Program Analysis and Constraint Programming

Joxan Jaffar

National University of Singapore

CPAIOR MasterClass, 18-19 May 2015
A trivial program whose input is values $x_1, x_2, \cdots, x_n$ subject to constraints $c_i$. A feasible path is one where the corresponding constraints is satisfiable.

$$
t = 0
\text{if } (c_1) t += \alpha_1 \text{ else } t += \beta_1 \quad // \quad \alpha_i, \beta_j \text{ are constants}
\text{if } (c_2) t += \alpha_2 \text{ else } t += \beta_2
\ldots
\text{if } (c_n) t += \alpha_n \text{ else } t += \beta_n
\text{assert( ...something about } t\ldots )
$$

- **Testing**: is there one feasible path resulting in $t \leq 99$?
- **Verification**: do all feasible paths result in $t \leq 99$?
- **Analysis**: which bound $b$ is such that for all feasible paths, $t \leq b$?

In the context of general program reasoning, many added complexities:

- no (unbounded) loops
- no functions (in particular, no external/system calls)
- no “hard” (eg. nonlinear, recursive) constraints
To see that the above program reasoning is in fact hard, model the problem in SAT/OPT:

Binary vars $x_1, x_2, \cdots, x_n$, and natural number vars $t_i$.

Feasibility Function: $f(x_1, x_2, \cdots, x_n)$

\[
f(x_1, x_2, \cdots, x_n) \]
\[
t_0 = 0 \]
\[
t_1' = t_0 + \alpha_1, t_1'' = t_0 + \beta_1 \]
\[
x_1 \rightarrow t_1 = t_1' \text{ else } t = t_1'' \]
\[
t_2' = t_1 + \alpha_2, t_2'' = t_1 + \beta_2 \]
\[
x_2 \rightarrow t_2 = t_2' \text{ else } t_2 = t_2'' \]
\[
\cdots \]
\[
t_n' = t_{n-1} + \alpha_n, t_n'' = t_{n-1} + \beta_n \]
\[
x_n \rightarrow t_n = t_n' \text{ else } t_n = t_n'' \]

- **SAT:** Is the formula conjoined with $t_n \leq 99$ satisfiable?
- **SAT:** Is the formula conjoined with $t_n > 99$ UN-satisfiable?
- **OPT:** Find max value of $t_n$ such that the formula is true.
Instances of Classic SAT/OPT Problems

\[ \langle 0 \rangle \] \( t = 0 \)
\[ \langle 1 \rangle \] if \((x_1)\) \( t += \alpha_1 \) else \( t += \beta_1 \) // \( \alpha_i, \beta_j \) are constants
\[ \langle 2 \rangle \] if \((x_2)\) \( t += \alpha_2 \) else \( t += \beta_2 \)

\[ \vdots \]
\[ \langle n \rangle \] if \((x_n)\) \( t += \alpha_n \) else \( t += \beta_n \)

assert( \ldots something about \( t \ldots \) )

- When \( \beta_i = 0 \) and the assertion is of the form \( t = \gamma \), it is instance of the sum-of-subsets problem:
  
  does a subset of \( \{ \alpha_1, \alpha_2, \cdots, \alpha_n \} \) sum to \( \gamma \)?

- Considering program points as vertices and increments as edge-costs, it is a variation of the Resource Constrainted Shortest Path (RCSP) problem

(\( \text{Resource constraint is realized after associating costs with edges} \))
What’s Special about (Traditional) Programs?

- TVA is often a special kind of Satisfiability/Optimization (SAT/OPT) problem

- What’s special about programs?
  - No global constraint(s)
  - Dynamic conditions for feasibility (path-sensitivity)
  - Dynamic condition for Optimality (context-sensitivity)
  - Lots of other PL stuff (loops, system calls, dynamic code, ...)

- Thus TVA is not often addressed with classic SAT/OPT algorithms
  - Testing addressed with dynamic inputs, and now DART
  - Verification addressed with Abstraction, and lately Interpolation
  - Analysis is addressed with Extreme Abstraction (to be fast)

- This Talk:
  - Overview of Symbolic Execution as basis for TVA
  - Emphasis on Interpolation and Dynamic Programming as core technologies
  - Can TVA techniques contribute to (some) general problems in SAT/OPT
Symbolic Execution

\[ \ell_1 \textbf{if} \ (x > y) \{ \]
\[ \ell_2 \quad x = x + y; \]
\[ \ell_3 \quad y = x - y; \]
\[ \ell_4 \quad x = x - y; \]
\[ \ell_5 \textbf{if} \ (x - y > 0) \]
\[ \ell_6 \quad \text{error}(); \]
\[ \ell_7 \]

<PP, Symbolic store, Path cond>

\[ \langle 1, (x : X, y : Y), \text{true} \rangle \]
\[ \text{assume}(x > y) \]
\[ \langle 2, (x : X, y : Y), X > Y \rangle \]
\[ x := x + y \]
\[ \langle 3, (x : X + Y, y : Y), X > Y \rangle \]
\[ y := x - y \]
\[ \langle 4, (x : X + Y, y : X), X > Y \rangle \]
\[ x := x - y \]
\[ \langle 5, (x : Y, y : X), X > Y \rangle \]
\[ \text{assume}(x - y \leq 0) \]
\[ \langle 6, (x : Y, y : X), X > Y \land Y - X > 0 \rangle \]
\[ \text{assume}(x - y > 0) \]
\[ \langle 7, (x : Y, y : X), X > Y \land Y - X \leq 0 \rangle \]
Symbolic Execution Tree and Interpolation

<0> t=0
<1> if (x<y) t += 10 else t += 11
<2> if (y<z) t += 20 else t += 22
<3> if (z<w) t += 30 else t += 33

assert( t ≤ 99)
Symbolic Execution Tree with Infeasible Path

<0> t=0
<1> if (x<y) t += 10 else t += 11
<2> if (y<z) t += 20 else t += 22
<3> if (z<x) t += 30 else t += 33

assert( t ≤ 99)
Symbolic Execution Tree and Reuse of Longest Path

Longest path in left subtree highlighted, reused in the right subtree

<0> t=0
<1> if (x<y) t += 10 else t += 11
<2> if (y<z) t += 20 else t += 22
<3> if (z<w) t += 30 else t += 33

<2> t += 10
<3> t += 11

Reuse

[ t ≤ 44] 10

[ t ≤ 66] 30

[ t ≤ 69] 30

[ t ≤ 66] 60

assert( t ≤ 99)
Symbolic Execution Tree and NO Reuse of Longest Path

Longest path in left subtree highlighted, NOT reusable in the right subtree

<0> t=0
<1> if (x<y) t += 10 else t += 11
<2> if (y<z) t += 20 else t += 22
<3> if (z<x) t += 30 else t += 33

[x < y ∧ t ≤ 44]

10

[t ≤ 44]

30

[y < z ∧ t ≤ 66]

20

[t ≤ 46]

[ y < z ∧ t ≤ 66]

32

[t ≤ 66]

31

[ t ≤ 66]

63

[z < x]

30

[t ≤ 69]

62

[ t ≤ 66]

65

33

61

[ t ≤ 66]

64

63

65

assert(t ≤ 99)
• Must not provide *wrong* information (soundness)
  – True alarms should not be missed

• Our goal: provide *precise* information
  – Reduce false alarms
  – Infeasible paths pose challenge

• “Path-sensitive” analyses are more precise
  – Path-explosion

• There is need to perform *efficient* path-sensitive analyses
• Identical symbolic states result in identical analyses
  – Can *merge* and
  – But very rare in practice

• *Interpolation & Witness paths*
  – Alternate conditions for merging
  – More likely in practice
  – Potentially exponential benefit!

\[ S_{\downarrow 1} \equiv S_{\downarrow 2} \implies \sigma_{\downarrow 1} \equiv \sigma_{\downarrow 2} \]
Merging Conditions

- **Interpolant $\Psi$**
  - Given an UNSAT formula $A \land B$, interpolant w.r.t. $A$ is s.t. $A \Rightarrow \Psi$ and $\Psi \land B$ is UNSAT
  - $\Psi$ removes information from $A$ not relevant to the unsatisfiability of $A \land B$
  - *Succinctly* captures the reason for infeasible paths in symbolic trees

- **Witness paths $\omega$**
  - Set of paths in the sub-tree that contribute to its analysis information
  - Typically only a few paths in the sub-tree
**Theorem**: If a new symbolic state $S \downarrow 2$ implies the interpolant $\Psi$ and all witness paths $\omega$ are feasible under $S \downarrow 2$, then by exploring $S \downarrow 2$ we would obtain exactly the same analysis information as before.

- Merge $S \downarrow 2$ with $S \downarrow 1$.
• If $\Psi$ holds at $S \downarrow 2$
  – $T \downarrow 2$ will contain at least those infeasible paths in $T \downarrow 1$
  – $T \downarrow 2$ will contain at most those feasible paths in $T \downarrow 1$
  – $\sigma \downarrow 2 \subseteq \sigma \downarrow 1$

• If $\omega$ is feasible at $S \downarrow 2$
  – $T \downarrow 2$ will contain at least the same analysis information in $T \downarrow 1$
  – $\sigma \downarrow 1 \subseteq \sigma \downarrow 2$
Motivation

• Given a program location \( l \) and a variable \( x \), find subset of code that affects the value of \( x \) at \( l \)
  – Software testing/debugging, optimization, verification...

```java
if (c) p = 1; else p = 0;
x = 0;
if (p > 0) x = 1;          P  P'  if (!c) z = 1;
if (x == 0) z = 1;
TARGET(z)
```

• No static slicing is effective on \( P \)
  – But it is equivalent to \( P' \) wrt target \( z \)
  – Analysis of \( P' \) is clearly easier

• Novel concept introduced: Tree Slicing
Motivation

```plaintext
if (c) p = 1; else p = 0;
x = 0;
if (p > 0) x = 1;
if (x == 0) z = 1;

TARGET(z)
```

Path-sensitive expansion

```plaintext
if (c) {
    p = 1;
x = 0;
x = 1;
}
else {
    p = 0;
x = 0;
z = 1;
}
TARGET(z)
```

Slicing

```plaintext
if (c) {
    }
else { z = 1; }
```

Post processing

```plaintext
if (!c) z = 1;
```
Why does this work?

- Slicing a program fragment is more effective when there is path-sensitivity.

- In general, *symbolic execution* displays path-sensitivity as it unfolds the path leading to the fragment.

- But path-sensitivity may not always be useful! Example...
When does this **not** work?

```plaintext
if (c) p = 1; else p = 0;
```

**Program Fragment where all values of z are unaffected by p**

**TARGET**(z)

- P′ is effectively twice the size of P but there is no benefit from the duplication due to path-sensitivity

- Worse: full path-sensitivity is **intractable**!
What is the solution?

• Some form of *merging*
  – At every merge point: original CFG
  – Not at all: full SE tree (intractable)

• Exactly when path-sensitivity is no longer useful for slicing

• Path-Sensitively Sliced CFG (PSSCFG)
Methodology

- Symbolically execute the program generating SE tree and computing dependencies for slicing

- Merge states if it does not cause loss of slicing (dependency) information
  - How to detect this?
  - Merging conditions – interpolant & witness paths
Tree Slicing

- Once SE tree is generated, apply slicing on the tree itself instead of on program statements
  - Rules are similar to traditional slicing using dependency sets
  - Must also consider infeasible paths and contexts in the tree (example soon)

- Advantage: the same program fragment can be sliced from one part of the tree but not another
  - Not applicable to static slicing!
Example: generate SE tree

\[
\text{if}(\text{read}(c)) \ \text{flag}=1 \\
\quad \text{else} \ \text{flag}=0 \\
x=2 \\
\text{if}(\text{read}(d)) \ y=4 \\
\quad \text{else} \ y=5 \\
\text{if}(\text{flag}) \ z=y+x \\
\quad \text{else} \ z=x+1 \\
\text{TARGET}(z) \\
\]

\[
c \land \text{flag}=1 \land x=2 \\
\neg d \land y=5 \\
\Rightarrow \\
\text{flag}=1
\]
Tree Slicing

Example: apply Tree Slicing

```plaintext
if(read(c)) flag=1
  else flag=0
x=2
if(read(d)) y=4
  else y=5
if(flag) z=y+x
  else z=x+1
TARGET(z)
```
Example: “decompile” to C

if(read(c)) flag=1
  else flag=0
x=2
if(read(d)) y=4
  else y=5
if(flag) z=y+x
  else z=x+1
TARGET(z)

if(read(c)) {
  x=2
  if(read(d))
    y=4
    else
      y=5
      z=y+x
  }
else {
  x=2
  z=x+1
}
Example: benefits of PSSCFG

\[
P
\begin{align*}
&\text{if(read(c)) flag=1} \\
&\quad \text{else flag=0} \\
&\quad x=2 \\
&\quad \text{if(read(d)) y=4} \\
&\quad \quad \text{else y=5} \\
&\quad \text{if(flag) z=y+x} \\
&\quad \quad \text{else z=x+1} \\
&\text{TARGET(z)}
\end{align*}
\]

\[
P'
\begin{align*}
&\text{if(read(c))} \\
&\quad \{ \\
&\quad \quad x=2 \\
&\quad \quad \text{if(read(d))} \\
&\quad \quad \quad y=4 \\
&\quad \quad \quad \text{else} \\
&\quad \quad \quad \quad y=5 \\
&\quad \quad \quad \quad z=y+x \\
&\quad \quad \} \\
&\quad \text{else} \\
&\quad \{ \\
&\quad \quad x=2 \\
&\quad \quad z=x+1 \\
&\quad \} \\
\end{align*}
\]

- Faster verification and analysis
  - Less # of paths/variables
- Less constraint solving for concolic testing
  - on P, always generates values for c and d
  - on P', generates d only if c was non-zero

- Completely off-the-shelf transformation!
Experiments

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Lines of code</th>
<th>Blow up</th>
<th>PSS Time</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Orig</td>
<td>St.slice</td>
<td>PSS</td>
</tr>
<tr>
<td>cdaudio</td>
<td>1817</td>
<td>1599</td>
<td>4452</td>
</tr>
<tr>
<td>diskperf</td>
<td>937</td>
<td>706</td>
<td>2967</td>
</tr>
<tr>
<td>floppy</td>
<td>1005</td>
<td>766</td>
<td>2086</td>
</tr>
<tr>
<td>floppy2</td>
<td>1513</td>
<td>1250</td>
<td>3507</td>
</tr>
<tr>
<td>kbfiltr</td>
<td>549</td>
<td>275</td>
<td>170</td>
</tr>
<tr>
<td>kbfiltr2</td>
<td>782</td>
<td>492</td>
<td>410</td>
</tr>
<tr>
<td>tcas</td>
<td>286</td>
<td>227</td>
<td>311</td>
</tr>
</tbody>
</table>

- Implemented on TRACER symbolic execution framework
- Manageable blow-up and generation time of PSSCFG
  - Compared with static slice from Frama-C
Experiments

- Off-the-shelf concolic tester CREST gains 3.1 times speed-up (24 hrs vs 8 hrs)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Testing Time</th>
<th>Speed up</th>
<th>#Solver calls</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>St.slicce</td>
<td>PSS</td>
<td></td>
</tr>
<tr>
<td>cdaudio</td>
<td>1m30s</td>
<td>43s</td>
<td>2.1</td>
</tr>
<tr>
<td></td>
<td>16k</td>
<td>7k</td>
<td></td>
</tr>
<tr>
<td>diskperf</td>
<td>900m</td>
<td>34m</td>
<td>26.5</td>
</tr>
<tr>
<td></td>
<td>26mil</td>
<td>1mil</td>
<td></td>
</tr>
<tr>
<td>floppy</td>
<td>9m6s</td>
<td>24s</td>
<td>22.8</td>
</tr>
<tr>
<td></td>
<td>260K</td>
<td>4k</td>
<td></td>
</tr>
<tr>
<td>floppy2</td>
<td>525m</td>
<td>429m</td>
<td>1.2</td>
</tr>
<tr>
<td></td>
<td>613k</td>
<td>479k</td>
<td></td>
</tr>
<tr>
<td>kbfiltr</td>
<td>2s</td>
<td>1s</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>63</td>
<td>52</td>
<td></td>
</tr>
<tr>
<td>kbfiltr2</td>
<td>22s</td>
<td>6s</td>
<td>3.7</td>
</tr>
<tr>
<td></td>
<td>7k</td>
<td>2k</td>
<td></td>
</tr>
<tr>
<td>tcas</td>
<td>4s</td>
<td>1s</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>1.5k</td>
<td>188</td>
<td></td>
</tr>
</tbody>
</table>

Total: 23h56m vs 7h44m (3.1 times speed-up)
Experiments

- Off-the-shelf verifiers gain 1.5 to 5.8 times speedup
  - ARMC unable to terminate on original program

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Verification Time</th>
<th>Speed up</th>
<th>Verification Time</th>
<th>Speed up</th>
<th>Verification Time</th>
<th>Speed up</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>St.Slice</td>
<td>PSS</td>
<td>St.Slice</td>
<td>PSS</td>
<td>St.Slice</td>
<td>PSS</td>
</tr>
<tr>
<td>cdaudio</td>
<td>95s</td>
<td>14s</td>
<td>6.8</td>
<td>T/O</td>
<td>21s</td>
<td>N/A</td>
</tr>
<tr>
<td>diskperf</td>
<td>146s</td>
<td>18s</td>
<td>8.1</td>
<td>T/O</td>
<td>6s</td>
<td>N/A</td>
</tr>
<tr>
<td>floppy</td>
<td>34s</td>
<td>8s</td>
<td>4.3</td>
<td>259s</td>
<td>6s</td>
<td>43.17</td>
</tr>
<tr>
<td>floppy2</td>
<td>39s</td>
<td>13s</td>
<td>3.0</td>
<td>T/O</td>
<td>17s</td>
<td>N/A</td>
</tr>
<tr>
<td>kbfiltre</td>
<td>4s</td>
<td>1s</td>
<td>4.0</td>
<td>3s</td>
<td>1s</td>
<td>3.00</td>
</tr>
<tr>
<td>kbfiltre2</td>
<td>8s</td>
<td>2s</td>
<td>4.0</td>
<td>13s</td>
<td>2s</td>
<td>6.50</td>
</tr>
<tr>
<td>tcas</td>
<td>3s</td>
<td>1s</td>
<td>3.0</td>
<td>3s</td>
<td>1s</td>
<td>3.00</td>
</tr>
<tr>
<td></td>
<td>329s</td>
<td>57s</td>
<td>5.8</td>
<td>T/O</td>
<td>54s</td>
<td>N/A</td>
</tr>
</tbody>
</table>
The TRACER framework

- Compiles LLVM and C into CLP(R) transitions
- Performs depth-first symbolic execution with interpolation
- Performs Two Kinds of Speculation
- Applied to Testing, Verification and Analysis (WCET, Var Dependency, Taint)
- For WCET, (a) loops, (b) cache
- Ongoing work: (a) explicit heaps, frame rule, recursive definitions (b) String constraints and directed testing of web programs
Comparison with AI and CEGAR

- Uses a priori abstraction
- Often fails to verify (counterexample), so refinement is needed
- A refinement is a new abstract domain, applied **globally** in the next iteration
- An iterations **agnostic** to previous iterations
Comparison with Classic Hoare Verification

- Weakest precondition based

- Generates a tree (like Sym Exec) preconditions, exponential in size

- Advantage: focus only on relevant variables and properties

- Disadvantage 1: ignores preconditions

- Disadvantage 2: not amenable to abstraction, the standard mechanism for reducting search space in verification (eg: loop invariants)

- Currently only used in niche full-function verifiers (eg: Daphny)
Comparison with SMT

- Represents disjunctions of constraints as boolean vars
- Uses constraint solver with interpolation for conjunctions, working a DPLL process for the booleans
- The constraint solver is aware only of one decision chain of the booleans at any one time
- For representing symbolic execution, no loops
- Huge advantage: easy to deploy as a black box
Comparison with Concolic Testing

- Concolic = Concretization (dynamization) + Directed (selected backtracking)

- Concolic Testing depends on “Pure Dumb Luck” (PDL)

- No exploitation of interpolation for potentially exponential reduction

- Huge advantage: “difficult code” like unbounded loops, nonlinear constraints, system functions
Part II: Incremental Analysis

- Stop anytime with sound analysis
- Progressively improve with increased budget
- Can often terminate early
Hybrid Symbolic Execution Tree

- HSET = SET but some subtrees replaced by an abstract node ⊗
- Each abstract node has upper bound analysis (eg. indicated by “u”)
- Other nodes may have a lower bound analysis (eg. indicated by “l”)
Refining a HSET

- The upper bound can efficiently computed (traditional Abstract Interpretation)
- The upper bound is represented by a witness path
- However, the witness may be spurious
- If not spurious, refine the HSET by keeping the path and abstracting the offshoots of the path as needed.

(a) [Diagram showing initial state]

(b) [Diagram showing refined state]
Refining a HSET (2)

- **Domination**: ignore subtrees whose upper bound does not exceed all lower bounds
- **Goal directed choice**: always choose a non-dominated abstract node to refine
- **Customized termination**: Stop when (all) upper bound is close enough to the lower
- **Early termination**: Obtain exact analysis when lower and upper bounds meet
## Benchmarks

<table>
<thead>
<tr>
<th>Benchmark</th>
<th># V</th>
<th>AI</th>
<th>Full SE</th>
<th>Incremental</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td># TV</td>
<td># TV</td>
<td>Time</td>
</tr>
<tr>
<td>cdaudio</td>
<td>1288</td>
<td>10663</td>
<td>9370</td>
<td>28 s</td>
</tr>
<tr>
<td>diskperf</td>
<td>1255</td>
<td>33598</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>floppy</td>
<td>1524</td>
<td>16627</td>
<td>13784</td>
<td>19 s</td>
</tr>
<tr>
<td>ssh</td>
<td>2213</td>
<td>12394</td>
<td>6075</td>
<td>17 s</td>
</tr>
<tr>
<td>nsichneu</td>
<td>2540</td>
<td>206788</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>tcas</td>
<td>235</td>
<td>29305</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>statemate</td>
<td>1187</td>
<td>31281</td>
<td>∞</td>
<td>285 s</td>
</tr>
</tbody>
</table>

**Table:** WCET Analysis results for AI based, SE based, and our incremental algorithm. ∞ means timeout or out-of-memory.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th># V</th>
<th>AI</th>
<th>Full SE</th>
<th>Incremental</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td># TV</td>
<td># TV</td>
<td>Time</td>
</tr>
<tr>
<td>cdaudio</td>
<td>330</td>
<td>50</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>diskperf</td>
<td>185</td>
<td>36</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>floppy</td>
<td>330</td>
<td>27</td>
<td>∞</td>
<td>∞</td>
</tr>
<tr>
<td>ssh</td>
<td>63</td>
<td>57</td>
<td>53</td>
<td>1 s</td>
</tr>
<tr>
<td>nsichneu</td>
<td>22</td>
<td>16</td>
<td>16</td>
<td>4 s</td>
</tr>
<tr>
<td>tcas</td>
<td>41</td>
<td>15</td>
<td>15</td>
<td>4 s</td>
</tr>
<tr>
<td>statemate</td>
<td>119</td>
<td>67</td>
<td>∞</td>
<td>∞</td>
</tr>
</tbody>
</table>

**Table:** Taint Analysis results. # TV measures the number of tainted variables.
Conclusion

- Verification, by exhaustive search over Symbolic Execution Tree, is akin to SAT
- Analysis of a Symbolic Execution Tree is a form of an OPT problem
- TVA techniques cannot directly use SAT/OPT methods, but use specialized methods (interpolation, witnesses, spines, ...)
- Some SAT/OPT problems can be FORMULATED as a TVA problem
- The specialized methods for TVA the offers new solutions to certain SAT/OPT problems
References

- Joxan Jaffar, Andrew Santosa and Razvan Voicu, Efficient Memoization for Dynamic Programming with Adhoc Constraints, AAAI’08

- Joxan Jaffar, Andrew Santosa and Razvan Voicu, An Interpolation Method for CLP traversal, CP’09.
  (See also McMillan, CAV’10 and CAV’14, who calls this algorithm Lazy Annotations)


- Joxan Jaffar and Vijayaraghavan Murali, A Path-Sensitively Sliced Control Flow Graph, FSE’2014