit-Vectors

- allows “bit-precise” reasoning
  - caputures semantics of low-level languages like assembler, C, C++, . . .
  - Java / C# also use two-complement representations for `int`
  - modelling of hardware / circuits on the word-level (RTL)
  - important for security applications and precise test case generation
- many operations
  - logical operations, bit-wise operations (and, or)
  - equalities, inequalities, disequalities
  - shift, concatenation, slicing
  - addition, multiplication, division, modulo, . . .
- main approach is reduction to SAT through *bit-blasting*
  - reduction of bit-vector operations similar to circuit synthesis
  - Ackermann’s Reduction only needs equality and disequality

# Bit-Blasting Bit-Vector Equality

for each bit-vector equality $u = v$ with $u$ and $v$ bit-vectors of width $w$

introduce new propositional variables for individual bits

$$u_1, \ldots, u_w \qquad v_1, \ldots, v_w$$

replace $u = v$ by new propositional variable $e_{u=v}$

add the propositional constraint

$$e_{u=v} \leftrightarrow \bigwedge_{i=1}^{w} (u_i \leftrightarrow v_i)$$

disequality $u \neq v$ is replaced by $\neg e_{u=v}$

resulting formula *satisfiable* iff original formula *satisfiable*

# Bit-Blasting Ackermann Example

$$x = f^y \wedge y = f^x \wedge x \neq y \wedge (x = y \rightarrow f^x = f^y)$$

now replacing the bit-vector equalities and the disequality by new *e* variables

$$e_{x=f^y} \wedge e_{y=f^x} \wedge \neg e_{x=y} \wedge (e_{x=y} \rightarrow e_{f^x=f^y})$$

and adding the equality constraints

$$\begin{array}{lcl}
e_{x=f^y} & \leftrightarrow & (x_1 \leftrightarrow f^y_1) \wedge (x_2 \leftrightarrow f^y_2) \\
e_{y=f^x} & \leftrightarrow & (y_1 \leftrightarrow f^x_1) \wedge (y_2 \leftrightarrow f^x_2) \\
e_{x=y} & \leftrightarrow & (x_1 \leftrightarrow y_1) \wedge (x_2 \leftrightarrow y_2) \\
e_{f^x=f^y} & \leftrightarrow & (f^x_1 \leftrightarrow f^y_1) \wedge (f^x_2 \leftrightarrow f^y_2)
\end{array}$$

gives an “equi-satisfiable” formula which can be checked by SAT solver

# Bit-Blasting Ackermann Example in Limboole Syntax

```
$ cat ackbitblasted.limboole
exfy & eyfx & !exy & (exy -> efxfy) &
(exfy <-> (x1 <-> fy1) & (x2 <-> fy2)) &
(eyfx <-> (y1 <-> fx1) & (y2 <-> fx2)) &
(exy <-> (x1 <-> y1) & (x2 <-> y2)) &
(efxfy <-> (fx1 <-> fy1) & (fx2 <-> fy2))
$ limboole ackbitblasted.limboole -s|grep -v SAT|sort
efxfy = 0
exfy = 1
exy = 0
eyfx = 1
fx1 = 0
fx2 = 1
fy1 = 1
fy2 = 1
x1 = 1
x2 = 1
y1 = 0
y2 = 1
```