SMT DETAILS
WS 2019 / 2020 (342.208)

Armin Biere armin.biere@jku.at
Martina Seidl martina.seidl@jku.at

Institute for Formal Models and Verification
Johannes Kepler Universität Linz

Version 2019.2
Propositional Skeleton

Example (arbitrary LRA formula)

\[ x \neq y \wedge (2 \times x \leq z \vee \neg(x - y \geq z \wedge z \leq y)) \]

eliminate \( \neq \) by disjunction

\[ (x < y \vee x > y) \wedge (2 \times x \leq z \vee \neg(x - y \geq z \wedge z \leq y)) \]

which is abstracted to a propositional formula called “propositional skeleton”

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

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(φ)

ψ = PropositionalSkeleton(φ)

let α be the abstraction function, mapping theory literals to prop. literals

while ψ has satisfiable assignment σ

let l₁, . . . , ln be all the theory literals with σ(α(li)) = 1

check conjunction L = l₁ ∧ · · · ∧ ln with theory solver

if theory solver returns satisfying assignment ρ return satisfiable

determine “small” sub-set {k₁, . . . , km} ⊆ {l₁, . . . , ln} where

K = k₁ ∧ · · · ∧ km remains unsatisfiable (by theory solver)

add lemma ¬K to ψ, actually replace ψ by ψ ∧ α(¬K)

return unsatisfiable

note that these lemmas ¬K are all clauses
Minimal Unsatisfiable Set (MUS)

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

\[
\text{MUS} = (a \lor \neg b) \land (a \lor b) \land (\neg a \lor \neg c) \land (\neg a \lor c) \land (a \lor \neg c) \land (a \lor c)
\]

■ 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 \( A \subset B \) 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

\[
\text{IterativeDestructiveMUS}(S) \\
M = S \\
D = S \\
\text{while } D \neq \emptyset \\
\quad \text{pick constraint } C \in D \\
\quad \text{if } M \setminus \{C\} \text{ unsatisfiable remove } C \text{ from } M \\
\quad \text{remove } C \text{ from } D \\
\text{return } M
\]

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)

\[
\text{QuickMUSRecursive}(D)
\]

if \(M \setminus D\) is satisfiable

if \(|D| > 1\)

let \(D = L \cup R\) with \(|L|, |R| > 0\) \(\ldots \geq \left\lfloor \frac{|D|}{2} \right\rfloor\)

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

\[
\forall a, i, j : i = j \rightarrow \text{read}(a, i) = \text{read}(a, j) \quad \text{array congruence}
\]

\[
\forall a, v, i, j : i = j \rightarrow \text{read}(\text{write}(a, i, v), j) = v \quad \text{read over write 1}
\]

\[
\forall a, v, i, j : i \neq j \rightarrow \text{read}(\text{write}(a, i, v), j) = \text{read}(a, j) \quad \text{read over write 2}
\]

- used to model memory (HW and SW)

- eagerly reduce arrays to uninterpreted functions by eliminating “write”

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

- more sophisticated non-eager algorithms are usually faster

  - such as for instance the “lemmas-on-demand” algorithm in Boolector
Simple Array Example

\[ i \neq j \land u = \text{read}(\text{write}(a, i, v), j) \land v = \text{read}(a, j) \land u \neq v \]

eliminate “write”

\[ i \neq j \land u = (i = j \ ? \ v : \text{read}(a, j)) \land v = \text{read}(a, j) \land u \neq v \]

simplify conditional by assuming “\(i \neq j\)”

\[ i \neq j \land u = \text{read}(a, j) \land v = \text{read}(a, j) \land u \neq v \]

applying congruence for both “read”

\[ i \neq j \land u = \text{read}(a, j) = \text{read}(a, j) = v \land u \neq v \]

which is clearly unsatisfiable
**More Complex Array Example for Checking Aliasing**

<table>
<thead>
<tr>
<th>Original</th>
<th>Optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>assert (i != k);</code></td>
<td><code>int t = a[k];</code></td>
</tr>
<tr>
<td><code>a[i] = a[k];</code></td>
<td><code>a[i] = t;</code></td>
</tr>
<tr>
<td><code>a[j] = a[k];</code></td>
<td><code>a[j] = t;</code></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>( i \neq k )</td>
<td>( t = \text{read}(a, k) )</td>
</tr>
<tr>
<td>( b_1 = \text{write}(a, i, t) )</td>
<td>( c_1 = \text{write}(a, i, t) )</td>
</tr>
<tr>
<td>( b_2 = \text{write}(b_1, j, s) )</td>
<td>( c_2 = \text{write}(c_1, j, t) )</td>
</tr>
<tr>
<td>( s = \text{read}(b_1, k) )</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>( \text{original} \neq \text{optimized} )</td>
<td>( b_2 \neq c_2 )</td>
</tr>
<tr>
<td></td>
<td>( \exists l \text{ with } \text{read}(b_2, l) \neq \text{read}(c_2, l) )</td>
</tr>
</tbody>
</table>
Thus \( \text{original} \neq \text{optimized} \) iff

\[
\begin{align*}
i &\neq k \\
t &\equiv \text{read}(a, k) \\
b_1 &\equiv \text{write}(a, i, t) \\
b_2 &\equiv \text{write}(b_1, j, s) \\
c_1 &\equiv \text{write}(a, i, t) \\
c_2 &\equiv \text{write}(c_1, j, t) \\
s &\equiv \text{read}(b_1, k) \\
\text{read}(b_2, l) &\neq \text{read}(c_2, l)
\end{align*}
\]

satisfiable
Aliasing Example Continued 2

thus original $\neq$ optimized iff

\[
i \neq k
\]
\[
t = \text{read}(a, k)
\]
\[
b_1 = \text{write}(a, i, t)
\]
\[
b_2 = \text{write}(b_1, j, s)
\]
\[
c_1 = \text{write}(a, i, t)
\]
\[
c_2 = \text{write}(c_1, j, t)
\]
\[
s = \text{read}(b_1, k)
\]
\[
u = \text{read}(b_2, l)
\]
\[
v = \text{read}(c_2, l)
\]
\[
u \neq v
\]

satisfiable
after eliminating $c_2$

\[
\begin{align*}
i & \neq k \\
t & = \text{read}(a, k) \\
b_1 & = \text{write}(a, i, t) \\
b_2 & = \text{write}(b_1, j, s) \\
c_1 & = \text{write}(a, i, t) \\
c_2 & = \text{write}(c_1, j, t) \\
s & = \text{read}(b_1, k) \\
u & = \text{read}(b_2, l) \\
v & = (l = j \ ? \ t : \text{read}(c_1, l)) \\
u & \neq v
\end{align*}
\]
Aliasing Example Continued 4

after eliminating $c_2, c_1$

\[ i \neq k \]
\[ t = \text{read}(a, k) \]
\[ b_1 = \text{write}(a, i, t) \]
\[ b_2 = \text{write}(b_1, j, s) \]
\[ c_1 = \text{write}(a, i, t) \]
\[ c_2 = \text{write}(c_1, j, t) \]
\[ s = \text{read}(b_1, k) \]
\[ u = \text{read}(b_2, l) \]
\[ v = (l = j \ ? \ t : (l = i \ ? \ t : \text{read}(a, l))) \]
\[ u \neq v \]
Aliasing Example Continued 5

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

\[\begin{align*}
i &\neq k \\
t &\equiv \text{read}(a,k) \\
b_1 &\equiv \text{write}(a,i,t) \\
b_2 &\equiv \text{write}(b_1,j,s) \\
c_1 &\equiv \text{write}(a,i,t) \\
c_2 &\equiv \text{write}(c_1,j,t) \\
s &\equiv \text{read}(b_1,k) \\
u &\equiv (l = j \ ? \ s : \text{read}(b_1,l)) \\
v &\equiv (l = j \ ? \ t : (l = i \ ? \ t : \text{read}(a,l))) \\
u &\neq v
\end{align*}\]
after eliminating $c_2, c_1, b_2, b_1$

\[
i \neq k \\
t = \text{read}(a, k) \\
b_1 = \text{write}(a, i, t) \\
b_2 = \text{write}(b_1, j, s) \\
c_1 = \text{write}(a, i, t) \\
c_2 = \text{write}(c_1, j, t) \\
s = (k = i \ ? t : \text{read}(a, k)) \\
u = (l = j \ ? s : (l = i \ ? t : \text{read}(a, l))) \\
v = (l = j \ ? t : (l = i \ ? t : \text{read}(a, l))) \\
u \neq v\]
result after “write” elimination

\[ i \neq k \]
\[ t = \text{read}(a, k) \]
\[ s = (k = i \ ? \ t : \text{read}(a, k)) \]
\[ u = (l = j \ ? \ s : (l = i \ ? \ t : \text{read}(a, l))) \]
\[ v = (l = j \ ? \ t : (l = i \ ? \ t : \text{read}(a, l))) \]
\[ u \neq v \]
Aliasing Example Continued 8

after eliminating conditionals (if-then-else)

\[ i \neq k \]
\[ t = \text{read}(a, k) \]
\[ k = i \rightarrow s = t \]
\[ k \neq i \rightarrow s = \text{read}(a, k) \]
\[ l = j \rightarrow u = s \]
\[ l \neq j \land l = i \rightarrow u = t \]
\[ l \neq j \land l \neq i \rightarrow u = \text{read}(a, l) \]
\[ l = j \rightarrow v = t \]
\[ l \neq j \land l = i \rightarrow v = t \]
\[ l \neq j \land l \neq i \rightarrow v = \text{read}(a, l) \]
\[ u \neq v \]

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

\[ x = f(y) \land y = f(x) \land x \neq y \]

after second step

\[ x = f^y \land y = f^x \land x \neq y \]

after adding lemmas in third step

\[ x = f^y \land y = f^x \land x \neq y \land (x = y \rightarrow f^x = f^y) \]

resulting formula has 4 variables thus needs bit-vectors of length 2
Example of Ackermann’s Reduction to Bit-Vectors

\[
\begin{align*}
\text{\$ cat ack.smt2} \\
(\text{set-logic QF\_BV}) \\
(\text{declare-fun x () (\_ BitVec 2))} \\
(\text{declare-fun y () (\_ BitVec 2))} \\
(\text{declare-fun fx () (\_ BitVec 2))} \\
(\text{declare-fun fy () (\_ BitVec 2))} \\
(\text{assert (and (= x fy) (= y fx) (distinct x y) (=> (= x y) (= fx fy))))} \\
(\text{check-sat}) \\
(\text{exit}) \\
\text{\$ boollector ack.smt2 -m -d} \\
\text{sat} \\
x 0 \\
y 3 \\
fx 3 \\
fy 0
\end{align*}
\]
Theory of Bit-Vectors

- allows “bit-precise” reasoning
  - captures 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

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

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

\[ x = f^y \land y = f^x \land x \neq y \land (x = y \rightarrow f^x = f^y) \]

now replacing the bit-vector equalities and the disequality by new \(e\) variables

\[ e_{x=y} \land e_{y=f^x} \land \neg e_{x=y} \land (e_{x=y} \rightarrow e_{f^x=f^y}) \]

and adding the equality constraints

\[
\begin{align*}
e_{x=f^y} & \iff (x_1 \iff f_1^y) \land (x_2 \iff f_2^y) \\
e_{y=f^x} & \iff (y_1 \iff f_1^x) \land (y_2 \iff f_2^x) \\
e_{x=y} & \iff (x_1 \iff y_1) \land (x_2 \iff y_2) \\
e_{f^x=f^y} & \iff (f_1^x \iff f_1^y) \land (f_2^x \iff f_2^y)
\end{align*}
\]

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