Lecture 14

Satisfiability Modulo Theories

1. Motivation: Path Sensitivity Analysis
2. A Basic SMT Solver
3. Optimizing the SMT Solver

Thanks to Clark Barrett, Nikolaj Bjørner Leonardo de Moura, Bruno Dutertre, Albert Oliveras, and Cesare Tinelli for contributing material used in this lecture.
What is Satisfiability Modulo Theories (SMT)?

• Satisfiability
  – the problem of determining whether a formula has a model (an assignment that makes the formula true)

• SAT: Satisfiability of **propositional formulas**
  – A model is a truth assignment to Boolean variables
  – SAT solvers: check satisfiability of propositional formulas
    • Decidable, NP-complete

• SMT: Satisfiability modulo theories
  – Satisfiability of first-order formulas containing operations from background theories such as arithmetic, arrays, uninterpreted functions, etc.
    E.g. \( g(a) = c \land f(g(a)) \neq f(c) \lor g(a) = d \land c \neq d \)
  – SMT Solvers:
    • check satisfiability of SMT formulas with respect to a theory
Use of SMT for Program Correctness & Test Generation

• Precision: Path sensitivity

• Given an assertion $A$, can we generate an input that triggers an error on a given path $p$?
  – Let $F$ be the formula representing the execution of $p$
  – Is the formula $F \land \neg A$ satisfiable?
    • Not satisfiable? No error on that path
    • Satisfiable? Find 1 assignment that satisfies the formula
      (1 set of test inputs)
Each Statement is a Logical Clause

Program  Assume data array bound is [0, N-1]
1 void ReadBlocks(int data[], int cookie)
2 {
3   int i = 0;
4   while (true)
5     {
6       int next;
7       next = data[i];
8       if (!(i < next && next < N)) return;
9       i = i + 1;
10      for (; i < next; i = i + 1){
11         if (data[i] == cookie)
12           i = i + 1;
13         else
14           Process(data[i]);
15     }
16 }  
17 }

One execution path
Static Single Assignment (SSA)

3 \[i_1 = 0;\]
7 \[next_1 = data_0[i_1];\]
8 \[i_1 < next_1 \&\& next_1 < N\]
9 \[i_2 = i_1 + 1;\]
10 \[i_2 < next_1;\]
11 \[data_0[i_2] = cookie_0;\]
12 \[i_3 = i_2 + 1;\]
10 \[i_4 = i_3 + 1;\]
10 \![i_4 < next_1];\]
7 \[next_2 = data_0[i_4];\]
An Execution Path as a Logic Formula

Program: Assume data array bound is [0, N-1]

```c
1 void ReadBlocks(int data[], int cookie)
2 {
3   int i = 0;
4   while (true)
5     {
6       int next;
7       next = data[i];
8       if (!((i < next && next < N)) return;
9       i = i + 1;
10      for (; i < next; i = i + 1){
11        if (data[i] == cookie)
12          i = i + 1;
13        else
14          Process(data[i]);
15      }
16   }
17 }
```

One execution path (SSA)

```plaintext
F = { 3 i_1 = 0;

3 i_1 = 0;
7 next_1 = data_0 [i_1];
8 i_1 < next_1 && next_1 < N_0
9 i_2 = i_1 + 1;
10 i_2 < next_1;
11 data_0 [i_2] = cookie_0;
12 i_3 = i_2 + 1;
13 !i_3 < next_1;
7 next_2 = data_0 [i_4];
10 i_4 = i_3 + 1;
10 !(i_4 < next_1);
```

Line 7: Array bound assertion A:

\[(0 \leq i_1 \land i_1 < N_0)\]
Checking for Out-of-Bound Array Access (Line 7, iteration 1)

Program: Assume data array bound is [0, N-1]

```c
void ReadBlocks(int data[], int cookie)
{
    int i = 0;
    while (true)
    {
        int next;
        next = data[i];
        if (!(i < next && next < N)) return;
        i = i + 1;
        for (; i < next; i = i + 1) {
            if (data[i] == cookie)
                i = i + 1;
            else
                Process(data[i]);
        }
    }
}
```

One execution path (SSA)

\[
F = \begin{cases}
3 \ i_1 = 0;
\end{cases}
\]

- \(0 \leq i_1 \land i_1 < N_0\)
- \(i_1 = 0 \land \neg(0 \leq i_1 \land i_1 < N_0)\)

Line 7: Array bound assertion \(A\):

Check: Is \(F \land \neg A\) satisfiable?
Program Assume data array bound is [0, N-1]

```c
1 void ReadBlocks(int data[], int cookie) {
2     int i = 0;
3     while (true) {
4         i = i + 1;
5         for (; i < next; i = i + 1){
6             if (data[i] == cookie)
7                 i = i + 1;
8             else
9                 Process(data[i]);
10         }
11     }
12 }
```

One execution path (SSA)

```
F = {
    3 i_1 = 0;
    7 next_1 = data_0 [i_1];
    8 i_1 < next_1 && next_1 < N_0
    9 i_2 = i_1 + 1;
    10 i_2 < next_1;
    11 data_0 [i_2] = cookie_0;
    12 i_3 = i_2 + 1;
    10 !(i_4 < next_1);
    7 next_2 = data_0 [i_4];
}
```

Line 7: Array bound assertion \( A \):

\( 0 \leq i_1 \land i_1 < N_0 \)

\( i_1 = 0 \land \neg (0 \leq i_1 \land i_1 < N_0) \)

Check: Is \( F \land \neg A \) satisfiable?

Yes! \{ i_1\rightarrow 0, \ N_0\rightarrow 0 \} BUG!!
Checking for Out-of-Bound Array Access (Line 7, iteration 2)

Program Assume data array bound is [0, N-1]

```c
1 void ReadBlocks(int data[], int cookie)
2 {
3     int i = 0;
4     while (true)
5         {
6             int next;
7             next = data[i];
8             if (!((i < next && next < N)) return;
9             i = i + 1;
10            for (; i < next; i = i + 1){
11                if (data[i] == cookie)
12                    i = i + 1;
13                else
14                    Process(data[i]);
15            }
16        }
17    }
```

One execution path (SSA)

```
3 i_1 = 0;
7 next_1 = data_0 [i_1];
8 i_1 < next_1 && next_1 < N_0
9 i_2 = i_1 + 1;
10 i_2 < next_1;
11 data_0 [i_2] = cookie_0;
12 i_3 = i_2 + 1;
10 i_4 = i_3 + 1;
10 !(i_4 < next_1);
7 next_2 = data_0 [i_4];
```

Line 7: Array bound assertion A:

\[(0 \leq i_4 \land i_4 < N_0)\]

Check: Is \(F \land \neg A\) satisfiable?

\[F \land \neg (0 \leq i_4 \land i_4 < N_0)\]
**Answer for Out-of-Bound Array Access (Line 7, iteration 2)**

**Program** Assume data array bound is \([0, N-1]\)

```c
1 void ReadBlocks(int data[], int cookie)
2 {
3   int i = 0;
4   while (true)
5   {
6     int next;
7     next = data[i];
8     if (!((i < next && next < N)) return;
9     i = i + 1;
10    for (; i < next; i = i + 1){
11      if (data[i] == cookie)
12        i = i + 1;
13      else
14        Process(data[i]);
15    }
16  }
17 }
```

**One execution path (SSA)**

```
3  i_1 = 0;
7  next_1 = data_0 [i_1];
8  i_1 < next_1 && next_1 < N_0
9  i_2 = i_1 + 1;
10 i_2 < next_1;
11 data_0 [i_2] = cookie_0;
12 i_3 = i_2 + 1;
10 !(i_4 < next_1);
7  next_2 = data_0 [i_4];
```

**Line 7: Array bound assertion A:**

\[(0 \leq i_4 \land i_4 < N_0)\]
Checking the Whole Program All at Once

- A program has many execution paths
- Conditional statements
  - Represent alternative paths symbolically with one formula using SSA
- Loops
  - Optimistically: Unroll a few times
  - Catches many errors, but not all errors
Conditional Statements

• Conditional statements: $\varphi$ functions in SSA

```plaintext
1 if (i > 0) {
  2   a = 2;
  3   b = 3;
 4} else
 5{
  6   a = 3;
  7   b = 2;
 8}
 9   c = a+b;
```

• Assert $A$: $c_3 = 5$

• Is $F \land \neg A$ satisfiable?

\[
\varphi_1 = (i_0 > 0) \land (\varphi_1 \rightarrow c_3 = 5) \land (\neg \varphi_1 \rightarrow c_3 = 5) \land (c_3 \neq 5)
\]
Applying the Resolution Rule to Example

• A resolution rule in propositional logic:

\[
\begin{array}{c}
\text{Given } p \lor A \text{ and } \neg p \lor B, \text{ add the resolvent } A \lor B
\\
\text{Resolve}
\\
\begin{array}{c}
p \lor A \\
\neg p \lor B
\\
\hline
A \lor B
\end{array}
\end{array}
\]

• Is \( F \land \neg A \) satisfiable?

\[
\varphi_1 = (i_0 > 0) \land (\varphi_1 \rightarrow c_3 = 5) \land (\neg \varphi_1 \rightarrow c_3 = 5) \land (c_3 \neq 5)
\]

• Recall: \( p \rightarrow q \equiv \neg p \lor q \)

\[
\begin{align*}
\varphi_1 & = (i_0 > 0) \land (\neg \varphi_1 \lor c_3 = 5) \land (\varphi_1 \lor c_3 = 5) \land (c_3 \neq 5) \\
\varphi_1 & = (i_0 > 0) \land (c_3 = 5) \land (c_3 \neq 5)
\end{align*}
\]

• \( F \land \neg A \) is not satisfiable
• The assertion \( A \) is true.
Loops

- Optimistically: Unroll two times

```plaintext
for (; i < next; i = i + 1){
    if (data[i] == cookie)
        i = i + 1;
    else
        Process(data[i]);
}
```

```plaintext
if (i < next) {
    if (data[i] == cookie)
        i = i + 1;
    else
        Process(data[i]);
    i = i + 1;
    if (i < next) {
        if (data[i] == cookie)
            i = i + 1;
        else
            Process(data[i]);
        i = i + 1;
    }
}
```
Loops: Apply SSA

1 if (i < next) {
2   if (data[i] == cookie)  
3     i = i + 1;
4   else
5       Process(data[i]);
6
7   i = i + 1;
8
9 if (i < next) {
10  if (data[i] == cookie) 
11     i = i + 1;
12  else
13     Process(data[i]);
14
15  i = i + 1;
16  }
17 }

1 φ₁ = (i₀ < next₀);
2 φ₂ = (data₀[i₀] == cookie₀);
3 i₁ = i₀ + 1;
4
5
6 i₂ = φ₂ ? i₁ : i₀;
7 i₃ = i₂ + 1;
8
9 φ₃ = (i₃ < next₀);
10 φ₄ = (data₀[i₃] == cookie₀);
11 i₄ = i₃ + 1;
12
13
14 i₅ = φ₄ ? i₄ : i₃;
15 i₆ = i₅ + 1;
16 i₇ = φ₃ ? i₆ : i₃;
17 i₈ = φ₁ ? i₇ : i₀;
## Major Categories of Program Analysis Tools

<table>
<thead>
<tr>
<th>Complete (Small programs)</th>
<th>Static Property Based</th>
<th>Dynamic Execution Based</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Verification</strong></td>
<td>Prove a property in a program</td>
<td>(Symbolic) Model Checking (SMT/BDD)</td>
</tr>
<tr>
<td><strong>Floyd-Hoare logic:</strong></td>
<td>{pre-condition} s {post-condition}</td>
<td>Given a system model (sw/hw), simulate the execution to check if a property is true for all possible inputs. Symbolic: many states all at once</td>
</tr>
<tr>
<td>Applicable to small programs</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Incomplete (Large programs)</th>
<th>Static Analysis (Data flow)</th>
<th>Test case generation (SMT/BDD)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Abstract the program conservatively</strong></td>
<td><strong>Sound: no false-negatives--find all bugs</strong></td>
<td><strong>Check a property opportunistically (e.g. unroll loops twice)</strong></td>
</tr>
<tr>
<td><strong>Check a property</strong></td>
<td><strong>False-positives: false warnings</strong></td>
<td><strong>Use analysis to generate test inputs</strong></td>
</tr>
<tr>
<td><strong>Too imprecise is useless</strong></td>
<td><strong>No false-positives: generate a test</strong></td>
<td><strong>No correctness/security guarantees</strong></td>
</tr>
</tbody>
</table>
2. A Basic SMT Solver

- SMT: Satisfiability modulo theories
  - Satisfiability of first-order formulas containing operations from background theories such as arithmetic, arrays, uninterpreted functions, etc.

- SMT Solvers:
  - check satisfiability of SMT formulas with respect to a theory
Uninterpreted function theory:
Functions assumed to be pure:
A function always returns the same value for a given input

Example: $x \geq 0 \land f(x) \geq 0 \land f(y) \geq 0 \land x \neq y$

Is this formula satisfiable?
SMT with Linear Inequalities & Function Theories

Uninterpreted function theory:
Functions assumed to be pure:
A function always returns the same value for a given input

Example: $x \geq 0 \land f(x) \geq 0 \land f(y) \geq 0 \land x \neq y$
This formula is satisfiable
An example model satisfying the formula
$$
\begin{align*}
x & \mapsto 0 \\
y & \mapsto 1 \\
f(0) & \mapsto 0 \\
f(1) & \mapsto 0
\end{align*}
$$
Adding a Theory of Arrays

Notation: write\((v, i, x)\) means \(v[i] := x;\)  
\hspace{1cm} read\((v, i)\) means returns \(v[i]\)

Array theory axioms:
\hspace{1cm} read\(\)\((\)write\((v, i, x)\), i\)\) \(= x\)
\hspace{1cm} \(i \neq j\) \(\rightarrow\) read\(\)\((\)write\((v, i, x)\), j\)\) \(=\) read\(\)\((v, j)\)

\hspace{1cm} \(v \neq w\) \(\rightarrow\) read\(\)\((v, k)\) \(\neq\) read\(\)\((w, k)\) for some \(k\)

Example: \(b + 2 = c \land f (\text{read}(\text{write}(a, b, 3), c - 2)) \neq f (c - b + 1)\)

Is this formula satisfiable?
Adding a Theory of Arrays

Notation: write(v, i, x) means \(v[i] := x\);
read(v, i) means returns \(v[i]\)

Array theory axioms:
- \(\text{read(\text{write(v, i, x)}, i)} = x\)
- \(i \neq j \rightarrow \text{read(\text{write(v, i, x)}, j)} = \text{read(v, j)}\)
- \(v \neq w \rightarrow \text{read(v, k)} \neq \text{read(w, k)}\) for some \(k\)

Example: \(b + 2 = c \land f(\text{read(\text{write(a, b, 3)}, c - 2)}) \neq f(c - b + 1)\)

By arithmetic reasoning, this is equivalent to
\(b + 2 = c \land f(\text{read(\text{write(a, b, 3)}, b)}) \neq f(3)\)

By first array axiom, \(b + 2 = c \land f(3) \neq f(3)\)

But in the theory of uninterpreted functions, \(f(3) \neq f(3)\) is not true

Therefore, this formula is not satisfiable
SMT Solvers

• Input: a first-order formula F
• Output
  – F is satisfiable, optionally: a model M
  – F is unsatisfiable, optionally: a proof of unsatisfiability
• Which is easier?
• Main issues
  – formula size (e.g., thousands of atoms or more)
  – formulas with complex Boolean structure
  – combination of theories
Overview of a SMT Solver

• SMT Solver = SAT Solver + Theory Solver
  – Given a formula F, the SAT solver enumerates possible truth assignments (M)
  – The theory solver checks whether the truth assignments are satisfiable in the theories
Example of a Basic Algorithm

\[ F: \quad g(a) = c \land f(g(a)) \neq f(c) \lor g(a) = d \land c \neq d \]

SAT SOLVER

1. choose a model M
2. choose a model M
3. unsat

THEORY SOLVER
(Uninterpreted Functions)

send F
unsat

\[ 1, \overline{2} \lor 3, 4 \]

send F
unsat

\[ 1, 3, \overline{4} \]

send F

\[ 1, \overline{2} \lor 3, 4, \overline{1} \lor 2 \lor 4 \]

\[ 1, \overline{2} \lor 3, 4, \overline{1} \lor 2 \lor 4, \overline{1} \lor 3 \lor 4 \]
Basic Algorithm

• DEFINITION
  T-conflict: check for conflicts with respect to theory T

• DESIGN
  Two independent solvers:
    • SAT solver that is independent of theory
    • Theory solver that checks for T-conflicts clause by clause

• ALGORITHM
  Repeat
  SAT Solver: propose a full propositional model M for formula F
    if no M is found, F is unsatisfiable.
  Theory Solver:
    Check for T-conflict on model M
    If M is satisfiable: F is satisfiable
    If M has a T-conflict, add constraint to F
3. Improvements (Example, Algorithm, Rules)

A. Incremental model decision:

Don’t just guess the entire model (all the assignments)
Check each assignment incrementally, not all at once.
Two kinds of assignments
  Propagated: deduced from the Boolean expression
  Arbitrary decisions (marked with •)
If unsatisfiable, backtrack on decisions
(Propagate, Decide, T-Conflict, Learn, Restart)

B. Use the theory to propagate and learn (T-Propagate)

C. Backtrack to conflicting decision (Conflict, Explain, Backjump)
### A. Incremental: Example

**F:**

\[
g(a) = c \land f(g(a)) \neq f(c) \lor g(a) = d \land c \neq d
\]

<table>
<thead>
<tr>
<th>M</th>
<th>F</th>
<th>C</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td></td>
<td>Propagate+, OK</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>2</td>
<td>Decide</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>2</td>
<td>T-Conflict</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>2</td>
<td>Learn</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>2</td>
<td>Restart</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>2</td>
<td>Propagate+</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>2</td>
<td>T-Conflict, Learn</td>
</tr>
<tr>
<td>fail</td>
<td></td>
<td></td>
<td>Fail</td>
</tr>
</tbody>
</table>
A. Incremental: Algorithm

- Build incrementally a satisfying truth assignment $M$ for a CNF formula $F$
  - CNF: conjunction of disjunctions of literals

- Algorithm
  Apply rules until there is a satisfying model or Fail, in decreasing priority
  
  **T-conflict**: if all the literals $l_1, \ldots, l_n$ in $M$ cannot be satisfied by $T$, set the conflict clause $C := T_1 \lor \ldots \lor T_n$
  
  **Learn**: add the new conflict constraint to $F$
  **Restart**: Restart the SAT solver after learning a new constraint
  **Propagate**: deduce the truth value of a literal from $M$ and $F$
  **Decide**: guess a truth value
  **Fail**: if there is no decision to roll back
# A. Incremental: Rules

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
</table>
| **Propagate** | Deduce the truth value of a literal from M and F<br>\[
\frac{l_1 \lor \ldots \lor l_n \lor l \in F}{\text{T}_1, \ldots, \text{T}_n \in M \quad l, \text{T} \notin M} \quad M := M \cdot l
\] |
| **Decide** | Guess a truth value<br>\[
\frac{l \in \text{Lit}(F)}{l, \neg l \notin M} \quad M := M \cdot l
\] |
| **T-Conflict** | If all the literals \(l_1, \ldots, l_n\) in M cannot be satisfied by T, set the conflict clause<br>\[
\frac{C = \text{no} \quad l_1, \ldots, l_n \in M \quad l_1, \ldots, l_n \models_T \bot}{C := \text{T}_1 \lor \ldots \lor \text{T}_n}
\] |
| **Learn** | Add the new learned constraint to formula F<br>\[
\frac{F \models_P C \quad C \notin F}{F := F \cup \{C\}}
\] |
| **Restart** | Restart the SAT solver<br>\[
M := M^{[0]} \quad C := \text{no}
\] <br>Each Decide defines a new level<br>\(M^{[i]}\) means Model M up to level \(i\)
A. Incremental: Rules

Fail if there is no decision to roll back

\[ l_1 \lor \ldots \lor l_n \in F \quad \overline{T_1}, \ldots, \overline{T_n} \in M \quad \bullet \notin M \]

\[ \text{fail} \]
Improvements (Example, Algorithm, Rules)

A. Incremental model decision
   (Propagate, Decide, T-Conflict, Learn, Restart)

B. Use the theory to propagate and learn (T-Propagate)
   In A, propagation is based only on the Boolean expression;
   Here, we add propagation due to the Theories

C. Backtrack to conflicting decision (Conflict, Explain, Backjump)
### B: T-Propagate: Example

\[ g(a) = c \land f(g(a)) \neq f(c) \lor g(a) = d \land c \neq d \]

<table>
<thead>
<tr>
<th>M</th>
<th>F</th>
<th>C</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1, ( \overline{2} \lor 3, \overline{4} )</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 ( \overline{4} )</td>
<td>1, ( \overline{2} \lor 3, \overline{4} )</td>
<td></td>
<td>Propagate+</td>
</tr>
<tr>
<td>1 ( \overline{4} ) ( 2 )</td>
<td>1, ( \overline{2} \lor 3, \overline{4} )</td>
<td></td>
<td>T-Propagate (1 ( \models_T ) 2)</td>
</tr>
<tr>
<td>1 ( \overline{4} ) ( 2 ) ( 3 )</td>
<td>1, ( \overline{2} \lor 3, \overline{4} )</td>
<td></td>
<td>T-Propagate (1, ( \overline{4} ) ( \models_T ) ( 3 ))</td>
</tr>
<tr>
<td>Fail</td>
<td>1, ( \overline{2} \lor 3, \overline{4} )</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Notation:**

1 \( \models_T \) 2: predicate 1 entails predicate 2 under theory \( T \)

If predicate 1 is true, predicate 2 is true under theory \( T \)
B. T-Propagate: Algorithm

- Add T-Propagate to increase deduced values using theory T

- Algorithm
  
  Apply rules until there is a satisfying model or Fail, in decreasing priority
  
  **T-conflict**: if all the literals $l_1, ..., l_n$ in $M$ cannot be satisfied by $T$, set the conflict clause $C := \overline{T_1} \lor ... \lor \overline{T_n}$
  
  **Learn**: add the new conflict constraint to $F$
  
  **Restart**: Restart the SAT server after learning a new constraint
  
  **Propagate**: deduce the truth value of a literal from $M$ and $F$
  
  **T-Propagate**: deduce the truth value of a literal using theory $T$

  **Decide**: guess a truth value

  **Fail**: if there is no decision to roll back
B. T-Propagate: Rules

Deduce the truth value of a literal using theory $T$

\[
\text{T-Propagate} \quad \frac{l \in \text{Lit}(F) \quad M \models_T l \quad \overline{l}, \overline{l} \notin M}{M := M \, l}
\]
Improvements (Example, Algorithm, Rules)

A. Incremental model decision
   (Propagate, Decide, T-Conflict, Learn, Restart)

B. Use the theory to propagate and learn (T-Propagate)

C. Backtrack to conflicting decision (Conflict, Explain, Backjump)
   Find the root cause that causes the conflict
   Backtrack by skipping decisions immaterial to the conflict
C. Backjumping: Example

\[ F := \{1, 1 \lor 2, 3 \lor 4, 3 \lor 6, 1 \lor 5 \lor 7, 2 \lor 5 \lor 6 \lor 7\} \]

<table>
<thead>
<tr>
<th>M</th>
<th>F</th>
<th>C</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>F</td>
<td></td>
<td>Propagate</td>
</tr>
<tr>
<td>12</td>
<td>F</td>
<td></td>
<td>Propagate</td>
</tr>
<tr>
<td>12 \cdot 3</td>
<td>F</td>
<td></td>
<td>Decide</td>
</tr>
<tr>
<td>12 \cdot 34</td>
<td>F</td>
<td></td>
<td>Propagate</td>
</tr>
<tr>
<td>12 \cdot 34 \cdot 5</td>
<td>F</td>
<td></td>
<td>Decide</td>
</tr>
<tr>
<td>12 \cdot 34 \cdot 5外交</td>
<td>F</td>
<td></td>
<td>Propagate</td>
</tr>
<tr>
<td>12 \cdot 34 \cdot 5外交外交</td>
<td>F</td>
<td></td>
<td>Propagate</td>
</tr>
<tr>
<td>12 \cdot 34 \cdot 5外交外交外交</td>
<td>F</td>
<td>(2 \lor 3 \lor 6 \lor 7)</td>
<td>Conflict</td>
</tr>
</tbody>
</table>
C. Backjumping: Example Details

\[ F := \{1, \overline{1} \lor 2, 3 \lor 4, 5 \lor 6, 1 \lor 3 \lor 5 \lor 7, 2 \lor 3 \lor 6 \lor 7\} \]
\[ M := 12 \cdot 34 \cdot \overline{567} \]
\[ C := \overline{2} \lor \overline{5} \lor 6 \lor \overline{7} \]

- Conflict: \( \overline{2} \lor 5 \lor 6 \lor \overline{7} \) last literal choice is 7
- Explain: Choice of 7 is due to \( 1 \lor 5 \lor 7 \)
- Learn: \( 1 \lor 2 \lor 5 \lor 6 = \) resolvent of \( \overline{2} \lor 5 \lor 6 \lor \overline{7} \) and \( 1 \lor 5 \lor 7 \)
- Conflict: \( 1 \lor 2 \lor 5 \lor 6 \) last literal choice is 6
- Explain: Choice of 6 is due to \( \overline{5} \lor \overline{6} \)
- Learn: \( 1 \lor 2 \lor 5 = \) resolvent of \( 1 \lor 2 \lor 5 \lor 6 \) and \( \overline{5} \lor \overline{6} \)
- Conflict: \( 1 \lor 2 \lor 5 \)
- Backjump: Choice of 5 was a decision
  - Conflict involves literals 1, 2, 5, the decision of 5 is at level 2
  - 1, 2 are both level 0
  - Back jump to level 0, propagate 1,2 and choose \( \overline{5} \)

Given \( p \lor A \) and \( \neg p \lor B \), add the resolvent \( A \lor B \)

\[
\begin{array}{c}
p \lor A \\
\neg p \lor B \\
\hline
A \lor B
\end{array}
\]

Resolve
### C. Backjumping: Example

\[ F := \{1, 1\lor 2, 3\lor 4, 3\lor 6, 1\lor 5\lor 7, 2\lor 5\lor 6\lor 7\} \]

<table>
<thead>
<tr>
<th>M</th>
<th>F</th>
<th>C</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( F )</td>
<td>( F )</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>( F )</td>
<td>( F )</td>
<td>Propagate</td>
</tr>
<tr>
<td>12</td>
<td>( F )</td>
<td>( F )</td>
<td>Propagate</td>
</tr>
<tr>
<td>12 \cdot 3</td>
<td>( F )</td>
<td>( F )</td>
<td>Decide</td>
</tr>
<tr>
<td>12 \cdot 34</td>
<td>( F )</td>
<td>( F )</td>
<td>Propagate</td>
</tr>
<tr>
<td>12 \cdot 34 \cdot 5</td>
<td>( F )</td>
<td>( F )</td>
<td>Decide</td>
</tr>
<tr>
<td>12 \cdot 34 \cdot 5\overline{6}</td>
<td>( F )</td>
<td>( F )</td>
<td>Propagate</td>
</tr>
<tr>
<td>12 \cdot 34 \cdot 5\overline{6} \overline{7}</td>
<td>( F )</td>
<td>( F )</td>
<td>Propagate</td>
</tr>
<tr>
<td>12 \cdot 34 \cdot 5\overline{6} \overline{7}</td>
<td>( F )</td>
<td>( F )</td>
<td>Propagate</td>
</tr>
<tr>
<td>12 \cdot 34 \cdot 5\overline{6} \overline{7}</td>
<td>( F )</td>
<td>( 2\lor 3\lor 6\lor 7 )</td>
<td>Conflict</td>
</tr>
<tr>
<td>12 \cdot 34 \cdot 5\overline{6} \overline{7}</td>
<td>( F )</td>
<td>( 1\lor 2\lor 5\lor 6 )</td>
<td>Explain with ( 1\lor 5\lor 7 )</td>
</tr>
<tr>
<td>12 \cdot 34 \cdot 5\overline{6} \overline{7}</td>
<td>( F )</td>
<td>( 1\lor 2\lor 5 )</td>
<td>Explain with ( 5\lor 6 )</td>
</tr>
<tr>
<td>125</td>
<td>( F )</td>
<td>( F )</td>
<td>Backjump</td>
</tr>
<tr>
<td>125 \cdot 3</td>
<td>( F )</td>
<td>( F )</td>
<td>Decide</td>
</tr>
<tr>
<td>125 \cdot 34</td>
<td>( F )</td>
<td>( F )</td>
<td>Propagate (SAT)</td>
</tr>
</tbody>
</table>
C. Backjumping: Algorithm

• If $M$ is $T$-unsatisfiable, backtrack to some point where the assignment was still $T$-satisfiable

• Trace back to the decision that causes the conflict $C$
  • Let $l$ be the last literal choice that causes conflict $C$, if $l$ is a decision, proceed to the next step
  • Explain:
    if $l$ is chosen due to clause $C_1$ in $F$ (explanation), new conflict $C = \text{resolvent of } C \text{ and } C_1$ (eliminating $l$)
  • Repeat the above

• Backtrack by skipping decisions immaterial to conflict $C$
  • Backjump: Keep model up to level $i$, (highest level of satisfiable decisions involved in $C$); add the latest literal $l$ in $C$
C. Backjumping Rules

**Conflict**

If one of the literals $T_1, \ldots, T_n$ in $M$ must be inverted in $F$, set the conflict clause $C := l_1 \lor \ldots \lor l_n$

- $C = \text{no}$
- $l_1 \lor \ldots \lor l_n \in F$
- $T_1, \ldots, T_n \in M$

**Explain**

Given conflict $C$ involving latest $l$, chosen due to a clause in $F$, their resolvent is the new conflict

- $C = l \lor D$
- $l_1 \lor \ldots \lor l_n \lor \overline{T} \in F$
- $T_1, \ldots, T_n <_M T$

**Backjump**

Keep model up to level $i$ (highest level of sat. decisions involved in $C$); add latest $l$ in $C$

- $C = l_1 \lor \ldots \lor l_n \lor l$
- $\text{lev } T_1, \ldots, \text{lev } T_n \leq i < \text{lev } \overline{l}$

- $C := \text{no} \quad M := M[i] l$

$l <_M l'$ if $l$ occurs before $l'$ in $M$

$M[i]$ means Model $M$ up to level $i$

$\text{lev } l = i$ iff $l$ occurs in decision level $i$ of $l$
C. Backjumping Rules (cont.)

Replace

<table>
<thead>
<tr>
<th>Fail</th>
<th>Fail if there is no decision to roll back</th>
</tr>
</thead>
<tbody>
<tr>
<td>$l_1 \lor \ldots \lor l_n \in F$</td>
<td>$T_1, \ldots, T_n \in M$ \quad &amp; $\notin M$</td>
</tr>
</tbody>
</table>


with

<table>
<thead>
<tr>
<th>Fail</th>
<th>Fail if there is a conflict and there is no decision to roll back</th>
</tr>
</thead>
<tbody>
<tr>
<td>$C \neq \text{no}$ \quad &amp; $\notin M$</td>
<td>fail</td>
</tr>
</tbody>
</table>
Putting it All Together

Apply rules until there is a satisfying model or Fail, in decreasing priority

**T-conflict**: if all the literals $l_1, \ldots, l_n$ in $M$ cannot be satisfied by $T$, set the conflict clause $C := \overline{l_1} \lor \ldots \lor \overline{l_n}$

**Explain**: If the last literal $l$ in conflict $C$ is not a decision, if $l$ chosen due to clause $C_1$ in $F$ (explanation), new conflict = resolvent of $C$ and $C_1$

**Backjump**: Keep model up to level $i$, (highest level of satisfiable decisions involved in $C$); add the latest literal $l$ in $C$

**Learn**: add the new conflict constraint to $F$

**Propagate**: deduce the truth value of a literal from $M$ and $F$

**T-Propagate**: deduce the truth value of a literal using theory $T$

**Decide**: guess a truth value

**Fail**: if there is no decision to roll back

**Restart**: Restart on the learned $F$ if too many conflicts have been found
Summary

• Use of SMT to handle path sensitivity in test generation & static analysis

• Basic optimizations in SMT Solver
  – Incremental model decision (Propagate, Decide, T-Conflict, Learn, Restart)
  – Use the theory to propagate and learn (T-Propagate)
  – Smart backtracking (Conflict, Explain, Backjump)

• More work is needed to handle combinations of theories etc

• Practical tools:
  – Z3 SMT solver
    • A widely used, open-source project from Microsoft
  – cvc5 SMT solver (developed at Stanford)
Applications of SMT

- Program Analysis and Verification!
- Security (e.g., checking access policies at AWS)
- Hardware configuration and verification
- Verification of smart contracts for blockchains
- Checking database integrity constraints
- Network configuration checking
- Program synthesis
- And many more...
Research in SMT

- New theories and theory solvers, often adapted to an application domain
- Independently checkable proofs for unsatisfiable formulas
- Extensions to reasoning about quantifiers
- Extensions to synthesis
- Better decision heuristics
- Better algorithms and performance
- Theory combination mechanisms
- Parallel SMT solving
Further Readings

• “Satisfiability Modulo Theories”
  Clark Barrett and Cesare Tinelli.
  In Handbook of Model Checking,
  (Ed Clarke, Thomas Henzinger, and Helmut Veith, eds.), 2016.
  In preparation.

• “Satisfiability Modulo Theories”
  Clark Barrett, Roberto Sebastiani, Sanjit Seshia, and Cesare Tinelli.
  In Handbook of Satisfiability,
  vol. 185 of Frontiers in Artificial Intelligence and Applications,
  (Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, eds.),

• Satisfiability Modulo Theories: Introduction and Applications
  Leonardo De Moura, Nikolaj Bjørner
  Communications of the ACM, Vol. 54 No. 9, Pages 69-77
  Sept 2011