Lecture 14
Satisfiability Modulo Theories

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

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 in a decidable first-order theory
User of SMT for Program Correctness & Test Generation

- Precision: Path sensitivity

- Given an assertion $A_i$, 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 input)

---

Each Statement is a Logical Clause

<table>
<thead>
<tr>
<th>Program</th>
<th>One execution path</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assume data array bound is $[0, N-1]$</td>
<td>Static Single Assignment (SSA)</td>
</tr>
<tr>
<td>1 void ReadBlocks(int data[], int cookie) {</td>
<td>3 $i_1 = 0;$</td>
</tr>
<tr>
<td>2</td>
<td>7 next, = data, [i, ];</td>
</tr>
<tr>
<td>3 int $i_0 = 0;$</td>
<td>8 $i_1 &lt; next, &amp;&amp; next, &lt; N_0$</td>
</tr>
<tr>
<td>4 while (true)</td>
<td>9 $i_1 = i_0 + 1;$</td>
</tr>
<tr>
<td>5</td>
<td>10 $i_1 &lt; next;$</td>
</tr>
<tr>
<td>6</td>
<td>11 data, [i, ] = cookie, ;</td>
</tr>
<tr>
<td>7 int next;</td>
<td>12 $i_2 = i_0 + 1;$</td>
</tr>
<tr>
<td>8 if (!($i_0 &lt; next &amp;&amp; next, &lt; N)) return;</td>
<td>10 !(i, &lt; next,);</td>
</tr>
<tr>
<td>9 $i_0 = i_0 + 1;$</td>
<td>11 $i_1 = i_1 + 1;$</td>
</tr>
<tr>
<td>10 for (; $i_0 &lt; next; i = i_0 + 1)$</td>
<td>7 next, = data, [i, ];</td>
</tr>
<tr>
<td>11 if (data,$[i, ] == cookie,)$</td>
<td>8 $i_1 &lt; next, &amp;&amp; next, &lt; N_0$</td>
</tr>
<tr>
<td>12 $i_0 = i_0 + 1;$</td>
<td>9 $i_1 = i_0 + 1;$</td>
</tr>
<tr>
<td>13 else</td>
<td>10 $i_1 = i_0 + 1;$</td>
</tr>
<tr>
<td>14 Process(data,$[i, ]$);</td>
<td>11 $i_1 = i_1 + 1;$</td>
</tr>
<tr>
<td>15 }</td>
<td>7 next, = data, [i, ];</td>
</tr>
<tr>
<td>16 }</td>
<td>10 !(i, &lt; next,);</td>
</tr>
<tr>
<td>17 }</td>
<td>11 $i_1 = i_1 + 1;$</td>
</tr>
</tbody>
</table>
An Execution Path as a Logic Formula

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 (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_1 < next_1;
11 data_0[i_2] = cookie_0;
12 i_3 = i_2 + 1;
10 i_1 = i_3 + 1;
10 !(i_4 < next_1);
7 next_2 = data_0[i_4];

Checking for Out-of-Bound Array Access 1

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 (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_1 < next_1;
11 data_0[i_2] = cookie_0;
12 i_3 = i_2 + 1;
10 i_1 = i_3 + 1;
10 !(i_4 < next_1);
7 next_2 = data_0[i_4];

Line 7: Array bound assertion A:

Check: Is F ^ ~A satisfiable?

\[ i_1 = 0 \land \neg(0 \leq i_1 \land i_1 < N_0) \]
Answer for Out-of-Bound Array Access 1

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

```c
void ReadBlocks(int data[], int cookie)
{
    int i = 0;
    while (true)
    {
        int 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)

- Line 7: Array bound assertion \(A:\)
  
  \[0 \leq i < N\]  

  \(\vdash\) maps to

  \(i_1 = 0\)

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

- \(F = \{ i_1 = 0, N_{i_1} = 0 \}\)
- \(\neg A = \{ 0 \leq i_1 \land i_1 < N_{i_1}\}\)
- Yes! \(\{ i_1 \mapsto 0, N_{i_1} \mapsto 0\}\)

Checking for Out-of-Bound Array Access 2

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

```c
void ReadBlocks(int data[], int cookie)
{
    int i = 0;
    while (true)
    {
        int 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)

- Line 7: Array bound assertion \(A:\)
  
  \[0 \leq i < N\]  

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

- \(F = \{ i_1 = 0, N_{i_1} = 0 \}\)
- \(\neg A = \{ 0 \leq i_1 \land i_1 < N_{i_1}\}\)

Yes! \(\{ i_1 \mapsto 0, N_{i_1} \mapsto 0\}\)
Checking for Out-of-Bound Array Access 2

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)

Line 7: Array bound assertion A:

<table>
<thead>
<tr>
<th>Var</th>
<th>(\Rightarrow)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(N)</td>
<td>3</td>
</tr>
<tr>
<td>(i)</td>
<td>0</td>
</tr>
<tr>
<td>(i_1)</td>
<td>1</td>
</tr>
<tr>
<td>(i_2)</td>
<td>2</td>
</tr>
<tr>
<td>(i_4)</td>
<td>3</td>
</tr>
<tr>
<td>(next)</td>
<td>2</td>
</tr>
<tr>
<td>(data_0)</td>
<td>&lt;2,6,5&gt;</td>
</tr>
<tr>
<td>(cookie)</td>
<td>6</td>
</tr>
</tbody>
</table>

Handling Multiple Paths

- 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: $\phi$ functions in SSA

```plaintext
1 if (i > 0) {
  1 $\phi_1 = (i_0 > 0)$
  2 $a_1 = 2$
  3 $b_1 = 3$
} else
  6 $a = 3$
  7 $b = 2$

8}                    

9  $c = a+b$;
```

- Assert $A$: $c_3 = 5$
- Is $F \land \neg A$ satisfiable?
  \[
  \phi_1 = (i_0 > 0) \land (\phi_1 \rightarrow c_3 = 5) \land (\neg \phi_1 \rightarrow c_3 = 5) \land (c_3 \neq 5)
  \]

A Resolution Example

- A resolution rule in propositional logic:

```plaintext
Given $p \land A$ and $\neg p \land B$, add the resolvent $A \lor B$
```

- Is $F \land \neg A$ satisfiable?
  \[
  \phi_1 = (i_0 > 0) \land (\phi_1 \rightarrow c_3 = 5) \land (\neg \phi_1 \rightarrow c_3 = 5) \land (c_3 \neq 5)
  \]

- Recall: $p \rightarrow q \equiv \neg p \lor q$
  \[
  \phi_1 = (i_0 > 0) \land (\neg \phi_1 \lor c_3 = 5) \land (\phi_1 \lor c_3 = 5) \land (c_3 \neq 5)
  \]

- $F \land \neg A$ is not satisfiable
- The assertion $A$ is true.
Loops

- Optimistically: Unroll two times

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

Loops: Apply SSA

```c
1 \phi_1 = (i_0 < next_0);
2 \phi_2 = (data_0[i_0] == cookie_0);
3 i_1 = i_0 + 1;
4 if (i < next) {
5 \phi_3 = \phi_2 ? i_1 : i_0;
6 i_2 = i_1 + 1;
7 if (i < next) {
8 \phi_4 = \phi_3 ? i_2 : i_1;
9 i_3 = i_2 + 1;
10 if (data[i] == cookie) {
11 i = i + 1;
12 else
13 \phi_0 = \phi_4 ? i : i_0;
14 i = i + 1;
15 }
16 }
17 }
```
Major Categories of Program Analysis Tools

<table>
<thead>
<tr>
<th>Static Property Based</th>
<th>Dynamic Execution Based</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Complete (Small programs)</strong></td>
<td><strong>Incomplete (Large programs)</strong></td>
</tr>
<tr>
<td>Verification</td>
<td>Static Analysis (Data flow)</td>
</tr>
<tr>
<td>Prove a property in a program</td>
<td>Abstract the program conservatively</td>
</tr>
<tr>
<td>Floyd-Hoare logic:</td>
<td>Check a property</td>
</tr>
<tr>
<td>(pre-condition) &amp; (post-condition)</td>
<td>Sound: no false-negatives--find all bugs</td>
</tr>
<tr>
<td>Applicable to small programs</td>
<td>False-positives: true warnings</td>
</tr>
<tr>
<td></td>
<td>Too imprecise is useless</td>
</tr>
<tr>
<td>(Symbolic) Model Checking (SMT/BDD)</td>
<td>Test case generation (SMT/BDD)</td>
</tr>
<tr>
<td>Given a system model (sw/hw), simulate the execution</td>
<td>Check a property opportunistically (e.g. unroll loops twice)</td>
</tr>
<tr>
<td>to check if a property is true for all possible inputs.</td>
<td>Use analysis to generate test inputs</td>
</tr>
<tr>
<td>Symbolic: many states all at once</td>
<td>No false-positives: generate a test</td>
</tr>
<tr>
<td></td>
<td>False-negatives: cannot find all bugs</td>
</tr>
<tr>
<td></td>
<td>No correctness/security guarantees</td>
</tr>
</tbody>
</table>

2. SMT Example with Linear Inequalities, Function Theories

\[ x \geq 0 \land f(x) \geq 0 \land f(y) \geq 0 \land x \neq y \]

Functions are “uninterpreted”:
Assumed to be pure, returning the same value for each input

This formula is satisfiable:
Example model
\[
\begin{align*}
x & \mapsto 1 \\
y & \mapsto 2 \\
f(1) & \mapsto 0 \\
f(2) & \mapsto 1
\end{align*}
\]
SMT Example: with Arrays

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

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

By arithmetic, this is equivalent to

\[ b + 2 = c \land f(\text{read}(\text{write}(a, b, 3), b)) \neq f(3) \]

By array theory axiom, read(write(v, i, x), i) = x

\[ b + 2 = c \land f(3) \neq f(3) \]

By 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 is a decision procedure that checks whether the truth assignments are satisfiable in the theories

Example of a Basic Algorithm

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

SAT SOLVER

1. Choose a model M

   - 1, 2, 3, 4

   - 1, 3

   - 1, 2, 3, 4

   - 1, 2, 3, 4

   - 1, 2, 3

   - 1, 2, 3

THEORY SOLVER

(Empty uninterpreted functions)

- Send F unsat
- Send F unsat
- Send F unsat
- Send F unsat
- Send F unsat
- Send F
Basic Algorithm

- **T-conflict:** check for conflicts with respect to theory T

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)
- Propagate the deduced assignments
- Make one decision at a time
- Check each assignment incrementally, not all at once.
  (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

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

<table>
<thead>
<tr>
<th>M</th>
<th>F</th>
<th>C</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(T \lor 3 \lor 4)</td>
<td></td>
<td>Propagate+, OK</td>
</tr>
<tr>
<td>(\overline{T})</td>
<td>(1 \lor 2 \lor 3 \lor 4)</td>
<td>(\overline{T} \lor 2 \lor 4)</td>
<td>Decide</td>
</tr>
<tr>
<td>(T \lor 2)</td>
<td>(1 \lor 2 \lor 3 \lor 4)</td>
<td>(T \lor 2 \lor 4)</td>
<td>T-Conflict</td>
</tr>
<tr>
<td>(T \lor 2)</td>
<td>(1 \lor 2 \lor 3 \lor 4)</td>
<td>(T \lor 2 \lor 4)</td>
<td>Learn</td>
</tr>
<tr>
<td>(T)</td>
<td>(1 \lor 2 \lor 3 \lor 4)</td>
<td>(T \lor 2 \lor 4)</td>
<td>Restart</td>
</tr>
<tr>
<td>(T \lor 2 \lor 3)</td>
<td>(1 \lor 2 \lor 3 \lor 4)</td>
<td>(T \lor 2 \lor 4)</td>
<td>Propagate+</td>
</tr>
<tr>
<td>(T \lor 2 \lor 3)</td>
<td>(1 \lor 2 \lor 3 \lor 4)</td>
<td>(T \lor 3 \lor 4 \lor 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 \(C := T_1 \lor \cdots \lor T_n\)
  
  **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 \(F\)
  
  **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>
<tbody>
<tr>
<td><strong>Propagate</strong></td>
<td>Deduce the truth value of a literal from M and F</td>
</tr>
<tr>
<td>( l_1 \lor \ldots \lor l_n \in F )</td>
<td>( T_1, \ldots, T_n \in M )</td>
</tr>
<tr>
<td>( l \notin M )</td>
<td>( M := M \setminus {l} )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Decide</strong></td>
<td>Guess a truth value</td>
</tr>
<tr>
<td>( l \in \text{Lit}(F) )</td>
<td>( l, \overline{l} \notin M )</td>
</tr>
<tr>
<td>( M := M \cup {l} \cup {\overline{l}} )</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>T-Conflict</strong></td>
<td>If all the literals ( l_1, \ldots, l_n ) in M cannot be satisfied by T, set the conflict clause</td>
</tr>
<tr>
<td>( C = \text{no} )</td>
<td>( l_1, \ldots, l_n \in M )</td>
</tr>
<tr>
<td>( l_1, \ldots, l_n \models \bot )</td>
<td>( C := T_1 \lor \ldots \lor T_n )</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Learn</strong></td>
<td>Add the new learned constraint to formula F</td>
</tr>
<tr>
<td>( F \models C )</td>
<td>( C \notin F )</td>
</tr>
<tr>
<td>( F := F \cup {C} )</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Restart</strong></td>
<td>Restart the SAT solver</td>
</tr>
<tr>
<td>( M := M^{[i]} )</td>
<td>( C := \text{no} )</td>
</tr>
</tbody>
</table>

Each Decide defines a new level \( M^{[i]} \) means Model M up to level \( i \)
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)

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, \exists a, \forall \exists a</td>
<td>\exists a, \forall \exists a</td>
<td>Propagate+</td>
</tr>
<tr>
<td>1 \rightarrow 2</td>
<td>1, \exists a, \forall \exists a</td>
<td>\exists a, \forall \exists a</td>
<td>T-Propagate (1 \models 2)</td>
</tr>
<tr>
<td>1 \rightarrow 2 \rightarrow 3</td>
<td>1, \exists a, \forall \exists a</td>
<td>\exists a, \forall \exists a</td>
<td>T-Propagate (1, \exists a \models \exists a)</td>
</tr>
<tr>
<td>1 \rightarrow 2 \rightarrow 3</td>
<td>1, \exists a, \forall \exists a</td>
<td>\exists a, \forall \exists a</td>
<td>Conflict</td>
</tr>
<tr>
<td>fail</td>
<td>fail</td>
<td>fail</td>
<td>Fail</td>
</tr>
</tbody>
</table>

Notation:

1 \models_2 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 $l_1, \ldots, l_n$ in $M$ cannot be satisfied by $T$, set the conflict clause $C := T_1 \lor \cdots \lor T_n$
- **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

B. T-Propagate: Rules

\[
\text{T-Propagate } \frac{I \in \text{Lit}(F) \quad M \models I \quad I \notin M}{M := M \cup I}
\]
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)

C. Backjumping: Example

\( F := \{ 1, 1 \lor 2, 3 \lor 4, 3 \lor 6, 5 \lor 7, 2 \lor 5 \lor 6 \lor 7 \} \)

<table>
<thead>
<tr>
<th>M</th>
<th>F</th>
<th>C</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>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 * 3</td>
<td>( F )</td>
<td></td>
<td>Decide</td>
</tr>
<tr>
<td>12 * 34</td>
<td>( F )</td>
<td></td>
<td>Propagate</td>
</tr>
<tr>
<td>12 * 34 * 5</td>
<td>( F )</td>
<td></td>
<td>Decide</td>
</tr>
<tr>
<td>12 * 34 * 5 * 6</td>
<td>( F )</td>
<td></td>
<td>Propagate</td>
</tr>
<tr>
<td>12 * 34 * 5 * 6</td>
<td>( F )</td>
<td>( 2 \lor 3 \lor 6 \lor 7 )</td>
<td>Conflict</td>
</tr>
</tbody>
</table>

M. Lam   CS243: SMT   31

M. Lam   CS243: SMT   32
C. Backjumping: Example Details

\[ F := \{1, 1v2, 3v4, 3v6, 5v5, 7, 2v3v6v7\} \]
\[ M := 12*34*567 \]
\[ C := \overline{2v3v6v7} \]

- Conflict: \( \overline{2v3v6v7} \) last literal choice is 7
- Explain: Choice of 7 is due to \( 1v2v3v7 \)
- Learn: \( 1v2v3v6 = \text{resolvent of } \overline{2v3v6v7} \) and \( 1v2v3v7 \)
- Conflict: \( 1v2v3v6 \) last literal choice is 6
- Explain: Choice of 6 is due to \( \overline{3v5} \)
- Learn: \( 1v2v3v5 = \text{resolvent of } 1v2v3v6 \) and \( \overline{3v5} \)
- Conflict: \( 1v2v3v5 \)
- 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 \( \overline{5} \)
C. Backjumping: Algorithm

- If \( M \) is T-unsatisfiable, backtrack to some point where the assignment was still T-satisfiable

- Find the root cause that causes the conflict \( C \)
  - **Explain**: Given conflict \( C \) involving latest choice \( l \), \( T \) chosen due to clause \( C_i \) in \( F \) (explanation), new conflict = resolvent of \( C \) and \( C_i \)
  - Since \( l \) is forced -> not the root cause, backtracking on \( l \) is meaningless
  - The resolvent distills down the constraint, eliminating the choice of \( l \)
  - Repeat application of "Explain" until a decision was made

- 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**
\[
\begin{align*}
C &= \text{no} \quad l_1 \lor \ldots \lor l_n \in F \\
C &= l_1 \lor \ldots \lor l_n \in M
\end{align*}
\]

**Explain**
\[
\begin{align*}
C &= l \lor D \\
C &= l_1 \lor \ldots \lor l_n \lor l_1 \lor \ldots \lor l_n \in F \\
C &= l_1 \lor \ldots \lor l_n \lor l_1 \lor \ldots \lor l_n \in M
\end{align*}
\]

**Backjump**
\[
\begin{align*}
C &= l_1 \lor \ldots \lor l_n \lor l \\
C &= \text{no} \quad M := M^{[i]} \lor l
\end{align*}
\]

- \( l \in M^{[i]} \) if \( l \) occurs before \( l_i \) 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

\[ \text{Fail if there is no decision to roll back} \]

\[
\begin{array}{c}
\text{Fail} \\
\vdash l_1 \lor \ldots \lor l_n \in F \\
\vdash \overline{T_1}, \ldots, \overline{T_n} \in M \\
\text{fail}
\end{array}
\]

with

\[ \text{Fail if there is a conflict and there is no decision to roll back} \]

\[
\begin{array}{c}
\text{Fail} \\
\vdash C \neq \text{no} \\
\vdash l_1 \lor \ldots \lor l_n \in M \\
\text{fail}
\end{array}
\]

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 \( C := \overline{l_1}, \ldots, \overline{l_n} \)

- **Explain**: Given conflict \( C \) involving latest choice \( l \), \( T \)chosen due to clause \( C_i \) in \( F \) (explanation), new conflict = resolvent of \( C \) and \( C_i \)

- **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)

- Many more optimizations to handle combinations of theory etc

- Practical tool: Z3 SMT solver
  - A widely used, open-source project from Microsoft

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.
  http://theory.stanford.edu/~barrett/pubs/BSST09-abstract.html

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