Program Analysis and Constraint Programming

Joxan Jaffar

National University of Singapore

CPAIOR MasterClass, 18-19 May 2015

Program Testing, Verification, Analysis (TVA) ... VS ... Satifiability/Optimization (SAT/OPT)

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 contraints is satisfiable.

```
t = 0
if (c_1) t += α_1 else t += β_1 // α_i, β_j are constants
if (c_2) t += α_2 else t += β_2
...
if (c_n) t += α_n else t += β_n
assert( ...something about t... )
```

- 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

Program Testing, Verification, Analysis (TVA) ... VS ... Satifiability/Optimization (SAT/OPT)

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

$$
\begin{array}{l}
f(x_1, x_2, \cdots, x_n) \\
t_0 = 0 \\
t_1' = t_0 + \alpha_1, t_1'' = t_0 + \beta_1 \\
x_1 \rightarrow t_1 = t_1' \;\; \textit{else} \;\; t = t_1'' \\
t_2' = t_1 + \alpha_2, t_1'' = t_1 + \beta_2 \\
x_2 \rightarrow t_2 = t_2' \;\; \textit{else} \;\; t_2 = t_2'' \\
\cdots \\
t_n' = t_{n-1} + \alpha_n, t_1'' = t_{n-1} + \beta_n \\
x_n \rightarrow t_n = t_n' \;\; \textit{else} \;\; t_n = t_n''
\end{array}
$$

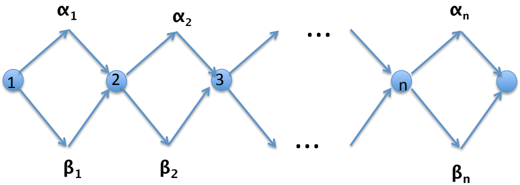
- SAT: Is the formula conjoined with $t_n \leq 99$ satisfiable?
- SAT: Is the formula conjoined with $t_n > 99$ UN-satisfiable?
- OPT: Find max value of t_n such that the formula is true.

$\langle 0 \rangle$ `t = 0`
$\langle 1 \rangle$ `if` (x_1) `t +=` α_1 `else t +=` β_1 `//` α_i, β_j `are constants`
$\langle 2 \rangle$ `if` (x_2) `t +=` α_2 `else t +=` β_2
...
$\langle n \rangle$ `if` (x_n) `t +=` α_n `else t +=` β_n
`assert(` *...something about t...* `)`

- 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 γ?

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



(Resource constraint is realized after associating costs with edges)

What's Special about (Traditional) Programs?

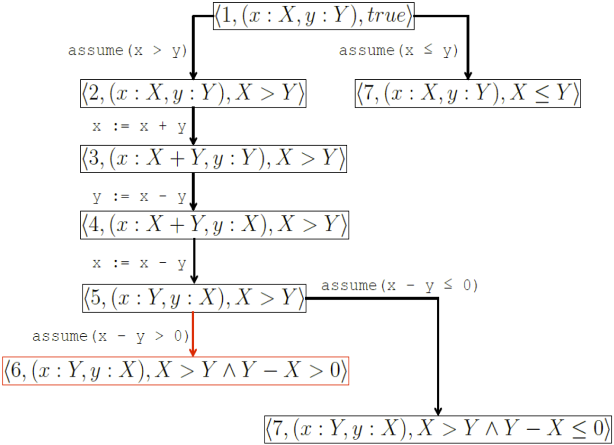
- TVA is often a special kind of Satisfiability/Optimization (SAT/OPT) problem
- What's special about programs?
  - No global constraint(s)
  - Dynamic conditions for feasibility (path-sensitivity)
  - Dynamic condition for Optimality (context-sensitivity)
  - Lots of other PL stuff (loops, system calls, 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

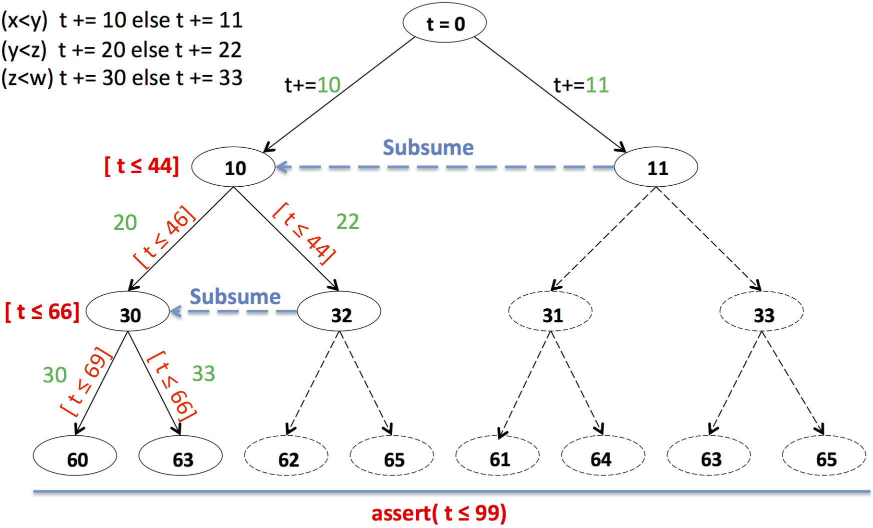
```
ℓ1  if (x > y) {
ℓ2      x = x + y;
ℓ3      y = x - y;
ℓ4      x = x - y;
ℓ5      if (x - y > 0)
ℓ6          error();
    }
ℓ7
```

ℓ_1	`assume(x > y)`	ℓ_2
ℓ_2	`x := x + y`	ℓ_3
ℓ_3	`y := x - y`	ℓ_4
ℓ_4	`x := x - y`	ℓ_5
ℓ_5	`assume(x-y > 0)`	ℓ_6
ℓ_5	`assume(x-y ≤ 0)`	ℓ_7
ℓ_1	`assume(x ≤ y)`	ℓ_7

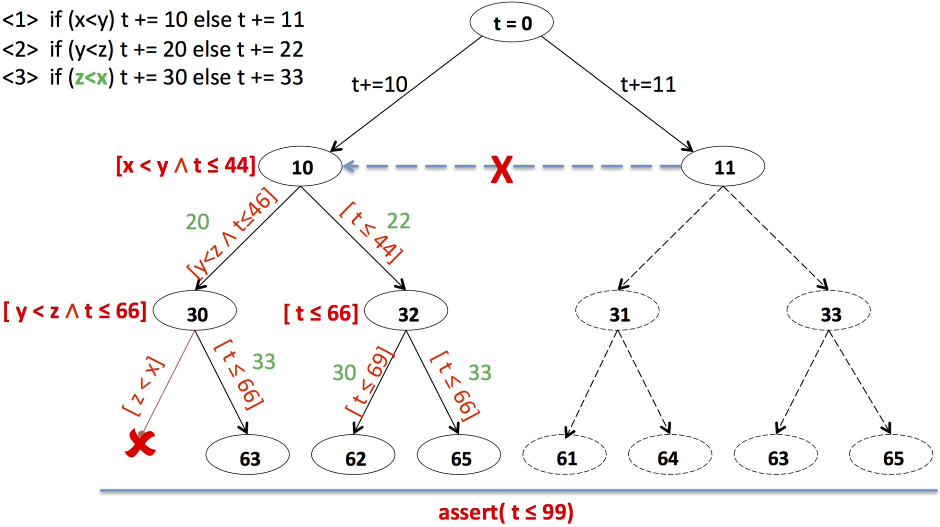
<PP, Symbolic store, Path cond>



```
<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
```



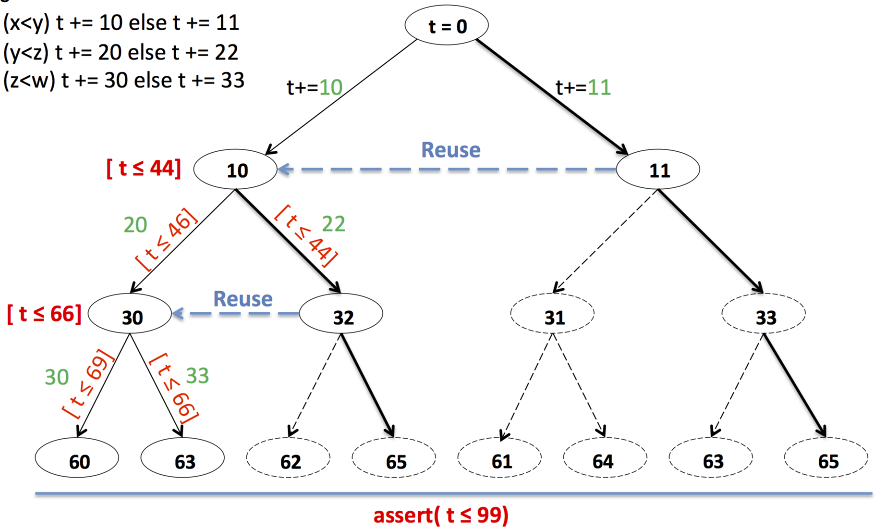
```
<0> t=0
<1>  if (x<y) t += 10 else t += 11
<2>  if (y<z) t += 20 else t += 22
<3>  if (z<x) t += 30 else t += 33
```



Symbolic Execution Tree and Reuse of Longest Path

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

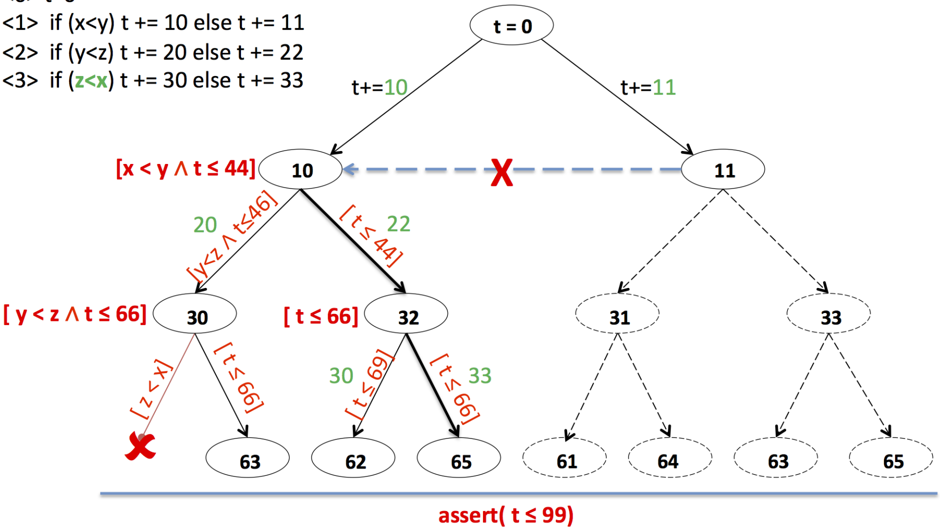
<0> t=0
<1> if (x<y) t += 10 else t += 11
<2> if (y<z) t += 20 else t += 22
<3> if (z<w) t += 30 else t += 33



Symbolic Execution Tree and NO Reuse of Longest Path

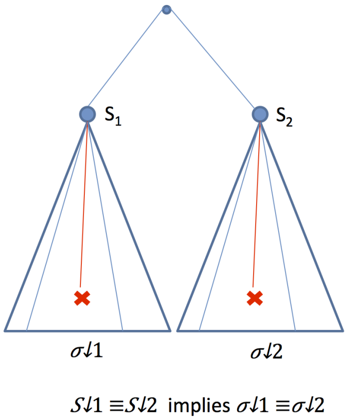
Longest path in left subtree highlighted, NOT reusable in the right subtree

<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

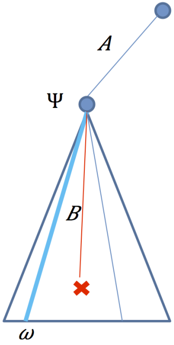


- Must not provide *wrong* information (soundness)
  - True alarms should not be missed
- Our goal: provide *precise* information
  - Reduce false alarms
  - Infeasible paths pose challenge
- “Path-sensitive” analyses are more precise
  - Path-explosion
- There is need to perform *efficient* path-sensitive 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!

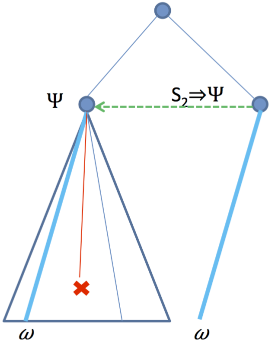


- Interpolant Ψ
  - Given an UNSAT formula $A \wedge B$, interpolant w.r.t. A is s.t. $A \Rightarrow \Psi$ and $\Psi \wedge B$ is UNSAT
  - Ψ removes information from A not relevant to the unsatisfiability of $A \wedge B$
  - *Succinctly* captures the reason for infeasible paths in symbolic trees
- Witness paths ω
  - 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 $S↓2$ implies the interpolant Ψ and all witness paths ω are feasible under $S↓2$ then by exploring $S↓2$ we would obtain exactly the same analysis information as before*

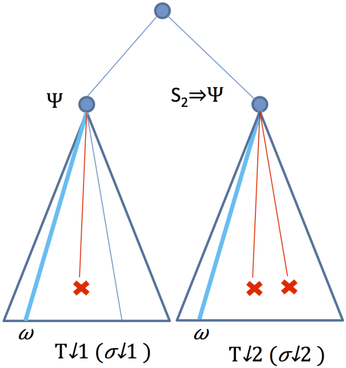
- Merge $S↓2$ with $S↓1$



Merging Conditions

- If Ψ 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 $T\downarrow 1$
  - $\sigma\downarrow 2 \sqsubseteq \sigma\downarrow 1$

- If ω is feasible at $S\downarrow 2$
  - $T\downarrow 2$ will contain *at least* the same analysis information in $T\downarrow 1$
  - $\sigma\downarrow 1 \sqsubseteq \sigma\downarrow 2$



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...

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

P → **P’**

```
if (!c) z = 1;
```

- 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

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

Path-sensitive expansion

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

Slicing

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

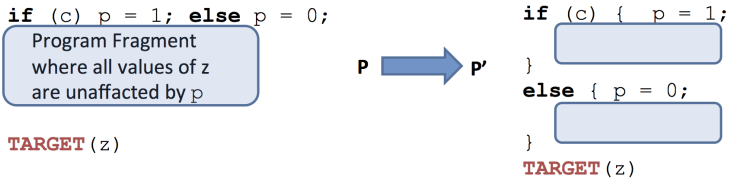
Post processing

```
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?



- 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?

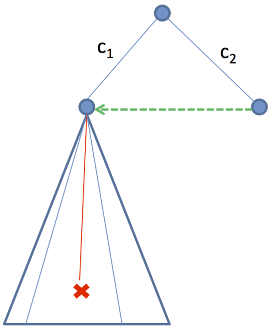
- Some form of *merging*
  - At every merge point: original CFG
  - Not at all: full SE tree (intractable)

- 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



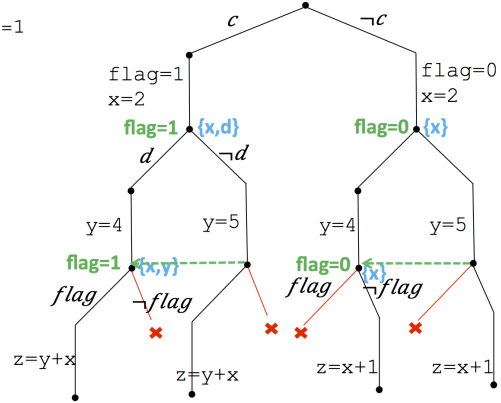
Tree Slicing

- 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

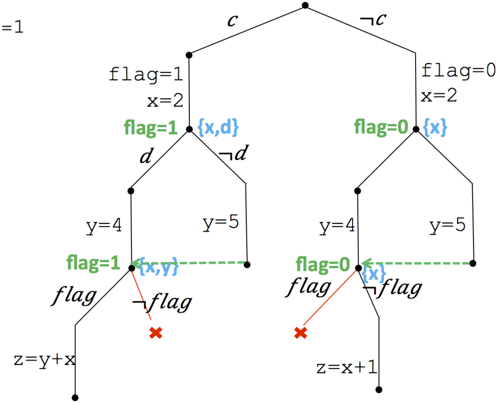
```
if(read(c)) flag=1
   else flag=0
x=2
if(read(d)) y=4
   else y=5
if(flag) z=y+x
   else z=x+1
TARGET(z)
```



$c \wedge flag=1 \wedge x=2$
$\wedge \neg d \wedge y=5$
$\Rightarrow$
$flag=1$

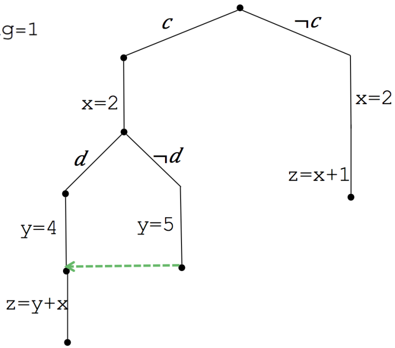
Example: apply Tree Slicing

```
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

```
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)
```

P'

```
if(read(c)) {
  x=2
  if(read(d))
    y=4
  else
    y=5
  z=y+x
}
else {
  x=2
  z=x+1
}
```

- 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

Benchmark	Lines of code			Blow up	PSS Time
	Orig	St.slice	PSS		
cdaudio	1817	1599	4452	2.78	24s
diskperf	937	706	2967	4.20	18s
floppy	1005	766	2086	2.72	7s
floppy2	1513	1250	3507	2.81	16s
kbfiltr	549	275	170	0.62	1s
kbfiltr2	782	492	410	0.83	1s
tcas	286	227	311	1.37	2s

- 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)

Benchmark	Testing Time		Speed up	#Solver calls	
	St.slice	PSS		St.slice	PSS
cdaudio	1m30s	43s	2.1	16k	7k
diskperf	900m	34m	26.5	26mil	1mil
floppy	9m6s	24s	22.8	260k	4k
floppy2	525m	429m	1.2	613k	479k
kbfiltr	2s	1s	2	63	52
kbfiltr2	22s	6s	3.7	7k	2k
tcas	4s	1s	4	1.5k	188
	23h56m	**7h44m**	**3.1**	**26.9mil**	**1.5mil**

Experiments

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

Benchmark	IMPACT Verification Time St.Slice	IMPACT Verification Time PSS	IMPACT Speed up	ARMC Verification Time St.Slice	ARMC Verification Time PSS	ARMC Speed up	CPA-CHECKER Verification Time St.Slice	CPA-CHECKER Verification Time PSS	CPA-CHECKER Speed up
cdaudio	95s	14s	6.8	T/O	21s	N/A	26s	14s	1.86
diskperf	146s	18s	8.1	T/O	6s	N/A	7s	6s	1.17
floppy	34s	8s	4.3	259s	6s	43.17	6s	5s	1.20
floppy2	39s	13s	3.0	T/O	17s	N/A	10s	8s	1.25
kbfiltr	4s	1s	4.0	3s	1s	3.00	3s	2s	1.50
kbfiltr2	8s	2s	4.0	13s	2s	6.50	4s	2s	2.00
tcas	3s	1s	3.0	3s	1s	3.00	2s	1s	2.00
	329s	**57s**	**5.8**	**T/O**	**54s**	**N/A**	**58s**	**38s**	**1.53**

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

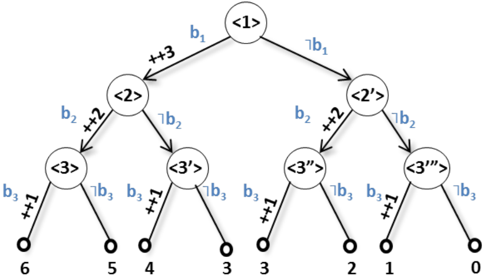
Comparison with Concolic Testing

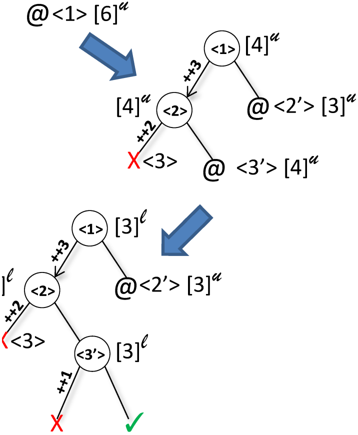
- Concolic = Concretization (dynamization) + Directed (selected backtracking)
- Concolic Testing depends on "Pure Dumb Luck" (PDL)
- No exploitation of interpolation for potentially exponential reduction
- Huge advantage: "difficult code" like unbounded loops, nonlinear constraints, system functions

Part II: Incremental Analysis

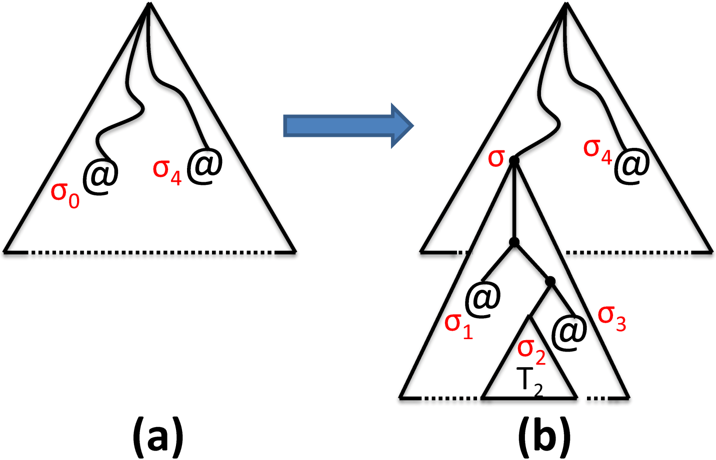
- Stop anytime with sound analysis
- Progressively improve with increased budget
- Can often terminate early

- 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")





- 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.



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

Benchmark	# V	AI	Full SE			Incremental			
		# TV	# TV	Time	Mem	$\#TV_{\mathcal{U}}$	$\#TV_{\mathcal{L}}$	Time	Mem
cdaudio	1288	10663	9370	28 s	212 MB	9370	9370	14 s	56 MB
diskperf	1255	33598	∞	∞	2 GB	29723	29723	231 s	400 MB
floppy	1524	16627	13784	19 s	136 MB	13784	13784	15 s	44 MB
ssh	2213	12394	6075	17 s	39 MB	6075	6075	17 s	51 MB
nsichneu	2540	206788	∞	∞	522 MB	52430	52430	156 s	133 MB
tcas	235	29305	∞	∞	1.4 GB	28788	23887	∞	432 MB
statemate	1187	31281	∞	285 s	∞	31151	18623	∞	767 MB

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

Benchmark	# V	AI	Full SE			Incremental			
		# TV	# TV	Time	Mem	$\#TV_{\mathcal{U}}$	$\#TV_{\mathcal{L}}$	Time	Mem
cdaudio	330	50	∞	∞	574 MB	45	45	17 s	227 MB
diskperf	185	36	∞	∞	425 MB	31	31	7 s	101 MB
floppy	330	27	∞	∞	581 MB	20	20	12 s	229 MB
ssh	63	57	53	1 s	8 MB	53	53	1 s	8 MB
nsichneu	22	16	16	4 s	24 MB	16	16	2 s	11 MB
tcas	41	15	15	4 s	55 MB	15	15	1 s	12 MB
statemate	119	67	∞	∞	770 MB	63	63	7 s	79 MB

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 $\boxed{\text{new}}$ 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, Vijayaraghavan Murali, Jorge Navas and Andrew Santosa, TRACER: A Symbolic Execution Tool for Verification, CAV'2012
- Joxan Jaffar and Vijayaraghavan Murali, A Path-Sensitively Sliced Control Flow Graph, FSE'2014