Interpolation Methods for Symbolic Execution

Duc-Hiep CHU
Advisor: Prof. Joxan JAFFAR

NUS Graduate School for Integrative Sciences and Engineering (NGS)
National University of Singapore (NUS)

March 14, 2013

Symbolic Execution

- Uses symbolic values as inputs instead of actual data
- Represents the values of program variables as symbolic expressions of the input symbolic values
- Originally introduced for testing (King [1976]; Clarke [1976])
- Subsequently used for bug finding (Cadar *et al.* [2006]) and verification condition (VC) generation (Beckert *et al.* [2007]; Jacobs and Piessens [2008]), among others

Why Symbolic Execution?

- Resembles closely human's reasoning
- Allows potentially exact reasoning
- Supports high level of automation

Challenges in Symbolic Execution

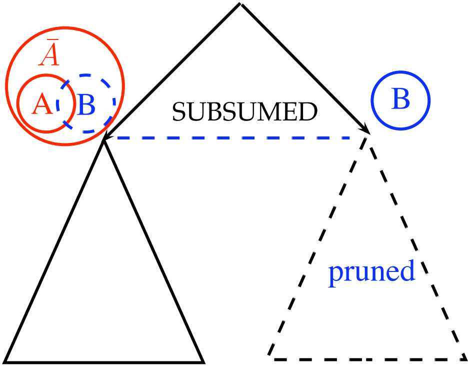
- Symbolic constraints to model real-life programs
- Constraint solving: automatically and efficiently
- The fundamental problem of path explosion

Main Contributions

- This thesis applies symbolic execution to two focus areas
  1. Path-sensitive analysis of worst-case resource usage
  2. Safety verification of concurrent systems
- We address the path explosion problem using interpolation methods
  - dynamic abstraction learning
  - dynamic reduction (pruning or reusing)
- Assumption: The symbolic execution tree is finite. Mechanisms for making that tree finite (e.g., abstraction, invariant discovery) are considered as orthogonal issues.

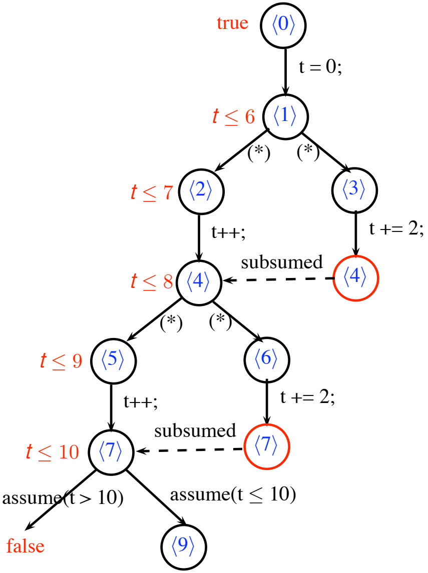
Interpolation for Program Verification (Jaffar *et al.* [2009])

- A and B share the same program point, i.e., $\ell_A = \ell_B$
- A does not subsume B
- Generalize the context of A to $\bar{A}$, aka an interpolant, while preserving the safety
- B is subsumed by $\bar{A}$



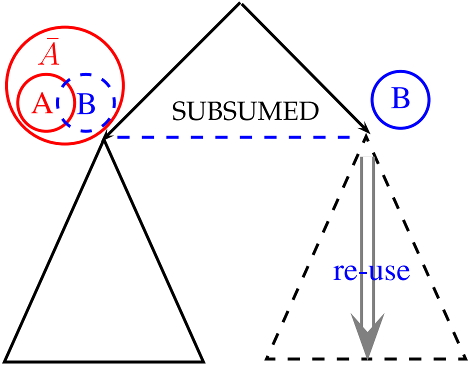
Example: Interpolation for Program Verification

```
⟨0⟩ t = 0;
⟨1⟩ if (*)
⟨2⟩     t++;
    else
⟨3⟩     t += 2;
⟨4⟩ if (*)
⟨5⟩     t++;
    else
⟨6⟩     t += 2;
⟨7⟩ if (t > 10)
⟨8⟩     error();
⟨9⟩
```



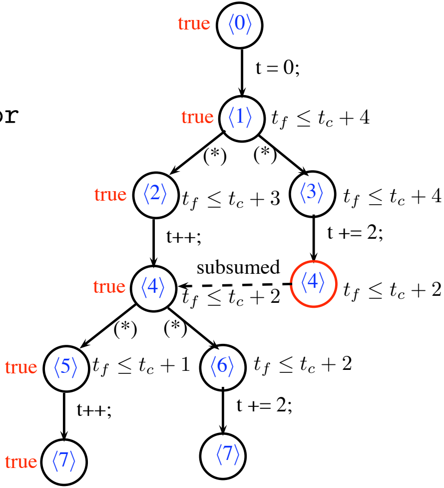
Interpolation for Program Analysis (Jaffar *et al.* [2008])

- A and B share the same program point, i.e., $\ell_A = \ell_B$
- A does not subsume B
- Generalize the context of A to $\bar{A}$, aka an interpolant, while preserving the infeasible paths
- B is subsumed by $\bar{A}$
- The summarized analysis of A can be safely reused in B



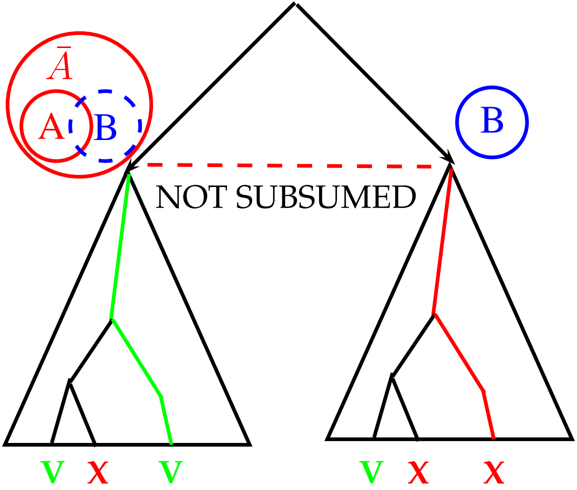
Example: Interpolation for Program Analysis

```
find a safe upper-bound for
the final value of t
⟨0⟩ t = 0;
⟨1⟩ if (*)
⟨2⟩     t++;
    else
⟨3⟩     t += 2;
⟨4⟩ if (*)
⟨5⟩     t++;
    else
⟨6⟩     t += 2;
⟨7⟩
```



Interpolation+Witness for Program Analysis (Jaffar *et al.* [2008])

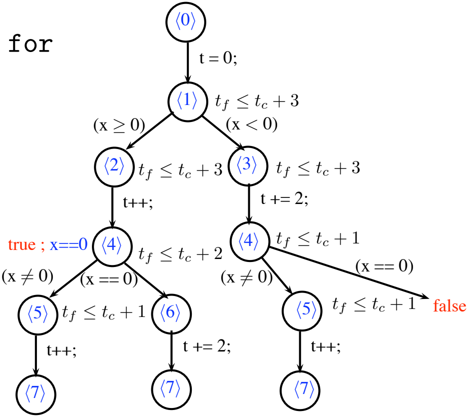
The representative path in A is infeasible in B



Example: Interpolation+Witness for Program Analysis

find a safe upper-bound for the final value of t

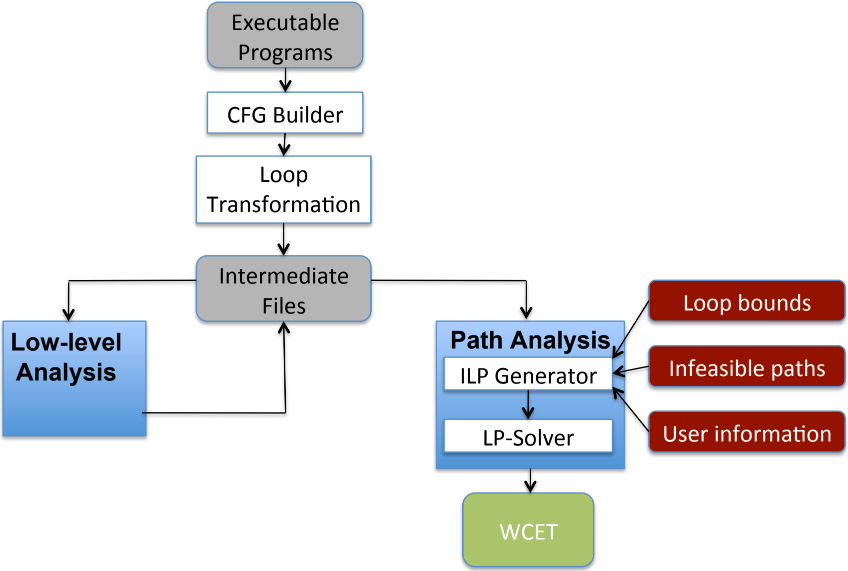
```
⟨0⟩ t = 0;
⟨1⟩ if (x >= 0)
⟨2⟩     t++;
    else
⟨3⟩     t += 2;
⟨4⟩ if (x != 0)
⟨5⟩     t++;
    else
⟨6⟩     t += 2;
⟨7⟩
```



Analysis of Worst-Case Resource Usage

- Important for designing real-time and embedded systems
- Ranges from *cumulative* resource (e.g., timing) to *non-cumulative* resource (e.g., memory high watermark)
- *Extremely hard* due to the requirement of high precision
- Redeeming factors:
    - Loops/recursions are statically bounded (i.e., termination is guaranteed)
    - The users/certifiers are on our side
- We restrict the presentation to Worst-Case Execution Time (WCET) analysis
    - Results are extensible to non-cumulative resource

Architecture of A Traditional WCET Analyzer



Implicit Path Enumeration Technique (IPET)

- Introduced by Li and Malik [1995]
- Employs Integer Linear Programming (ILP)
- Simple, elegant, fast, but *path-insensitive*
- Supports user information

Example: IPET

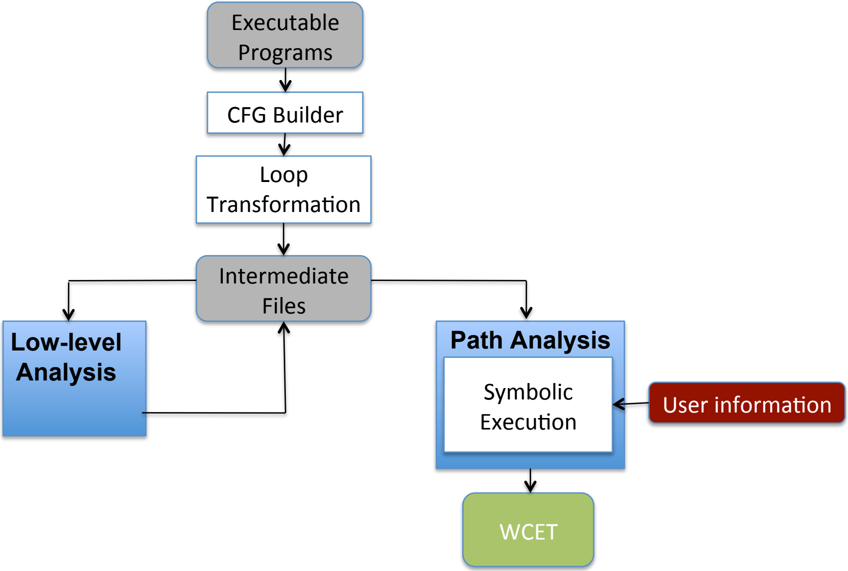
```
c1 = 0, c2 = 0, c3 = 0, i = 0, t = 0;
while (i < 9) {
    if (*) {B1: c1++; t += 10; }
    else {
        if (i == 1) {B2: c2++, t += 5; }
        else {B3: c3++; t += 1; }
    }
    i++;
    assert(c1 <= 4);
}
```

$$\text{maximize}(10 * c_1 + 5 * c_2 + 1 * c_3) \text{ wrt. } c_1 + c_2 + c_3 \leq 9 \wedge c_1 \leq 4$$

Manual Annotations

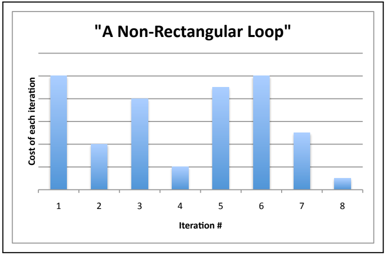
- Annotations of loop bounds
  - Is *mandatory* to produce a bound
  - Precision depends on precise loop bounds
  - Can be automated via some form of loop bound analysis: This is non-trivial due to *complicated* loops
- Annotations of infeasible paths
  - Fundamentally hard due to the exponential number of infeasible paths
  - Automation: usually ad-hoc (e.g., detecting simple conflict patterns)
- Annotations of user information (assertions) which is not readily extractable from the programs
  - Information which is too hard to automatically extract from the code
  - Additional information the users know, but not in the code

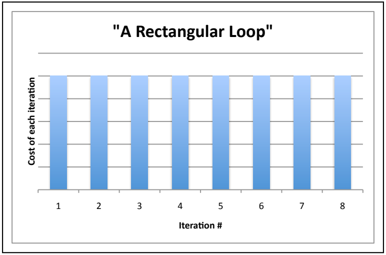
Our Method


Executable Programs
CFG Builder
Loop Transformation
Intermediate Files
Low-level Analysis
Path Analysis
Symbolic Execution
User information
WCET

Symbolic Execution with Loop Unrolling

- Is essential for capturing non-uniform behaviors of loops





- Challenge: how to make it scalable?
- The symbolic execution tree is huge
  - Its depth is at least proportional to the execution of the WCET path
  - Estimated number of states $= 2^{\{average\ length\ of\ a\ path\}}$

Solutions

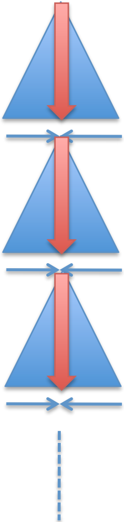
- Iteration abstraction
  - Path merging as in (Lundqvist and Stenström [1999] and Gustafsson *et al.* [2005])
  - We only perform at the end of each loop body
  - We use polyhedral domain
- Compounded summarization with interpolation for reuse
  - Summarizations are compounded both horizontally and vertically
  - Interpolants tell us when we can safely reuse
- Witness paths: tell us when we can precisely reuse

Iteration Abstraction

- Contexts are merged into one at the end of each loop iteration
- We use polyhedral domain (convex hull)
  - Capture linear relationship between variables
  - More precise compared to state-of-the-art
- Benefits:
  - Invariant constraints can be propagated through a loop
  - Common constraints in different paths of each iteration are kept
  - Non-uniform behaviors of loops can still be captured

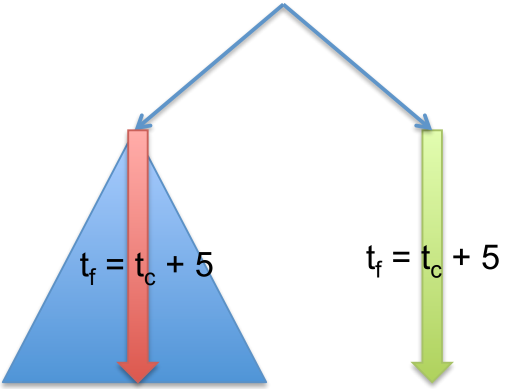
Illustration: Iteration Abstraction

Red arrows denote summarizations



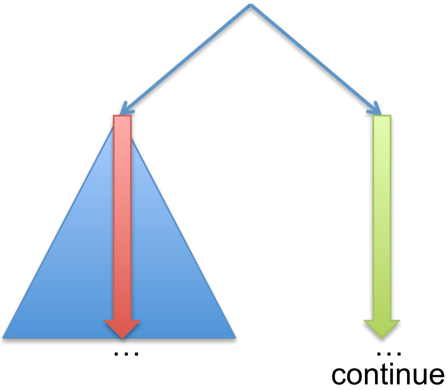
Breadth-wise Reuse of Summarization

- Green arrows denote reuse
- The condition for reuse is determined by interpolation+witness



Breadth-wise Reuse of Summarization

- With summarization for each iteration
  - The leaves of the sub-tree need not be terminal
  - We need to produce continuation contexts



Abstract Transformer

- Gives an abstract input-ouput relationship for a finite sub-tree
- Again, we compute it using polyhedral domain

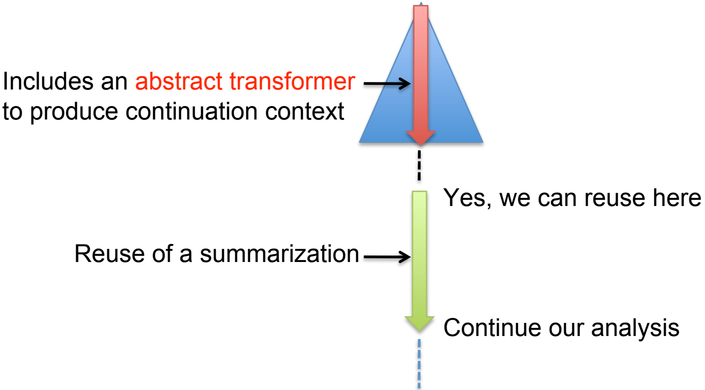
Example

`if (*) x += 1; else x += 2;`

Abstract transformer $\Delta = \mathtt{x + 1} \leq \mathtt{x'} \leq \mathtt{x + 2}$

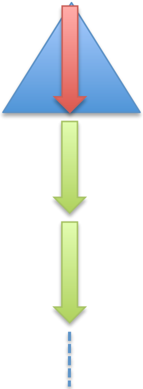
Depth-wise Reuse of Summarization

Reuse between different iterations



Depth-wise Reuse of Summarization

Analysis of a rectangular loop

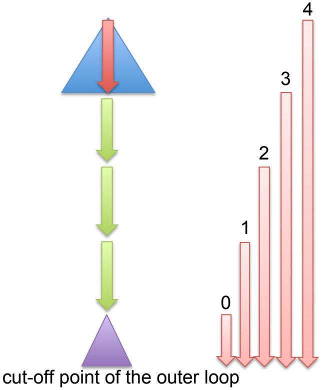


Depth-wise Loop Compression

- So far, we have shown the benefits of abstracting and summarizing each iteration of a loop
- How about summarizing the whole loop?
  - It benefits when dealing with nested loops
  - It results in depth-wise loop compression

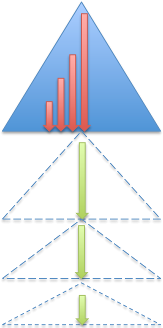
Depth-wise Loop Compression

- A serialization of summarizations for a single program point (inner loop head)
- In case of rectangular loops: we will mainly reuse 4
- In case of non-rectangular loops: 0,1,2,3 will likely be reused



Example: Depth-wise Loop Compression

- Consider `bubblesort` program
- We discover the whole triangle by exploring the first iteration of the outer loop
- The number of inner loop's iterations being explored is linear
- This separates us from other loop unrolling techniques



Experiments

Benchmark	Size Parameter (SP)	Actual WCET	Complexity (wrt. SP)	Symbolic Simulation (SS) States	Time (ms)	WCET	Exact? Manual	Auto
bubblesort	n = 25	1648	$O(n^2)$	135	233	1648	Y	N
	n = 50	6423		260	701	6423	Y	N
	n = 100	25348		510	2438	25348	Y	N
expint	NA	859	-	519	8247	859	Y	Y
fft1	n = 8	181	$O(nlogn)$	111	446	181	Y	Y
	n = 16	379		176	927	379	Y	Y
	n = 32	791		287	2197	791	Y	Y
	n = 64	1661		495	6818	1661	Y	Y
fir	NA	760	-	108	387	760	Y	Y
insertsort	n = 25	1120	$O(n^2)$	159	387	1120	Y	N
	n = 50	4120		309	1504	4120	Y	N
	n = 100	15745		609	7542	15745	Y	N
j_complex	NA	133	-	165	491	534	N	N
ns	n = 5	2655	$O(n^4)$	63	59	2655	Y	Y
	n = 10	35555		103	116	35555	Y	Y
	n = 20	522105		183	344	522105	Y	Y
nsichneu	NA	281	-	334	15542	281	Y	N
ud	NA	819	-	487	1137	819	Y	Y
amortized	n = 50	394	$O(n)$	95	287	394	Y	Y
	n = 100	792		186	1035	792	Y	Y
	n = 200	1590		339	4057	1590	Y	Y
two_shapes	n = 50	2199	$O(n^2)$	259	797	2199	Y	Y
	n = 100	8149		509	3235	8149	Y	Y
	n = 200	31299		1009	19839	31299	Y	Y
non_deter	n = 25	3904	$O(n^2)$	129	509	3904	Y	Y
	n = 50	15304		242	1876	15304	Y	Y
	n = 100	60604		467	9253	60604	Y	Y
tcas	NA	99	-	6020	15925	99	Y	Y

The Need for Assertions

- Path-sensitivity is necessary for precision
- Incorporation of user information is crucial too

```
t = c = c1 = 0;
for (i = 0; i < 100; i++) {
    c++;
    if (A[i] != 0) {
        c1++;
        t += 1000;
    } else { t += 1; }
}
assert(c1 <= c / 10);
```

The Need for Assertions

- Consider memory high watermark analysis
- We cannot automatically reason about the external function parity

```
c1 = c2 = 0;
m = 0; m = m + 10;
for (i = 0; i < 100; i++) {
    if (parity(n)) {
        c1++; m = m + 10;
    } else { c2++; m = m - 10; }
    n++;
    assert(|c1 - c2| <= 1);
}
```

The Need for Local Assertions

- Consider `bubblesort`, input `a[]` contains element in $[min, max]$
- User information: there are M elements equal to max
- Local assertion is easier to derive (counter `c` is reset at the beginning of the inner loop)
- IPET does not support local assertions

```
for (i = N-1; i >= 1; i--) {
    c = 0;
    for (j = 0; j <= i-1; j++)
        if (a[j] > a[j+1]) {
            c++; t += 100; tp = a[j];
            a[j] = a[j+1]; a[j+1] = tp;
        } else { t += 1; }
        assert(c <= N-M);
    }}
```

Loop Unrolling and Assertions Don't Mix

- To tighten the bound, users need to provide only information about c

```
c = 0, i = 0, t = 0;
while (i < 9) {
    if (*) {B1:  c++; t += 10; }
    else {
        if (i == 1) {B2:  t += 5; }
        else {B3:  t += 1; }
    }
    i++;
    assert(c <= 4);
}
```

Loop Unrolling and Assertions Don't Mix

- Apply loop unrolling in previous section, performing the merge at the end of each loop iteration
  - Information about c is lost
  - The provided assertion will never be fired
  - Worst-case bound: 90
- Try greedy (under-approximation) by keeping the context of c from the worst-case path
  - Worst-case bound: $10 + 10 + 10 + 10 + 1 + 1 + 1 + 1 + 1 = 45$
  - This bound is unsound
  - Counter example:
    - Replace "`if (*)`" with "`if prime(i)`"
    - The timing: $1 + 5 + 10 + 10 + 1 + 10 + 1 + 10 + 1 = 49$

Loop Unrolling and Assertions Don't Mix

- Fundamental Issues
  - "Being compliant with assertions" requires the analysis to be fully path-sensitive wrt. assertion variables
  - This interferes with greedy treatment of loop (merge & summarize)

Solution: A Two-Phase Algorithm (for each loop)

- Phase 1:
  - Perform loop unrolling with iteration abstraction
  - Eliminate two kinds of paths:
    - Infeasible paths (detected from path-sensitivity)
    - Dominated paths. (1) We track frequency variables which will be used later in some assertion. (2) For paths which modify the tracked variables *in the same way*, we keep the one whose resource usage *dominates* the rest
- Phase 2:
  - Disregard all paths *violating* the assertions
  - Employ a dynamic programming approach with interpolation Jaffar *et al.* [2008]

Example: Removal of Infeasible Paths

```
c = 0, i = 0, t = 0;
while (i < 9) {
    if (*) {B1:  c++; t += 10; }
    else {
        if (i == 1) {B2:  t += 5; }
        else {B3:  t += 1; }
    }
    i++;
    assert(c <= 4);
}
```

- First iteration (eliminate the path executing **B2**):

$$\langle\langle 0\rangle, \mathtt{c} := \mathtt{c}+1 \wedge \mathtt{t} := \mathtt{t}+10, \langle 1\rangle\rangle$$
$$\langle\langle 0\rangle, \mathtt{t} := \mathtt{t}+1, \langle 1\rangle\rangle$$

- Second iteration (eliminate the path executing **B3**):

$$\langle\langle 1\rangle, \mathtt{c} := \mathtt{c}+1 \wedge \mathtt{t} := \mathtt{t}+10, \langle 2\rangle\rangle$$
$$\langle\langle 1\rangle, \mathtt{t} := \mathtt{t}+5, \langle 2\rangle\rangle$$

- Other iterations, i.e., $i = 2..8$, reuse the analysis of the first iteration:

$$\langle\langle \mathtt{i}\rangle, \mathtt{c} := \mathtt{c}+1 \wedge \mathtt{t} := \mathtt{t}+10, \langle \mathtt{i+1}\rangle\rangle$$
$$\langle\langle \mathtt{i}\rangle, \mathtt{t} := \mathtt{t}+1, \langle \mathtt{i+1}\rangle\rangle$$

Example: Removal of Dominated Paths

```
c = 0, i = 0, t = 0;
while (i < 9) {
    if (*) {B1: c++; t += 10; }
    else {
        if (*) {B2: t += 5; }
        else {B3: t += 1; }
    }
    i++;
    assert(c <= 4);
}
```

- All iterations, i.e., $i = 0..8$ (eliminate the path executing **B3**):

$$\langle\langle \mathtt{i}\rangle, \mathrm{c} := \mathrm{c}{+}1 \wedge \mathrm{t} := \mathrm{t}{+}10, \langle \mathtt{i{+}1}\rangle\rangle$$
$$\langle\langle \mathtt{i}\rangle, \mathrm{t} := \mathrm{t}{+}5, \langle \mathtt{i{+}1}\rangle\rangle$$

Experiments

Benchmark	LOC	Path-Sensitive (Symbolic execution w. loop unrolling)				Path-Insensitive (IPET)	
		w.o. Assertions		w. Assertions		w.o. As	w. As
		Bound	T(s)	Bound	T(s)		
sparse_array	< 100	110404	1.50	10404	3.48	110404	10404
bubblesort100	< 100	515398	5.52	**49798**	11.45	1019902	1019902
watermark	< 100	1010	1.74	**20**	5.45	*	*
conflict100	< 100	1504	3.47	**759**	9.22	1504	1129
insertsort100	< 100	515794	4.91	**30802**	7.78	1020804	1020804
crc	128	1404	7.73	1084	8.61	1404	1084
expint	157	15709	4.40	**859**	4.56	-	-
matmult100	163	3080505	4.55	131705	5.54	3080505	131705
fir	276	1129	2.35	**793**	2.39	-	-
fft64	219	7933	5.52	**1733**	6.04	-	-
tcas	400	159	3.84	**81**	3.9	172	94
statemate	1276	2103	9.65	**1103**	9.73	2271	1271
nsichneu_small	2334	483	9.43	**383**	9.51	2559	2459

Safety Verification of Concurrent Systems

- Extremely hard because of state explosion problem
  - Exploration of all possible interleavings of concurrent events
  - Example: The execution of n concurrent events is investigated by exploring all $n!$ interleavings of these events
- Two prominent techniques for state space reduction: Partial Order Reduction (POR) and Symmetry Reduction
  - Little (or no) sensitivity wrt. the target safety property
    - Slicing to remove irrelevant events does not count
  - Hardly work with symbolic methods

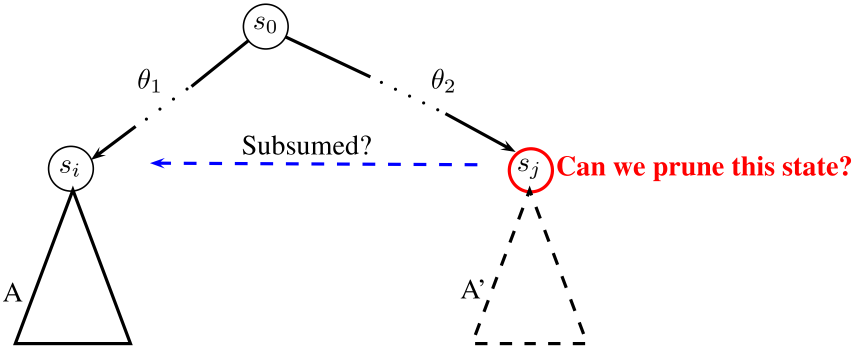
Traditional Partial Order Reduction (POR)

- Weaken the concept of a trace by abstracting the total order into a partial order
  - Two transitions are independent if their consecutive occurrences in a trace can be swapped without changing the final state
  - Two traces are equivalent if one can be transformed into another by repeatedly swapping adjacent independent transitions
  - For each class of equivalent traces, only one representative needs to be checked
- Distinguish two cases:
  - Deadlock verification
  - Safety verification (in general)

Our Contributions

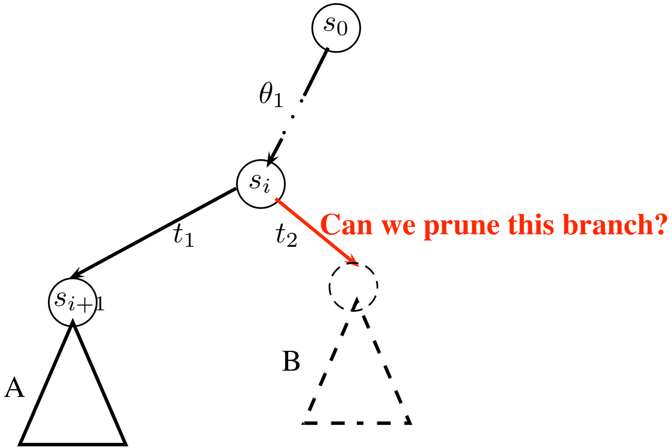
- Enable POR to work with symbolic search
- Synergize POR with State Interpolation (SI)
  - Replace the concept of trace equivalence with *trace coverage*
  - Weaken POR to Property Dependent POR (PDPOR)
  - Weaken PDPOR to Trace Interpolation

State Interpolation: State Pruning



s_i and s_j share the same program point ℓ

POR: Branch Pruning



t_1 and t_2 emanate from the same state s_i

Trace Coverage

Definition (Trace Coverage)

Let ρ_1, ρ_2 be two traces of a concurrent program. We say ρ_1 *covers* ρ_2 wrt. a safety property ψ, denoted as $\rho_1 \sqsupseteq_\psi \rho_2$, iff $\rho_1 \models \psi \rightarrow \rho_2 \models \psi$. □

- To replace the concept of trace equivalence
- The safety of one trace implies the safety of the other

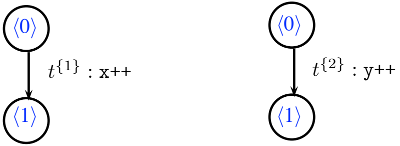
Property Dependent POR

Definition (Semi-Commutativity Relation)

Given a feasible derivation $s_0 \overset{\theta}{\Longrightarrow} s$, for all $t_1, t_2 \in \mathcal{T}$ which cannot dis-schedule each other, we say t_1 *semi-commutes* with t_2 *after state* s wrt. $\sqsupseteq_\psi$, denoted by $\langle s, t_1 \uparrow t_2, \psi \rangle$, iff for all $w_1, w_2 \in \mathcal{T}^*$, if $\theta w_1 t_1 t_2 w_2$ and $\theta w_1 t_2 t_1 w_2$ both are execution traces of the program, then we have $\theta w_1 t_1 t_2 w_2 \sqsupseteq_\psi \theta w_1 t_2 t_1 w_2$. □

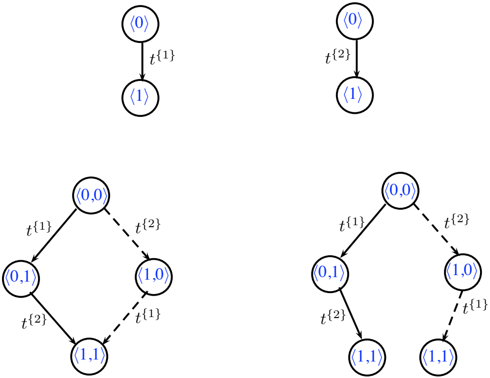
- To replace the concept of transition independence relation
- Traces with t_1 right before t_2 cover traces with t_1 right after t_2

Example: Independence vs. Semi-Commutativity



- $t^{\{1\}}$ is independent with $t^{\{2\}}$ wrt. deadlock verification
- $t^{\{1\}}$ is dependent with $t^{\{2\}}$ wrt. general safety property
- $t^{\{1\}}$ is semi-commutative with $t^{\{2\}}$ and vice versa wrt. safety property $\psi \equiv x + y \leq C$
- $t^{\{1\}}$ is semi-commutative with $t^{\{2\}}$ wrt. safety property $\psi \equiv x - y \leq C$, but not the other way around

Example: Independence vs. Semi-Commutativity



Property Dependent POR

Definition (New Persistent Set)

A set $T \subseteq \mathcal{T}$ of transitions enabled in a state s is *persistent in s* wrt. a property ψ iff, for all feasible derivation $s \xrightarrow{t_1} s_1 \xrightarrow{t_2} s_2 \ldots \xrightarrow{t_{m-1}} s_{m-1} \xrightarrow{t_m} s_m$ including only transitions $t_i \in \mathcal{T}$ and $t_i \notin T, 1 \leq i \leq m$, each transition in T *semi-commutes* with t_i after s wrt. $\sqsupseteq_\psi$. □

- Traces derived with transitions not in the persistent set first are covered by traces derived with transitions in the persistent set first

Property Dependent POR

- Selective search algorithm: at each state, we only consider transitions that belong to its persistent set

Theorem

The selective search algorithm with our new definition for persistent set is sound

- Given the semi-commutativity relation, to compute new persistent sets is similar to computing old persistent sets from the independence relation

Trace Interpolation

Definition (Semi-Commutative After A Program Point)

We say t_1 *semi-commutes* with t_2 *after program point* ℓ wrt. $\sqsupseteq_\psi$ and ϕ, denoted as $\langle \ell, \phi, t_1 \uparrow t_2, \psi \rangle$, iff for all feasible state $s \equiv \langle \ell, [\![s]\!] \rangle$ reachable from the initial state s_0, if $[\![s]\!] \models \phi$ then t_1 semi-commutes with t_2 after state s wrt. $\sqsupseteq_\psi$. □

Definition (Persistent Set Of A Program Point)

A set $T \subseteq \mathcal{T}$ of transitions schedulable at program point ℓ is *persistent at* ℓ under the trace-interpolant $\overline{\Psi}$ wrt. a property ψ iff, for all feasible derivation $s_0 \Longrightarrow s$ such that $s \equiv \langle \ell, [\![s]\!] \rangle$, if $[\![s]\!] \models \overline{\Psi}$ then for all feasible derivations $s \xrightarrow{t_1} s_1 \xrightarrow{t_2} s_2 \ldots \xrightarrow{t_{m-1}} s_{m-1} \xrightarrow{t_m} s_m$ including only transitions $t_i \in \mathcal{T}$ and $t_i \notin T, 1 \leq i \leq m$, each transition in T semi-commutes with t_i after state s wrt. $\sqsupseteq_\psi$. □

Implementing Trace Interpolation

- It is about approximating the semi-commutativity relation
  - Syntactic conditions (as in traditional POR)
  - Semantic conditions for some classes of problem and simple properties
  - General algorithm (opportunistically) when the weakest preconditions are available (on going)

Experiments: Producers and Consumer

`N producers increment x; N producers double x;`
`the consumer consumes value of x; prove x` $\leq N * 2^N$

	POR		SI		POR+SI		TI+SI	
N	States	T(s)	States	T(s)	States	T(s)	States	T(s)
2	449	0.03	514	0.17	85	0.03	10	0.01
3	18745	2.73	6562	2.43	455	0.19	14	0.01
4	986418	586.00	76546	37.53	2313	1.07	18	0.01
5	–	–	–	–	11275	5.76	22	0.01
6	–	–	–	–	53261	34.50	26	0.01
7	–	–	–	–	245775	315.42	30	0.01

Experiments: Sum of Ids

Comparing with the state-of-the-art

	POR = None		Kahlon *et al.* [2009] w. Z3			POR+SI = SI		TI+SI	
N	States	T(s)	Conflicts	Decisions	T(s)	States	T(s)	States	T(s)
6	2676	0.44	1608	1795	0.08	193	0.05	7	0.01
8	149920	28.28	54512	59267	10.88	1025	0.27	9	0.01
10	–	–	–	–	–	5121	1.52	11	0.01
12	–	–	–	–	–	24577	8.80	13	0.01
14	–	–	–	–	–	114689	67.7	15	0.01

Experiments: Dining Philosophers and Bakery

	None		POR		SI		POR+SI	
Problem	States	T(s)	States	T(s)	States	T(s)	States	T(s)
din-2(a)	22	0.01	22	0.01	21	0.01	21	0.01
din-3(a)	1773	0.10	646	0.05	153	0.03	125	0.02
din-4(a)	–	–	155037	19.48	1001	0.17	647	0.09
din-5(a)	–	–	–	–	6113	1.01	4313	0.54
din-6(a)	–	–	–	–	35713	22.54	24201	4.16
din-7(a)	–	–	–	–	202369	215.63	133161	59.69
bak-2	86	0.05	48	0.03	38	0.03	31	0.02
bak-3	1755	3.13	1003	1.85	264	0.42	227	0.35
bak-4	47331	248.31	27582	145.78	1924	5.88	1678	4.95
bak-5	–	–	–	–	14235	73.69	12722	63.60

- Method by Kahlon *et al.* [2009] also performs safety verification on DP with a simpler property: Our approach is about 3 times faster
- To disprove an unsafe property (b), we require only one trace (< 0.1 seconds) while they required a similar amount of time compared to (a)

Experiments: Concurrent Programs from Cordeiro and Fischer [2011]

Comparing with SMT-based context-bounded model checking

		Cordeiro and Fischer [2011]		SI		TI+SI	
Problem	LOC	C	T(s)	States	T(s)	States	T(s)
micro_2	247	17	1095	20201	10.88	201	0.04
stack	105	12	225	529	0.26	529	0.26
circular_buffer	111	∞	477	29	0.03	29	0.03
stateful20	60	10	95	1681	1.13	41	0.01

Symmetry Reduction: Settings and Motivations

- Input concurrent system is defined parametrically
- The number of processes (n) is known
- The state space contains many symmetric subtrees
  - A subtree might have up to n! symmetric images
- For each class of symmetric subtrees, only one representative needs to be checked
- Contributions:
  - We introduce the notion of weak symmetry
  - Our symmetry detection and reduction are performed dynamically
  - We completely exploit weak symmetry

Preliminaries

- Given an n-process system, let
  - $\mathcal{I} = [1 \cdots n]$ denote its *indices*
  - *Sym* $\mathcal{I}$ denote the set of all permutations on $\mathcal{I}$
  - A permutation π acts on a formula F by simultaneously replacing each occurrence of index i by $\pi(i)$

Example

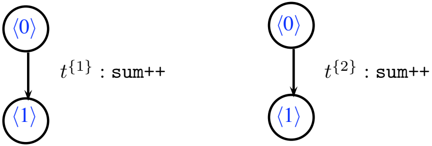
Let n $= 2$, $\pi = \{1 \mapsto 2, 2 \mapsto 1\}$

$\pi(id_1 < 3 \wedge id_2 > 4 \wedge x = 10) \equiv (id_2 < 3 \wedge id_1 > 4 \wedge x = 10)$

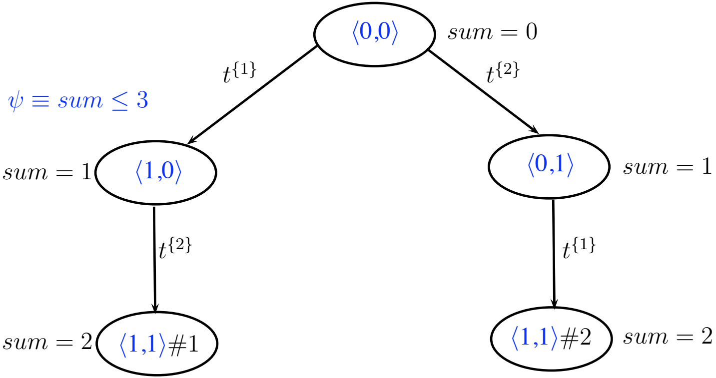
$\pi(id_2 = 2 \wedge x[id_1] = 5) \equiv (id_1 = 2 \wedge x[id_2] = 5)$

$\pi(id_2 = 2 \wedge x[2] = 5) \equiv (id_1 = 2 \wedge x[2] = 5)$

Example: Increment

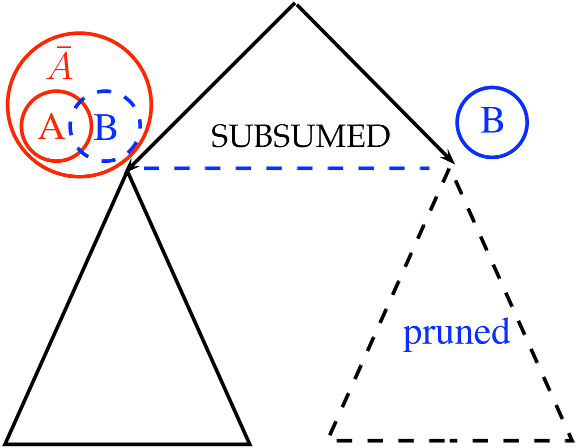


$Safety : \psi \equiv sum \leq 3$



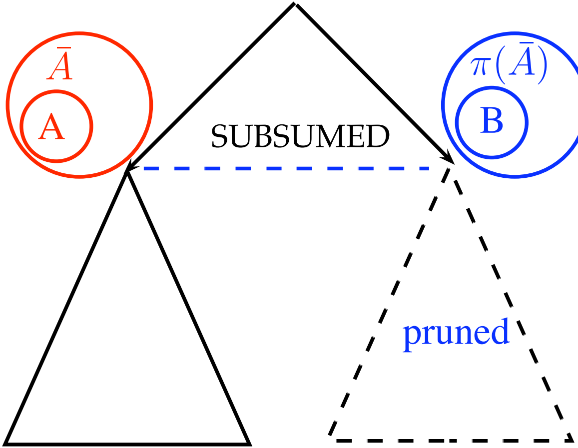
State Interpolation (recall)

A and B share the same program point



Pruning with Weak Symmetry

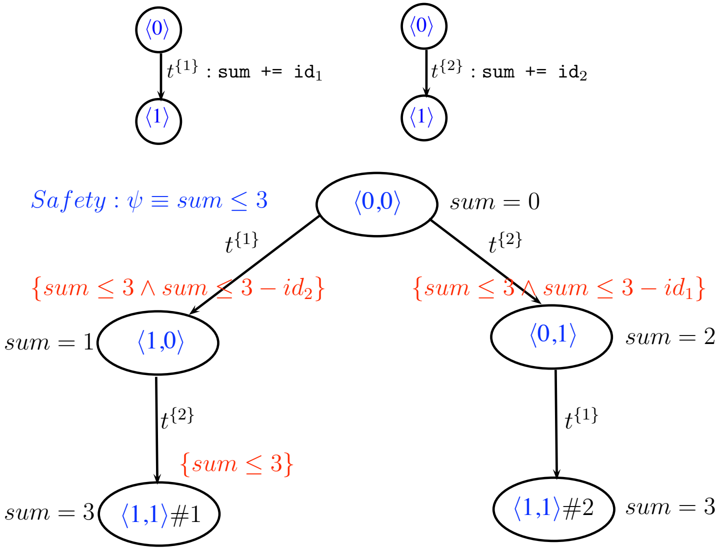
- A (program point ℓ_A) and B (program point ℓ_B) having $\pi(\ell_A) = \ell_B$ i.e., symmetric program points
- Generalize A to $\bar{A}$ while preserving safety, then apply π to $\bar{A}$



Our Language

- Allow the use of local variable `id`
  - `id` is initialized to a unique value in each process
  - for simplicity, `id` ranges from $1 \ldots n$
  - value of id can not be changed
- The behaviors of processes can range from *totally identical* to *arbitrarily divergent*

Example: Weak Symmetry



Complete Symmetry Reduction

- Completeness means that "given two states which are weakly symmetric, we will not explore them both in our search space"
- $\text{pre}(t, \phi)$ computes the precondition wrt postcondition ϕ and transition t

Definition

The precondition operator pre is said to be monotonic wrt. transition t if for all ϕ_1, ϕ_2 such that ϕ_1 is weaker than ϕ_2, we have $\text{pre}(t, \phi_1)$ is weaker than $\text{pre}(t, \phi_2)$

Theorem

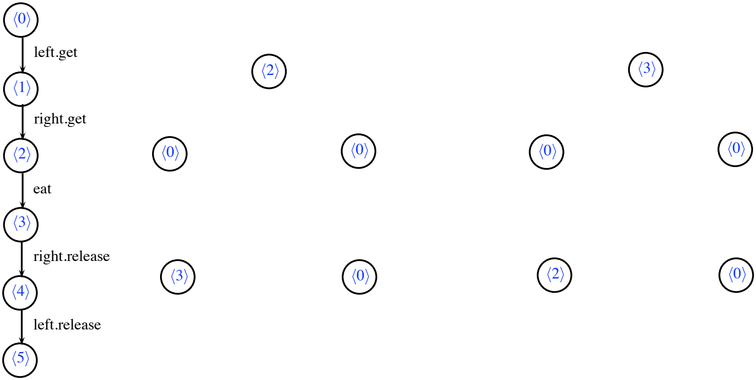
Our symmetry reduction is complete wrt. weak symmetry if our precondition operator is monotonic wrt. every transition

Experiments: Dining Philosophers

There are more symmetries than those statically known

	Complete Reduction			Rotational only			State Interpolation only		
# P	Visited	Subsumed	T(s)	V	S	T(s)	V	S	T(s)
4	230	134	0.09	328	184	0.13	1246	702	0.81
5	662	446	0.28	1509	981	0.71	7517	4893	4.93
6	1778	1304	0.85	7356	5216	4.18	43580	30908	34.53
7	4584	3552	2.55	35079	26335	28.83	–	–	–
8	11526	9281	7.54	–	–	–	–	–	–
9	28287	23432	22.6	–	–	–	–	–	–
10	67920	57504	58.07	–	–	–	–	–	–
11	159738	137609	226.86	–	–	–	–	–	–

Example: Dining Philosophers



Experiments: Reader-Writer Protocol

Comparing with the state-of-the-art

		Complete Reduction			Lazy Reduction (Wahl [2007])	
# R	# W	Visited	Subsumed	T(s)	Abstract States	T(s)
2	1	35	20	0.01	9	0.01
4	2	226	175	0.19	41	0.10
6	3	779	658	0.93	79	67.80
8	4	1987	1750	3.23	165	81969.00
10	5	4231	3820	9.21	–	–

Experiments: Sum of Ids

Weak symmetry

	Complete Reduction			SPIN-NSR		
# Processes	Visited	Subsumed	T(s)	Visited	Subsumed	T(s)
10	57	45	0.02	6146	4097	0.03
20	212	190	0.04	11534338	9437185	69.70
40	822	780	0.37	–	–	–
60	1832	1770	1.91	–	–	–
80	3242	3160	7.62	–	–	–
100	5052	4950	22.09	–	–	–

Experiments: Bakery

It is possible to work with infinite domain

	Complete Symmetry Reduction			State Interpolation		
# Processes	Visited	Subsumed	T(s)	Visited	Subsumed	T(s)
3	65	31	0.10	265	125	0.43
4	182	105	0.46	1925	1089	5.89
5	505	325	2.26	14236	9067	74.92
6	1423	983	11.10	–	–	–

Conclusion

- We proposed a path-sensitive analysis with efficient loop unrolling
  - Often achieved *exact* analysis
  - Reduced to *superlinear* complexity
  - Impactful as loop unrolling is performed in a wide range of analyses
- We extended our analysis to be compliant with (local) assertions
  - This enables the development of a system which possesses 3 key features: *accuracy*, *scalability*, and *usability*.
- We synergistically combined state-based and trace-based methods in safety verification of concurrent systems
- We weakened the traditional concept of symmetry and *completely* exploited it

Future Work

- Extend path-sensitivity to low-level analysis
  - Interpolation method for cache
- Use concurrency model and techniques to solve combinatorial optimization problems
  - Need to adapt the reduction techniques to analysis
  - Combine them with other well-known concepts in Constraint Programming (e.g., branch-and-bound, forward checking)

B. Beckert, R. Hähnle, and P. H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. 2007.

C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: Automatically Generating Inputs of Death. In *CCS*, 2006.

Lori A. Clarke. A System to Generate Test Data and Symbolically Execute Programs. *IEEE Trans. Software Eng.*, 1976.

L. Cordeiro and B. Fischer. Verifying multi-threaded software using smt-based context-bounded model checking. In *ICSE*, 2011.

E. A. Emerson and R. J. Trefler. From asymmetry to full symmetry: New techniques for symmetry reduction in model checking. In *Conference on Correct Hardware Design and Verification Methods*, 1999.

E. A. Emerson, J. W. Havlicek, and R. J. Trefler. Virtual symmetry reduction. In *LICS*, 2000.

J. Gustafsson, A. Ermedahl, and B. Lisper. Towards a flow analysis for embedded system C programs. In *WORDS*, 2005.

B. Jacobs and F. Piessens. The Verifast Program Verifier, 2008.

J. Jaffar, A. E. Santosa, and R. Voicu. Efficient memoization for dynamic programming with ad-hoc constraints. In *AAAI*, 2008.

J. Jaffar, A. E. Santosa, and R. Voicu. An interpolation method for CLP traversal. In *CP*, 2009.

V. Kahlon, C. Wang, and A. Gupta. Monotonic partial order reduction: An optimal symbolic partial order reduction technique. In *CAV*, 2009.

J. C. King. Symbolic Execution and Program Testing. *Com. ACM*, 1976.

Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *DAC*, 1995.

T. Lundqvist and P. Stenström. An integrated path and timing analysis method based on cycle-level symbolic execution. *RTS*, 1999.

A. P. Sistla and P. Godefroid. Symmetry and reduced symmetry in model checking. *ACM TOPLAS*, 2004.

T. Wahl. Adaptive symmetry reduction. In *CAV*, 2007.

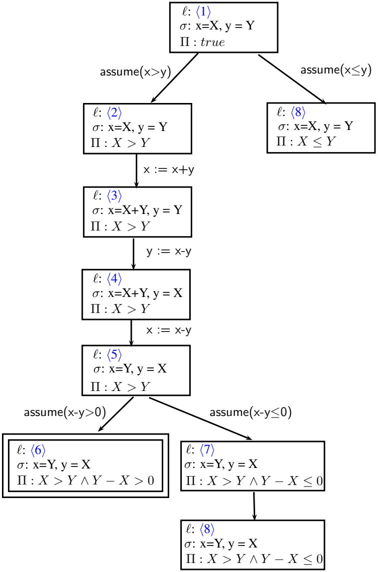
Questions & Answers

Example: Symbolic Execution

```
⟨1⟩ if (x > y) {
⟨2⟩     x = x + y;
⟨3⟩     y = x - y;
⟨4⟩     x = x - y;
⟨5⟩     if (x - y > 0) {
⟨6⟩         error();
        }
⟨7⟩
    }
⟨8⟩
```

$\langle\langle 1\rangle, \text{assume}(x > y), \langle 2\rangle\rangle$
$\langle\langle 2\rangle, x := x + y, \langle 3\rangle\rangle$
$\langle\langle 3\rangle, y := x - y, \langle 4\rangle\rangle$
$\langle\langle 4\rangle, x := x - y, \langle 5\rangle\rangle$
$\langle\langle 5\rangle, \text{assume}(x - y > 0), \langle 6\rangle\rangle$
$\langle\langle 5\rangle, \text{assume}(x - y \leq 0), \langle 7\rangle\rangle$
$\langle\langle 6\rangle, \text{void}, \langle 7\rangle\rangle$
$\langle\langle 7\rangle, \text{void}, \langle 8\rangle\rangle$
$\langle\langle 1\rangle, \text{assume}(x \leq y), \langle 8\rangle\rangle$

Example: Symbolic Execution



Trace Coverage

Definition (Equivalence)

Two traces are (Mazurkiewicz) *equivalent* iff one trace can be transformed into another by repeatedly swapping adjacent independent transitions. □

Definition (Trace Coverage)

Let ρ_1, ρ_2 be two traces of a concurrent program. We say ρ_1 *covers* ρ_2 wrt. a safety property ψ, denoted as $\rho_1 \sqsupseteq_\psi \rho_2$, iff $\rho_1 \models \psi \rightarrow \rho_2 \models \psi$. □

Property Dependent POR

Definition (Independence Relation)

$\mathcal{R} \subseteq \mathcal{T} \times \mathcal{T}$ is an *independence relation* iff for each $\langle t_1, t_2 \rangle \in \mathcal{R}$ the following properties hold for every state s:

1. if t_1 is enabled in s and $s \xrightarrow{t_1} s'$, then t_2 is enabled in s iff t_2 is enabled in s' ; and
2. if t_1 and t_2 are enabled in s, then there is a unique state s'' such that $s \overset{t_1 t_2}{\Longrightarrow} s''$ and $s \overset{t_2 t_1}{\Longrightarrow} s''$. □

Definition (Semi-Commutativity Relation)

Given a feasible derivation $s_0 \overset{\theta}{\Longrightarrow} s$, for all $t_1, t_2 \in \mathcal{T}$ which cannot dis-schedule each other, we say t_1 *semi-commutes* with t_2 *after state* s wrt. $\sqsupseteq_\psi$, denoted by $\langle s, t_1 \uparrow t_2, \psi \rangle$, iff for all $w_1, w_2 \in \mathcal{T}^*$, if $\theta w_1 t_1 t_2 w_2$ and $\theta w_1 t_2 t_1 w_2$ both are execution traces of the program, then we have $\theta w_1 t_1 t_2 w_2 \sqsupseteq_\psi \theta w_1 t_2 t_1 w_2$. □

Property Dependent POR

Definition (Old Persistent Set)

A set $T \subseteq \mathcal{T}$ of transitions enabled in a state s is *persistent in s* iff, for all feasible derivations $s \xrightarrow{t_1} s_1 \xrightarrow{t_2} s_2 \ldots \xrightarrow{t_{m-1}} s_{m-1} \xrightarrow{t_m} s_m$ including only transitions $t_i \in \mathcal{T}$ and $t_i \notin T$, $1 \leq i \leq m$, t_i is *independent* with all the transitions in T. □

Definition (New Persistent Set)

A set $T \subseteq \mathcal{T}$ of transitions enabled in a state s is *persistent in s* wrt. a property ψ iff, for all feasible derivation $s \xrightarrow{t_1} s_1 \xrightarrow{t_2} s_2 \ldots \xrightarrow{t_{m-1}} s_{m-1} \xrightarrow{t_m} s_m$ including only transitions $t_i \in \mathcal{T}$ and $t_i \notin T, 1 \leq i \leq m$, each transition in T *semi-commutes* with t_i after s wrt. $\sqsupseteq_\psi$. □

Traditional Symmetry Reductions

Definition (Strong Symmetry)

For $\pi \in \mathit{Sym}\ \mathcal{I}$, and a safety property ψ, for $s, s' \in \mathit{State}$, we say that s is strongly π-similar to s' wrt. ψ, denoted by $s \overset{\pi,\psi}{\approx} s'$ if ψ is symmetric wrt. π and the following conditions hold:

- $\pi(s) = s'$
- for each transition t such that $s \xrightarrow{t} d$ we have $s' \xrightarrow{\pi(t)} d'$ and $d \overset{\pi,\psi}{\approx} d'$
- for each transition t' such that $s' \xrightarrow{t'} d'$ we have $s \xrightarrow{\pi^{-1}(t')} d$ and $d \overset{\pi,\psi}{\approx} d'$.

- Rely on the fact that component processes are identical

State-of-the-art

- Traditional symmetry reduction methods exploit perfect symmetry, relying on the fact that all component processes are identical
- Emerson and Trefler [1999] considered near and rough symmetry, which later generalized to virtual symmetry (Emerson *et al.* [2000])
  - No implementation is provided
- Approaches by Sistla and Godefroid [2004] and Wahl [2007] are closest to us, in allowing behaviors of processes to range from totally identical to arbitrarily divergent
- All of them attempt to capture strong symmetry

Weak Symmetry

Definition (Weak Symmetry)

For $\pi \in Sym\ \mathcal{I}$, and a safety property ψ, for $s, s' \in State$, we say that s is weakly π-similar to s' wrt. ψ, denoted by $s \overset{\pi,\psi}{\sim} s'$ if ψ is symmetric wrt. π and the following conditions hold:

- $\pi(\mathrm{pc}(s)) = \mathrm{pc}(s')$
- $s \models \psi$ iff $s' \models \pi(\psi)$
- for each transition t such that $s \xrightarrow{t} d$ we have $s' \xrightarrow{\pi(t)} d'$ and $d \overset{\pi,\psi}{\sim} d'$
- for each transition t' such that $s' \xrightarrow{t'} d'$ we have $s \xrightarrow{\pi^{-1}(t')} d$ and $d \overset{\pi,\psi}{\sim} d'$.

Complete Symmetry Reduction

$$\boxed{\text{if (id == 2) \{ x[2] = 5; \}}}$$

- Problem of aliasing
  - Our method is still sound
  - It might affect the monotonicity of pre, hence the completeness reduction

Complete Symmetry Reduction

- Could be counter-productive if the system has little symmetry
- Optimization:
  - Quick test to avoid enumerate all the π. E.g., easy to see that there is no π such that $x = 2 \models \pi(x > 3)$
  - Let the users restrict the kind of symmetries to look for