

Improved Testing of Multithreaded Programs with Dynamic Symbolic Execution

Keijo Heljanko, Kari Kähkönen, Olli Saarikivi
(firstname.lastname@aalto.fi)

Department of Computer Science and Engineering,
Aalto University &
Helsinki Institute for Information Technology

Validation Methods for Concurrent Systems

There are many system validation approaches:

- Model based approaches:
 - Model-based Testing: Automatically generating tests for an implementation from a model of a concurrent system
 - Model Checking: Exhaustively exploring the behavior of a model of a concurrent system
 - Theorem proving, Abstraction, ...
- Source code analysis based approaches:
 - Automated test generation tools
 - Static analysis tools
 - Software model checking, Theorem Proving for source code, ...

Model Based vs. Source Code Based Approaches

- Model based approaches require building the verification model
 - In hardware design the model is your design
 - Usually not so for software:
 - Often a significant time effort is needed for building the system model
 - Making the cost-benefit argument is not easy for non-safety-critical software
- Source code analysis tools make model building cheap:
The tools build the model from source code as they go

The Automated Testing Problem

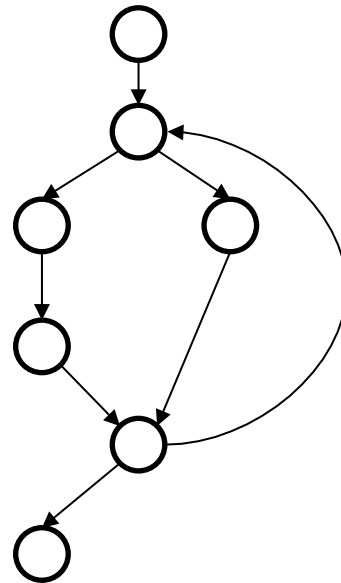
- How to automatically test the local state reachability in multithreaded programs that read input values
 - E.g., find assertion violations, uncaught exceptions, etc.
- Our tools use a subset of Java as its input language
- The main challenge: path explosion and numerous interleavings of threads
- One popular testing approach: dynamic symbolic execution (DSE) + partial order reduction
- New approach: DSE + unfoldings

Dynamic Symbolic Execution

- DSE aims to systematically explore different execution paths of the program under test

```
x = input
x = x + 5

if (x > 10) {
  ...
}
...
```



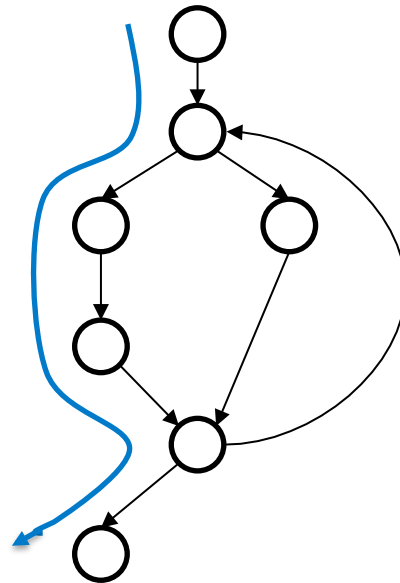
Control flow graph

Dynamic Symbolic Execution

- DSE typically starts with a random execution
- The program is executed concretely and symbolically

```
x = input
x = x + 5

if (x > 10) {
  ...
}
...
```

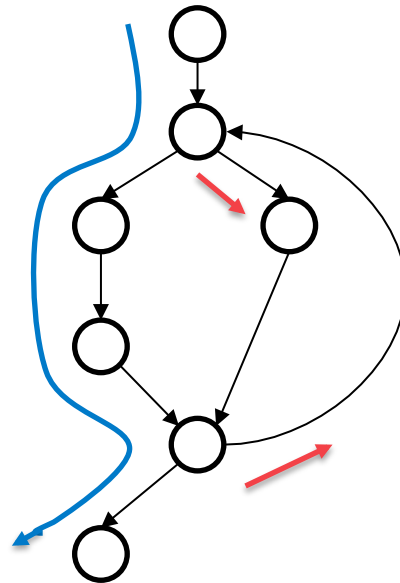


Control flow graph

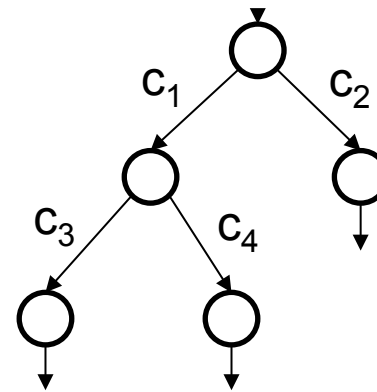
Dynamic Symbolic Execution

- Symbolic execution generates constraints at branch points that define input values leading to true and false branches

```
x = input
x = x + 5
if (x > 10) {
  ...
}
...
```



Control flow graph

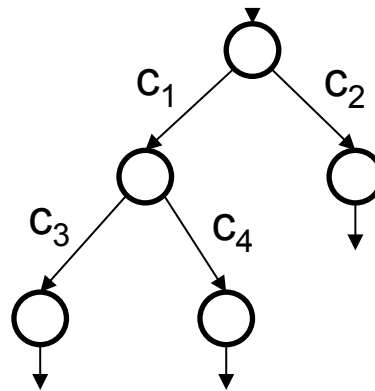


$$c_1 = \text{input}_1 + 5 > 10$$

$$c_2 = \text{input}_1 + 5 \leq 10$$

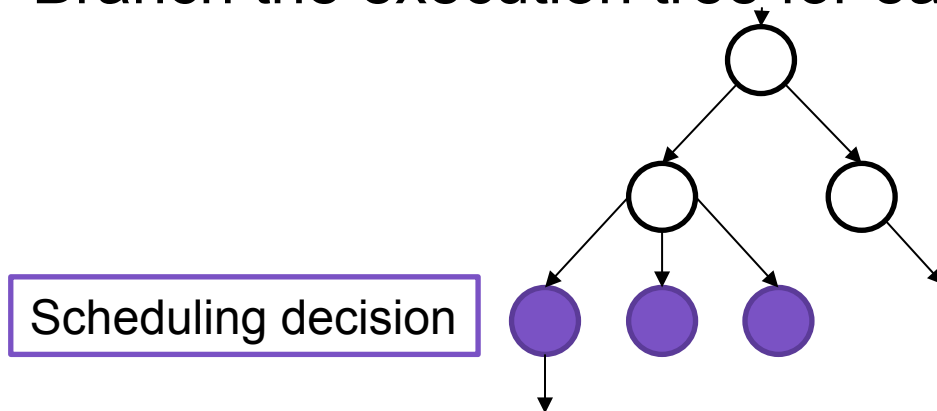
Dynamic Symbolic Execution

- A conjunction of symbolic constraints along an execution path is called a path constraint
 - Solved using SAT modulo theories (SMT)-solvers to obtain concrete test inputs for unexplored execution paths
 - E.g., pc: $\text{input}_1 + 5 > 10 \wedge \text{input}_2 * \text{input}_1 = 50$
 - Solution: $\text{input}_1 = 10$ and $\text{input}_2 = 5$



What about Multithreaded Programs?

- We need to be able to reconstruct scheduling scenarios
- Take full control of the scheduler
- Execute threads one by one until a global operation (e.g., access of shared variable or lock) is reached
- Branch the execution tree for each enabled operation



What about Multithreaded Programs?

- We need to be able to reconstruct scheduling scenarios
- Take full control of the scheduler
- Execute threads one by one until a global operation (e.g., access of shared variable or lock) is reached
- Branch the execution tree for each enabled operation

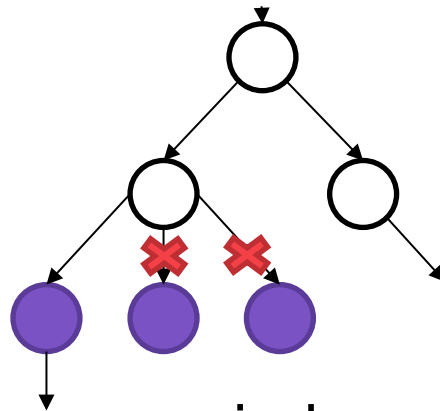


Problem: a large number of irrelevant interleavings

The diagram shows a state transition graph. At the top, a small circle has a self-loop arrow. Below it, a red rectangular box contains the text 'Problem: a large number of irrelevant interleavings'. Below the box, there are two white circles. The left white circle has three arrows pointing to three purple circles. The right white circle has one arrow pointing to the right. Each purple circle has a downward-pointing arrow.

One Solution: Partial-Order Reduction

- Ignore **provably irrelevant** parts of the symbolic execution tree



- Existing algorithms use independence of state transitions:
 - dynamic partial-order reduction (DPOR) [[FlaGod05](#)]
 - race detection and flipping [[SenAgh06](#)]

Independence of State Transitions

- There are several notions of independence that can be statically derived from the program source code:
 - Two operations in different processes on local data are independent
 - Two reads in different processes to the same global variable are independent with each other
 - Two writes in different processes to two different global variables are independent
 - Etc.

Independence of State Transitions

- Independence induces diamond structures in the state space as follows
- A pair t and u independent state transitions satisfy the following two properties for all sequences of state transitions w, w' of the system:
 1. If $w t u w'$ is an execution of the system, then so is $w u t w'$; and
 2. If $w t$ and $w u$ are executions of the system, then so are $w t u$ and $w u t$

Mazurkiewics Traces

- If we have an execution $w'' = w t u w'$ such that t and u are independent, then the execution $w u t w'$ will lead to the same final state
- If we take the union of all execution sequences that can be obtained from w'' by the transitive closure of repeatedly permuting any two adjacent independent state transitions we obtain the equivalence class called *Mazurkiewicz trace* [w'']
- All executions in the Mazurkiewicz trace are executable and will lead to the same final state – **testing one of them suffices!**
- If a partial order reduction method preserves one test from each Mazurkiewicz trace, it will also preserve all deadlocks

Dynamic Partial-Order Reduction (DPOR)

- DPOR algorithm by Flanagan and Godefroid (2005) calculates what additional interleavings need to be explored based on the history of the current execution
- Once DPOR has fully explored the subtree from a state it will have explored a persistent set of operations from that state
 - Will find all deadlocks and assertions on local states
- As any persistent set approach preserves one interleaving from each Mazurkiewicz trace



DPOR – Algorithm Intuition

- The DPOR algorithm does a depth-first traversal of the execution tree
- It tries to **detect races between state transitions**, e.g., a case when in a test run a global variable X is first written by process 1, that could be also concurrently read by process 2
- When race conditions are detected a **backtracking point** is added to try both outcomes of the race
- In the example, a backtracking point would be added just before write by process 1 to also explore the other interleaving where the read by process 2 happens first

Basic DPOR Pseudocode – Part 1

start: Explore (ϵ , $\lambda x.\perp$, $\lambda x.\perp$)

```
1 procedure Explore ( $E$ ,  $CP$ ,  $CE$ )
2    $s \leftarrow \text{last}(E)$ 
3   forall the processes  $p$  do
4      $v \leftarrow \text{next}(s, p)$ 
5     if  $\exists i = \max(\{i \in \{1 \dots |E|\} \mid E_i \text{ is dependent}$ 
        and may be co-enabled with  $v$  and
         $i \not\subseteq CP(p)(\text{proc}(E_i))\})$  then
6       if  $v \in \text{enabled}(\text{pre}(E, i))$  then
7         | add  $v$  to  $\text{backtrack}(\text{pre}(E, i))$ 
8       else
9         | add  $\text{enabled}(\text{pre}(E, i))$  to
          |  $\text{backtrack}(\text{pre}(E, i))$ 
10      end
11    end
12  end
```

- Looks for races:
Adds backtrack points for any potential races found

Basic DPOR Pseudocode – Part 2

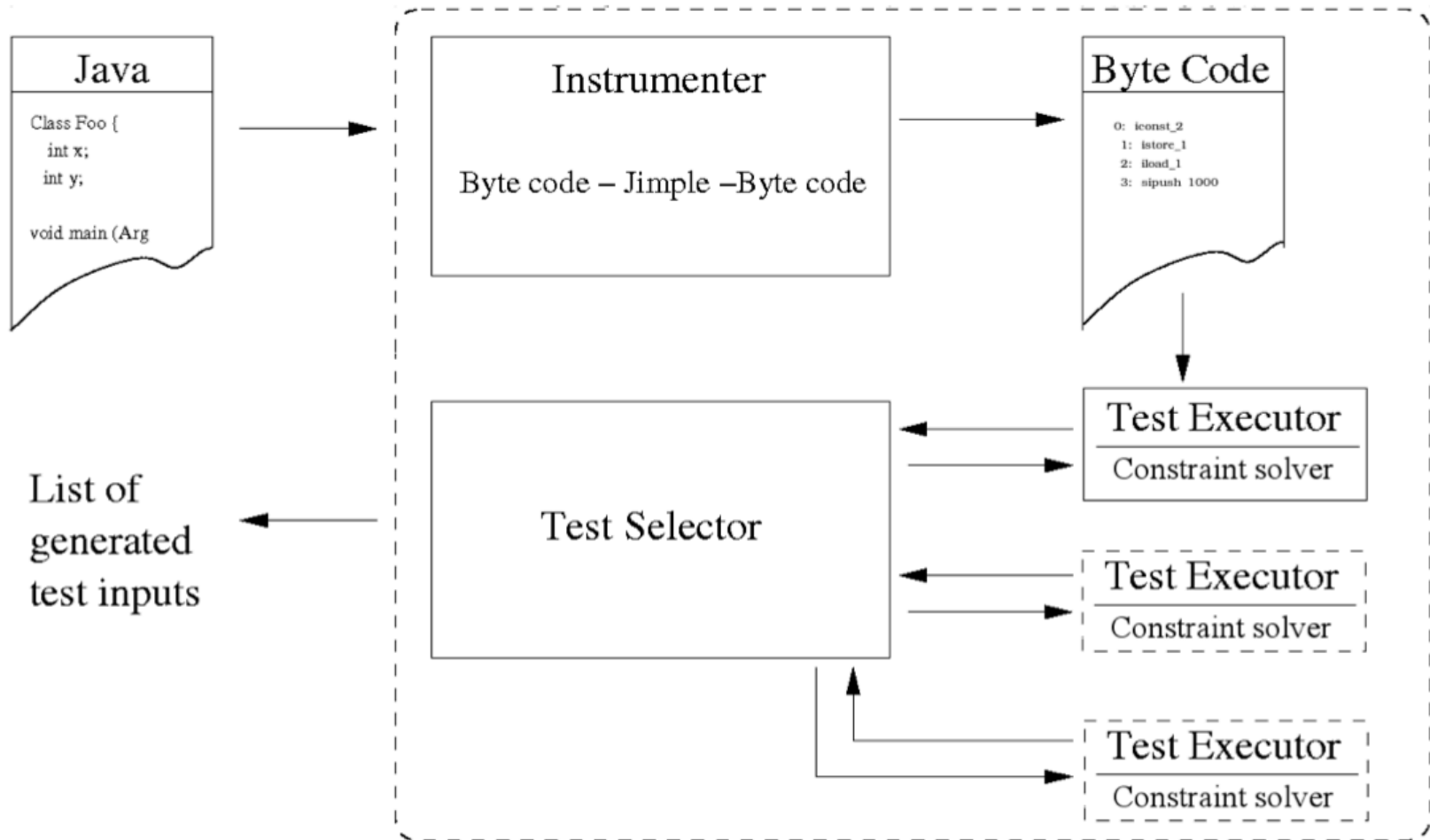
```
13  if  $\exists v_0 \in \text{enabled}(s)$  then
14       $\text{backtrack}(s) \leftarrow \{v_0\}$  // Initialize the
        backtrack set
15
16       $\text{done} \leftarrow \emptyset$ 
17      while  $\exists v_{\text{next}} \in (\text{backtrack}(s) \setminus \text{done})$  do
18          add  $v_{\text{next}}$  to  $\text{done}$ 
19           $E' \leftarrow E.v_{\text{next}}$ 
20           $cv \leftarrow \text{vmax}(\{CE(i) \mid i \in 1..|E| \text{ and } E_i$ 
        dependent with  $v_{\text{next}}\})$ 
21           $cv \leftarrow cv[\text{proc}(v_{\text{next}}) := |E'|]$ 
22           $CP' \leftarrow CP[\text{proc}(v_{\text{next}}) := cv]$ 
23           $CE' \leftarrow CE[|E'| := cv]$ 
24          Explore( $E', CP', CE'$ ) // Execute
        the visible operation  $v_{\text{next}}$ 
25
26      end
27  end
28 end
```

- Execute forward using backtrack sets
- Update vector clocks for race detection

Identifying Backtracking Points in DPOR

- To detect races, DPOR tracks the causal relationships of global operations in order to identify backtracking points
- In typical implementations the causal relationships are tracked by using vector clocks
- An optimized DPOR approach pseudocode can be found from:
 - Saarikivi, O., Kähkönen, K., and Heljanko, K.: Improving Dynamic Partial Order Reductions for Concolic Testing. ACSD 2012.
- Slideset: **Testing Multithreaded programs with DPOR**

Parallelizing DSE: LCT Architecture



Parallelization of LCT (PDMC'12)

- DSE+DPOR parallelizes excellently, LCT speedups from: Kähkönen, K., Saarikivi, O., and Heljanko, K.: [LCT: A Parallel Distributed Testing Tool for Multithreaded Java Programs](#), PDMC'12
- The experiments below have both DPOR and Sleep Sets enabled but LCT still achieves good speedups

Benchmark	Avg. paths	Avg. time	Avg. speedup			
	1 client		2 clients	5 clients	10 clients	20 clients ¹
Indexer (13)	671	285s	1.89	4.68	8.94	16.97
File System (18)	138	47s	1.92	4.55	8.88	14.91
Parallel Pi (5)	1252	250s	1.95	4.73	9.14	18.06
Synthetic 1 (3)	1020	176s	1.99	4.91	9.74	18.13
Synthetic 2 (3)	4496	783s	2.00	4.86	9.61	18.17

Sleep Sets

- *Sleep sets* were invented by Patrice Godefroid in:
 - P. Godefroid. [Using Partial Orders to Improve Automatic Verification Methods](#). In Proc. 2nd Workshop on Computer Aided Verification, LNCS 531, p. 176-185, 1990
- The algorithm maintains Sleep sets, which allow for a sound truncation of some of the execution tree branches
- [Sleep sets provide nice additional reduction on top of DPOR](#) and thus are an easy extra addition to it
- For details, see: Patrice Godefroid: [Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem](#). LNCS 1032, 1996

Sleep Set Pseudocode (P. Godefroid'96)

```
1 Initialize: Stack is empty; H is empty;
2     s0.Sleep = ∅;
3     push (s0) onto Stack;
4 Loop: while Stack ≠ ∅ do {
5     pop (s) from Stack;
6     if s is NOT already in H then {
7         enter s in H;
8         T = Persistent_Set(s) \ s.Sleep
9     }
10    else {
11        T = {t | t ∈ H(s).Sleep ∧ t ∉ s.Sleep};
12        s.Sleep = s.Sleep ∩ H(s).Sleep;
13        H(s).Sleep = s.Sleep
14    }
15    for all t in T do {
16        s' = succ(s) after t; /* t is executed */
17        s'.Sleep = {t' ∈ s.Sleep | (t, t') are independent in s };
18        push (s') onto Stack;
19        s.Sleep = s.Sleep ∪ {t}
20    }
21 }
```

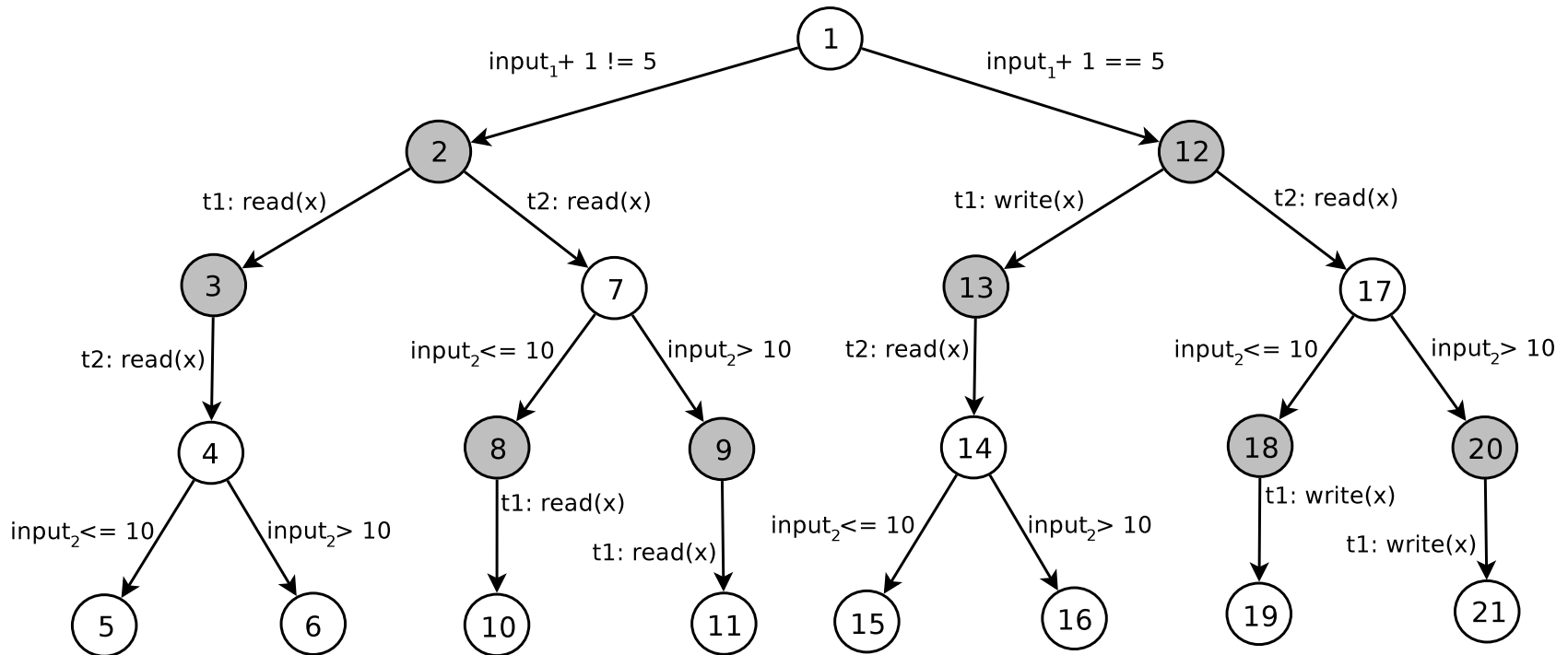
- Can be combined with persistent sets (e.g., DPOR)
- Ignore: Only for stateful search
- Fire transitions and update sleep sets

Example of Sleep Set POR Reduction

Global vars:	Thread 1:	Thread 2:
<code>X = 0;</code>	<code>1: i1 = input();</code>	<code>7: i2 = input();</code>
	<code>2: i1 = i1 + 1;</code>	<code>8: b = X;</code>
	<code>3: if (i1 == 5)</code>	<code>9: if (i2 > 10)</code>
	<code>4: X = 5;</code>	<code>10: b = 0;</code>
	<code>5: else</code>	
	<code>6: a = X;</code>	

Sleep Set Reduction Example

- At node 2 only one of the two independent read interleavings is needed
- The sleep set method is able to prune the subtrees rooted under node 7



DPOR is Search Order Dependent

Global variables:

$X = 0;$

$Y = 0;$

Thread 1:

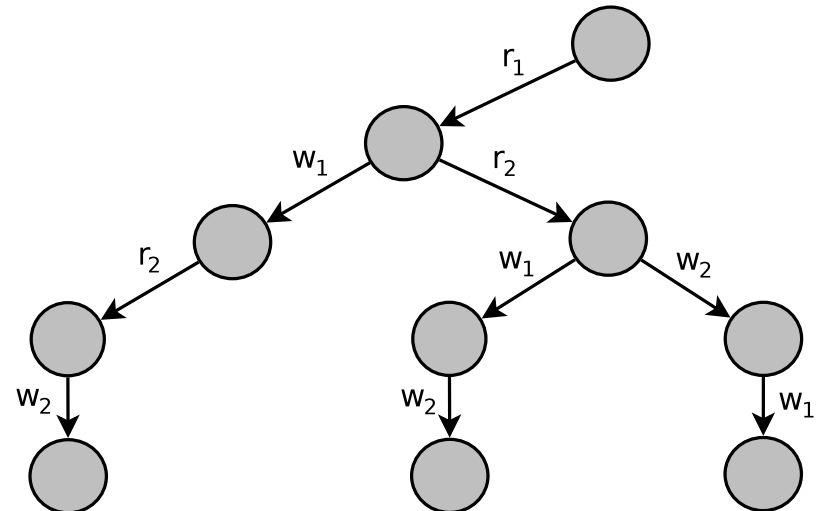
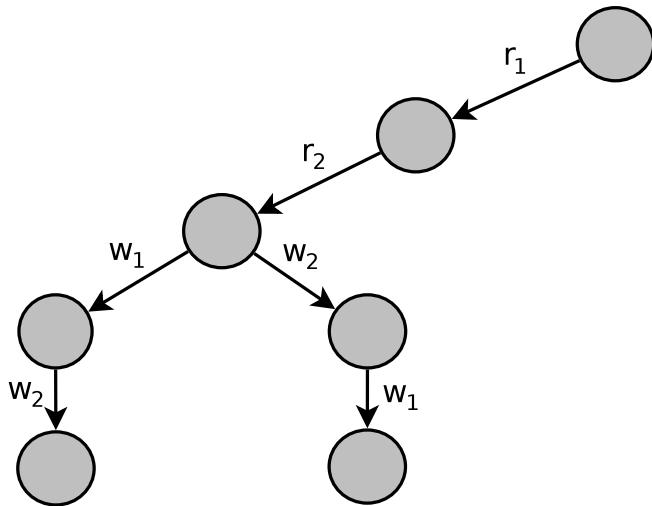
1: $a = X;$

2: $Y = 1;$

Thread 2:

3: $b = X;$

4: $Y = 2;$

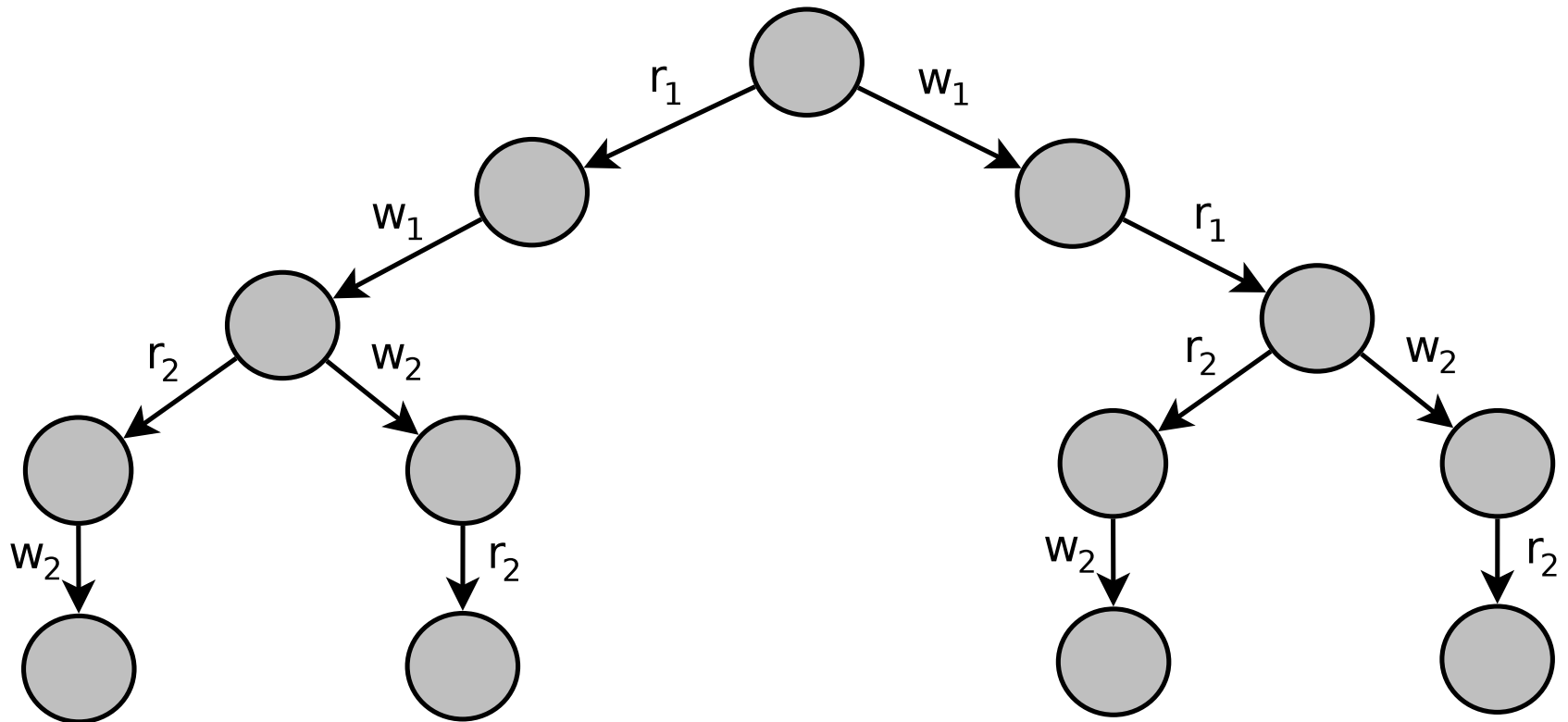


Family of Programs with Exponential DPOR Reduced Execution Trees

- Example: Add N variables and $2N$ threads: There will be 2^N test runs as there are 2^N deadlocks (Mazurkiewicz traces), and DPOR preserves one test run for each deadlock (Mazurkiewicz trace)

Global variables:	Thread 1:	Thread 2:
$X = 0;$	1: $a = X;$	2: $X = 1;$
$Y = 0;$		
	Thread 3:	Thread 4:
	3: $b = Y;$	4: $Y = 2;$

Exponentially Growing DPOR Example, $N=2$ with $2^N = 4$ test runs

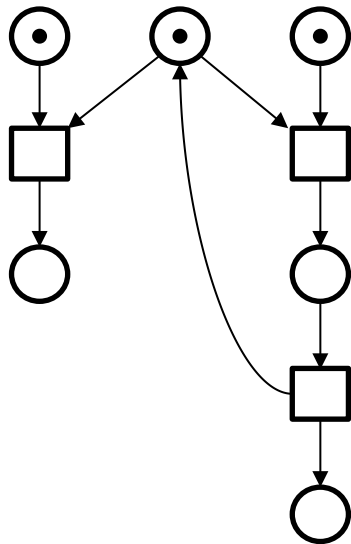


Another Solution?

- Can we create a symbolic representation of the executions that contain all the interleavings but in more compact form than with execution trees?
- **Yes, with Unfoldings**
- Originally created by Ken McMillan
- Book: Esparza, J. and Heljanko, K.: **Unfoldings - A Partial-Order Approach to Model Checking**. EATCS Monographs in Theoretical Computer Science, Springer, ISBN 978-3-540-77425-9, 172 p., 2008.
- <http://users.ics.aalto.fi/kepa/publications/Unfoldings-Esparza-Heljanko.pdf>

What Are Unfoldings?

- Unwinding of a control flow graph is an execution tree
- Unwinding of a Petri net (Java code) is an unfolding
- Can be exponentially more compact than exec. trees



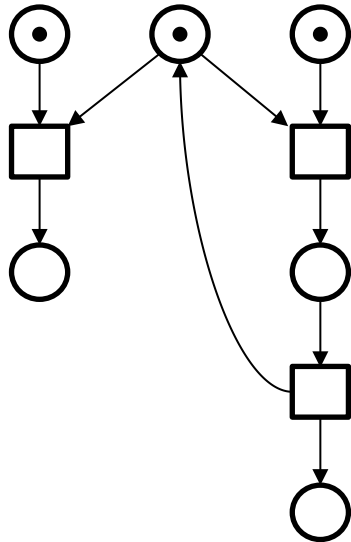
Petri net



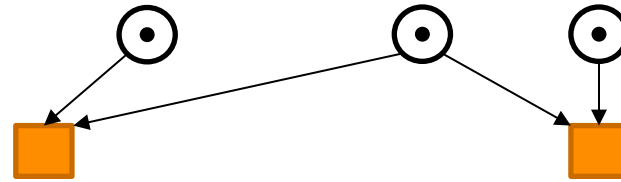
Initial unfolding

What Are Unfoldings?

- Unwinding of a control flow graph is an execution tree
- Unwinding of a Petri net is an unfolding
- Can be exponentially more compact than exec. trees



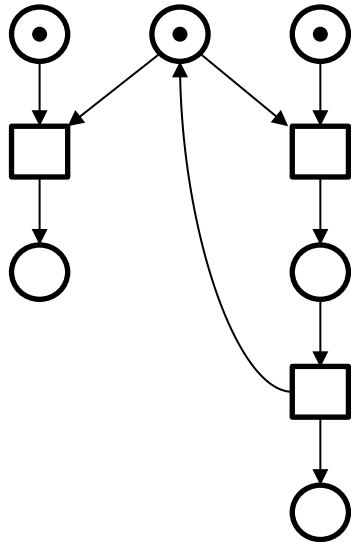
Petri net



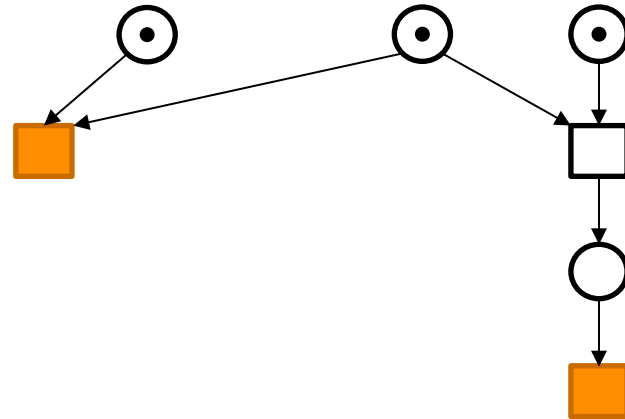
Unfolding

What Are Unfoldings?

- Unwinding of a control flow graph is an execution tree
- Unwinding of a Petri net is an unfolding
- Can be exponentially more compact than exec. trees



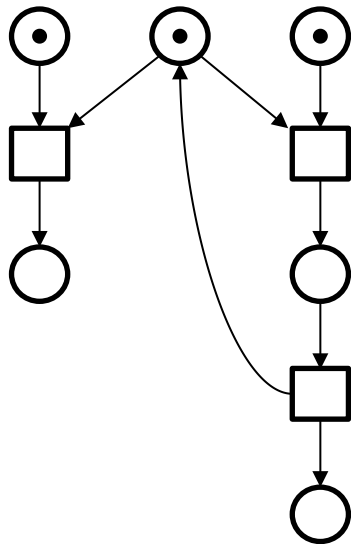
Petri net



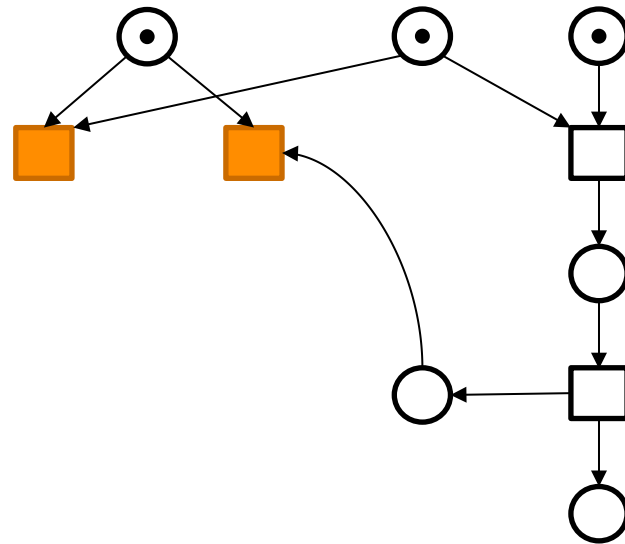
Unfolding

What Are Unfoldings?

- Unwinding of a control flow graph is an execution tree
- Unwinding of a Petri net is an unfolding
- Can be exponentially more compact than exec. trees



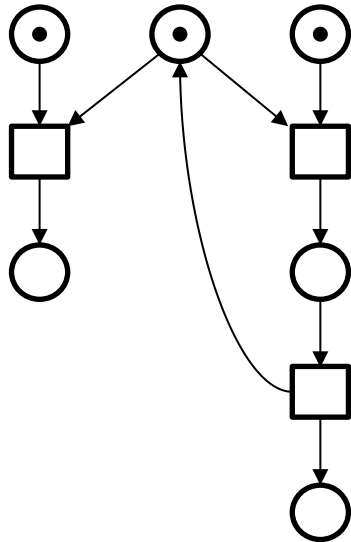
Petri net



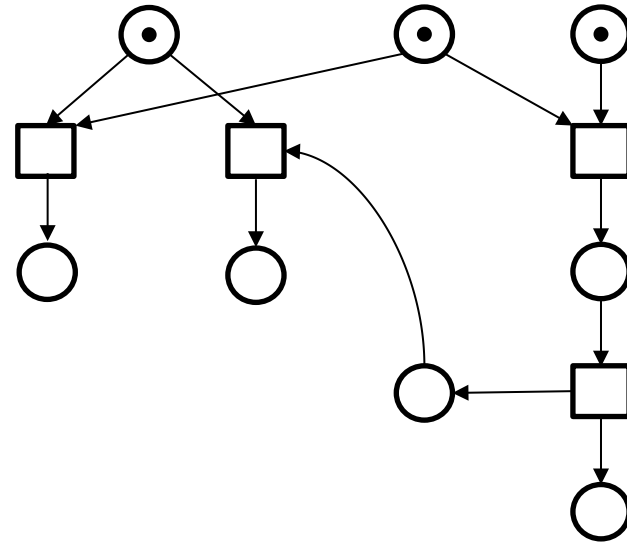
Unfolding

What Are Unfoldings?

- Unwinding of a control flow graph is an execution tree
- Unwinding of a Petri net is an unfolding
- Can be exponentially more compact than exec. trees

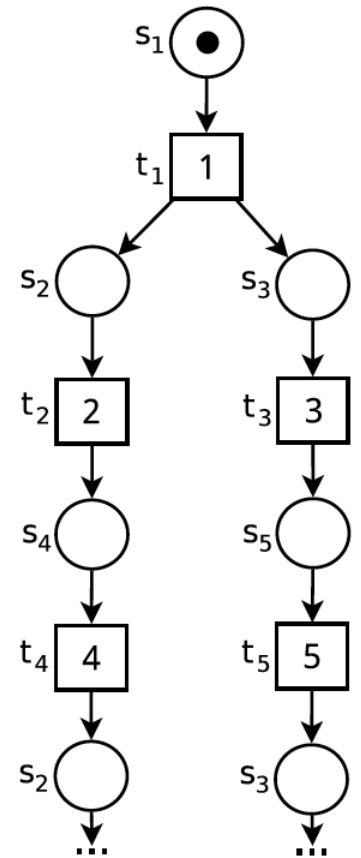
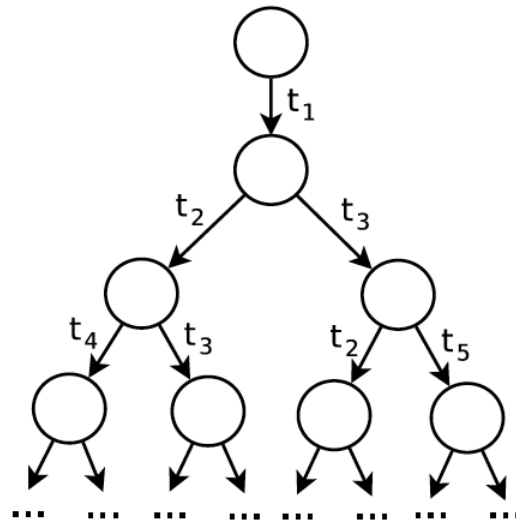
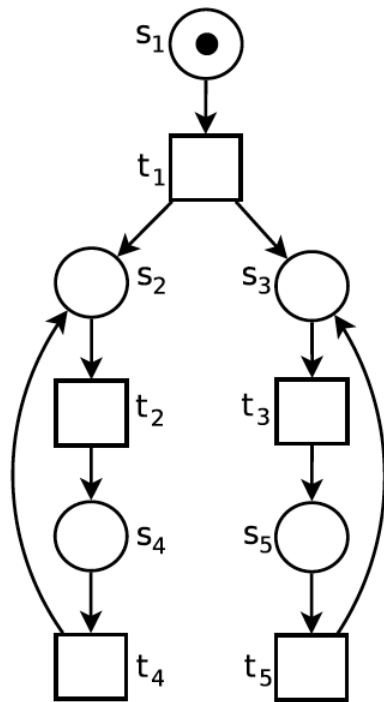


Petri net



Unfolding

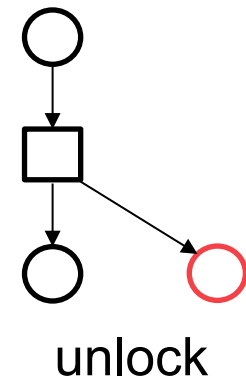
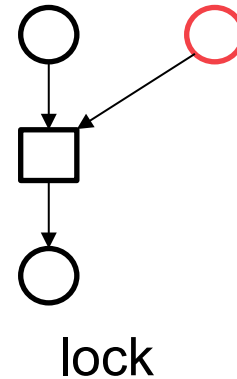
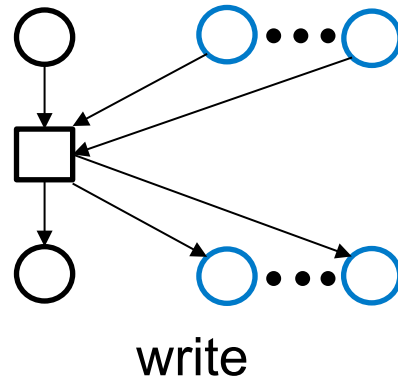
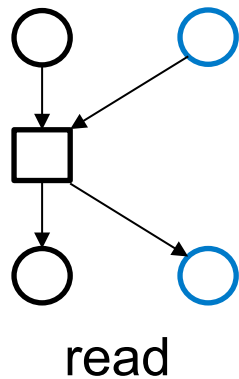
Example: Petri net, Execution Tree, Unfolding



- Note: Execution tree grows exponentially in the levels, unfolding grows only linearly

Using Unfoldings with DSE

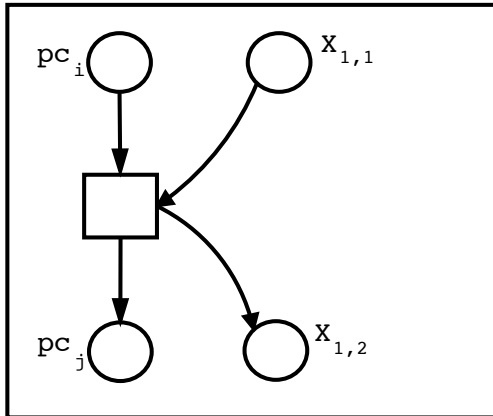
- When a test execution encounters a global operation, extend the unfolding with one of the following events:



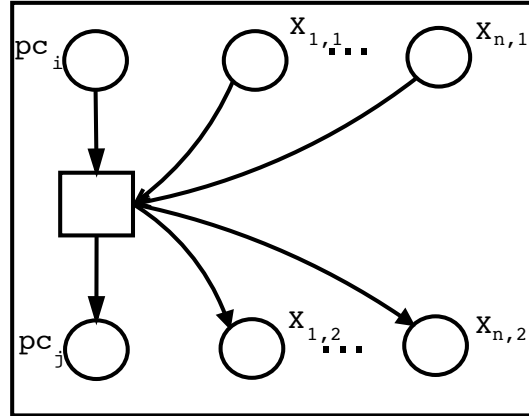
- Trick: **Place Replication** of global variables into N variables, one per each process – makes all reads independent of other reads

Shared Variables have Local Copies

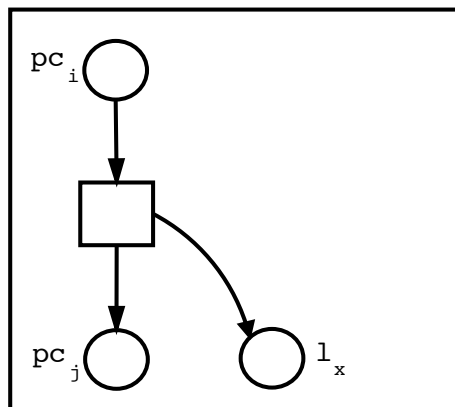
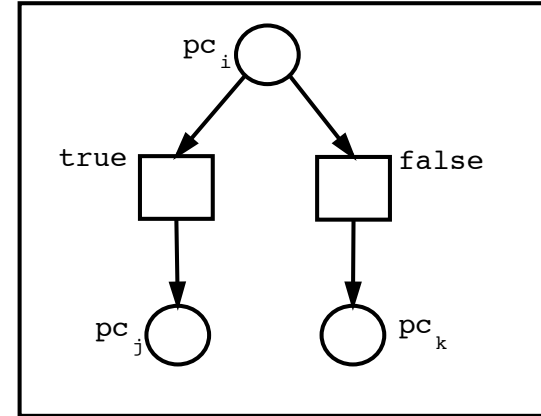
read global variable



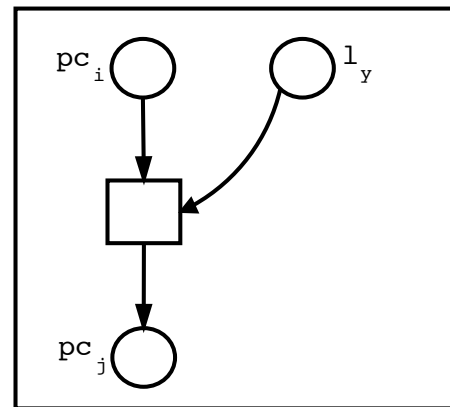
write global variable



symbolic branching



release lock l



acquire lock l

Write modifies
all copies of
the variable X

From Java Source Code to Unfoldings

- The unfolding shows the control and data flows possible in all different ways to solve races in the Java code
- **The underlying Petri net is never explicitly built in the tool**, we compute possible extensions on the Java code level
- **Our unfolding has no data in it** – The unfolding is an over-approximation of the possible concurrent executions of the Java code
- Once a potential extension has been selected to extend the unfolding, the **SMT solver is used to find data values** that lead to that branch being executed, if possible
- Branches that are non-feasible (due to data) are pruned

Example – Unfolding Shows Data Flows

Global variables:

```
int x = 0;
```

Thread 1:

```
local int a = x;
```

```
if (a > 0)
```

```
  error();
```

Thread 2:

```
local int b = x;
```

```
if (b == 0)
```

```
  x = input();
```



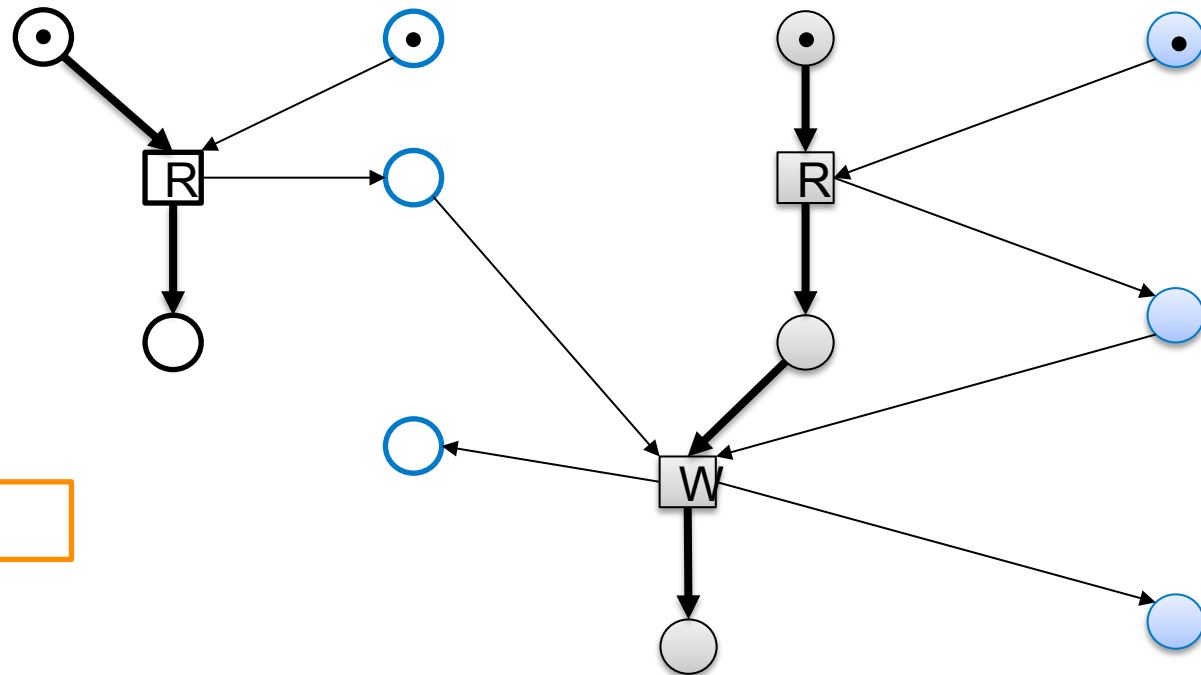
Initial unfolding

Example – Unfolding Shows Data Flows

Global variables:
int x = 0;

Thread 1:
local int a = x;
if (a > 0)
error();

Thread 2:
local int b = x;
if (b == 0)
x = input();



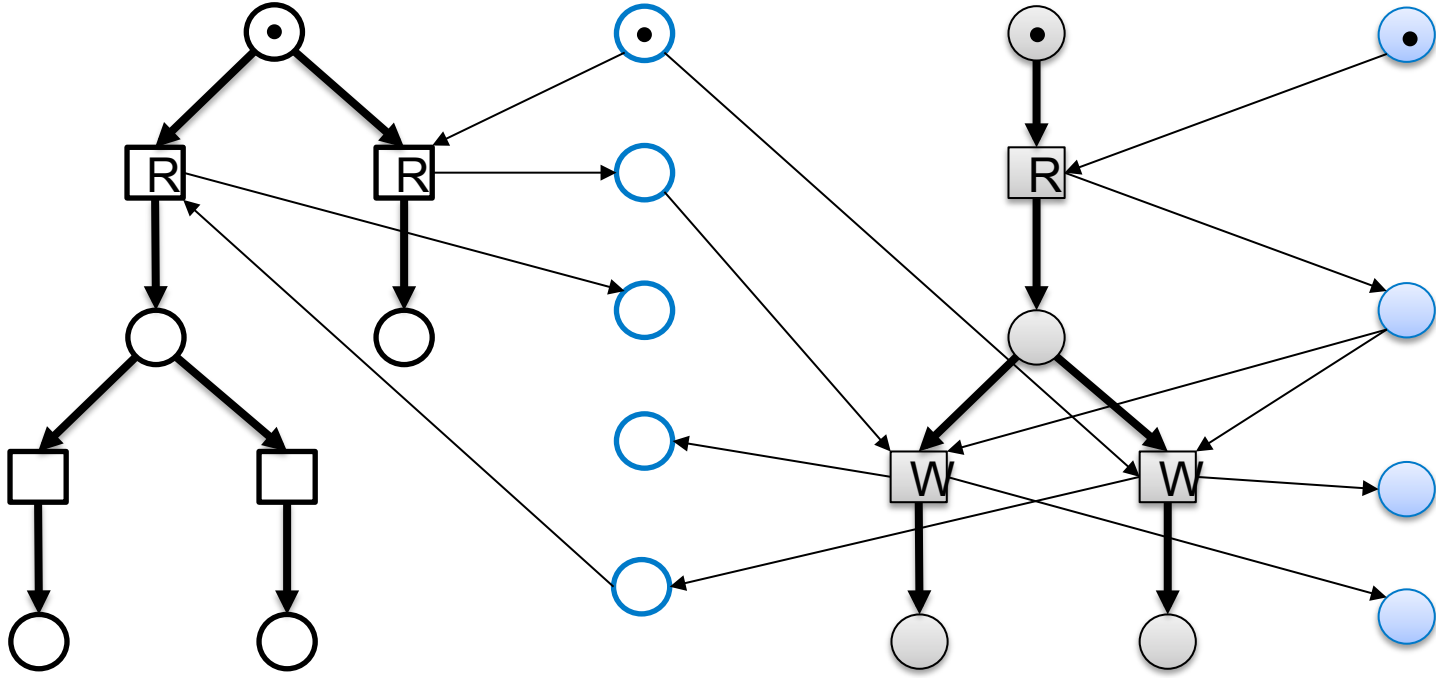
First test run

Example– Unfolding Shows Data Flows

Global variables:
int x = 0;

Thread 1:
local int a = x;
if (a > 0)
error();

Thread 2:
local int b = x;
if (b == 0)
x = input();



Another Example Program

Global vars:

```
X = 0;
```

Thread 1:

```
1: i1 = input();
```

```
2: i1 = i1 + 1;
```

```
3: if (i1 == 5)
```

```
4:     X = 5;
```

```
5: else
```

```
6:     a = X;
```

Thread 2:

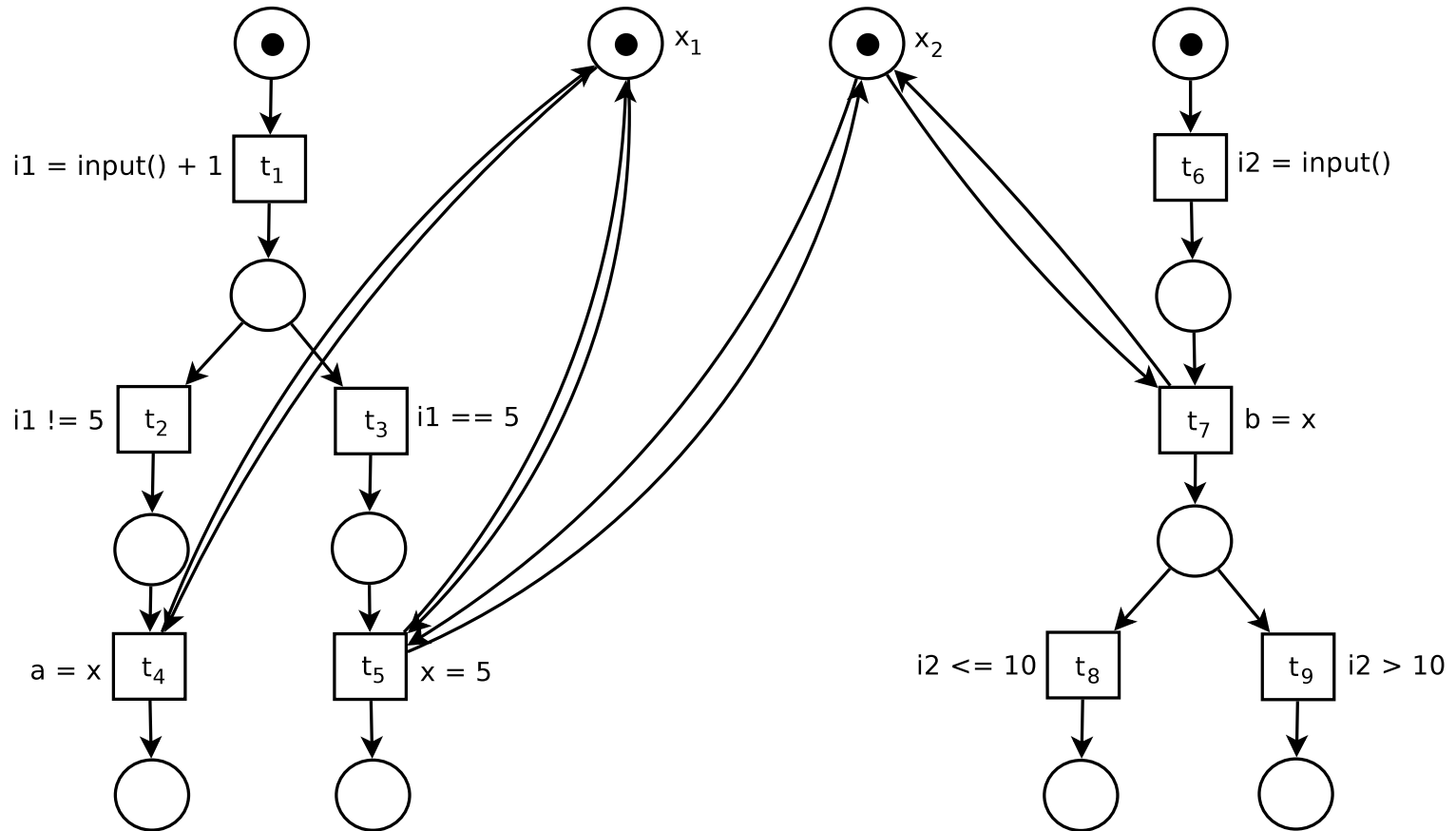
```
7: i2 = input();
```

```
8: b = X;
```

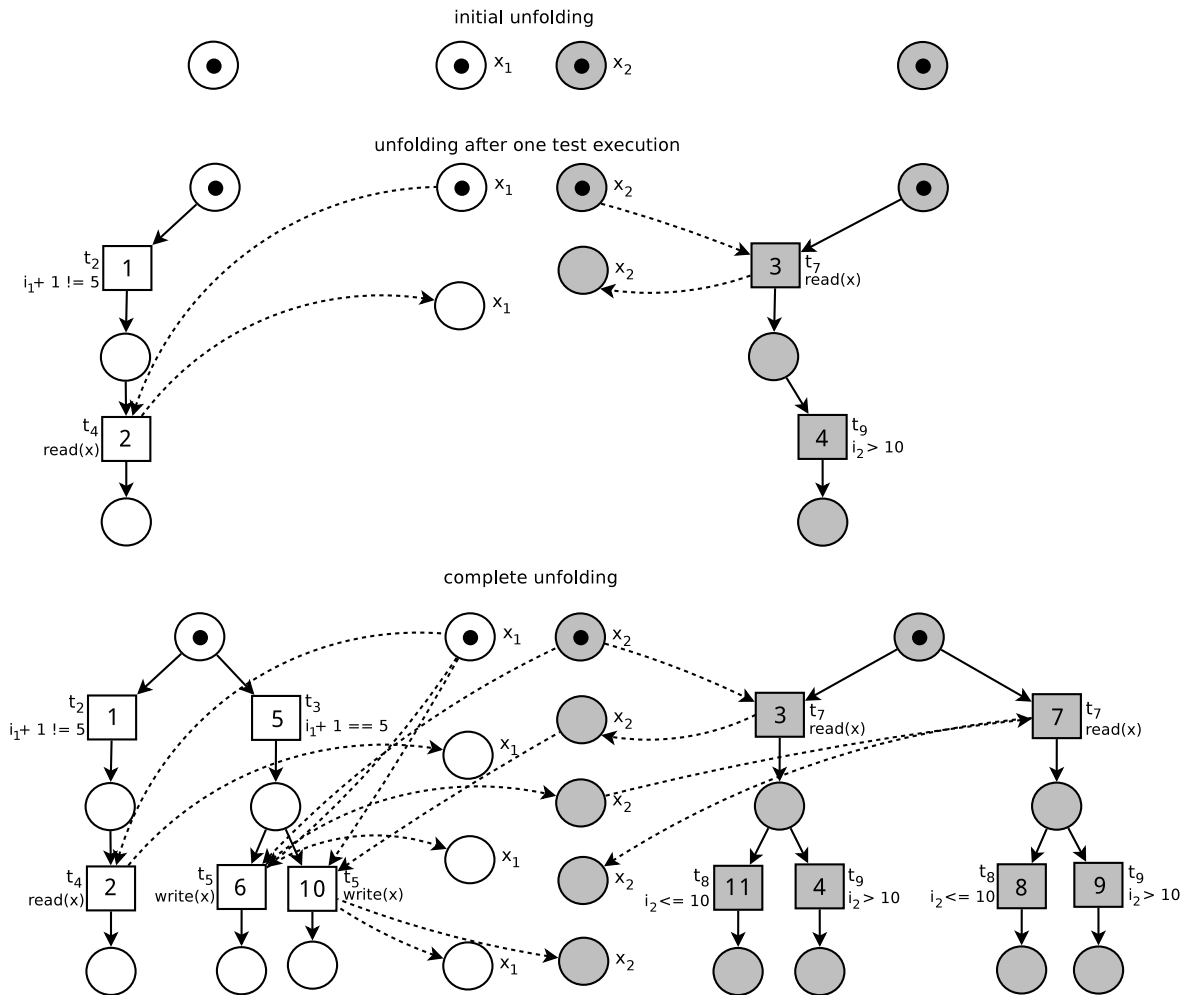
```
9: if (i2 > 10)
```

```
10:     b = 0;
```

Program Representation as Petri Net



Unfolding of the Program Representation



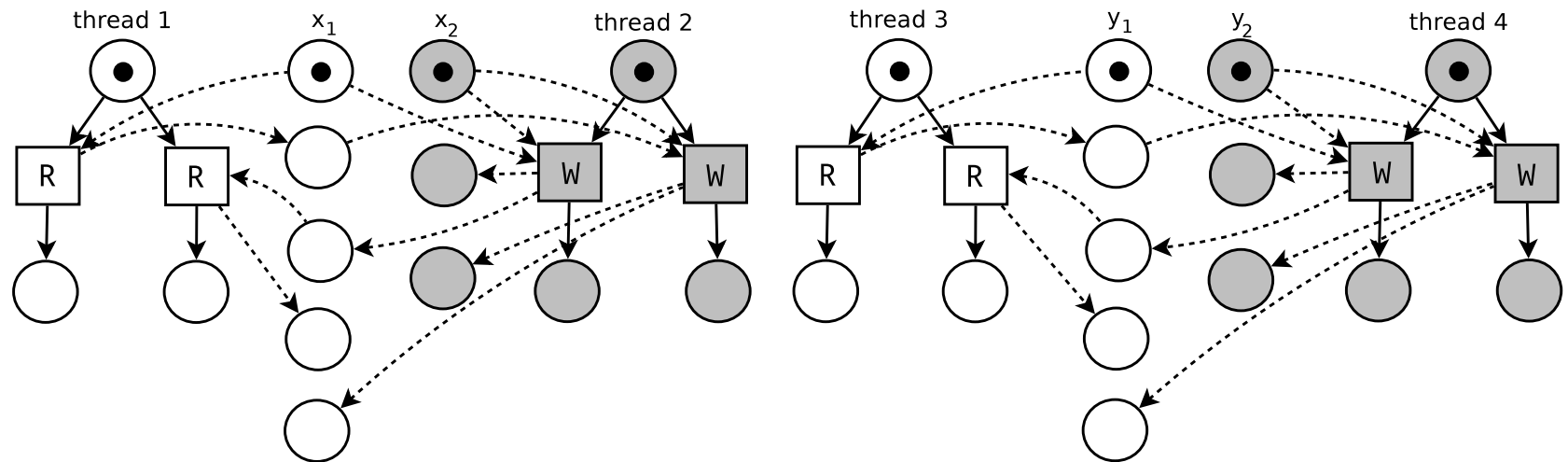
Recap: Family of Programs with Exponential DPOR Reduced Execution Trees

- Example: Add N variables and $2N$ threads: There will be 2^N test for any partial order reduction method preserving all Mazurkiewicz traces, e.g., DPOR, persistent, ample, stubborn, and sleep sets

Global variables:	Thread 1:	Thread 2:
$X = 0;$	1: $a = X;$	2: $X = 1;$
$Y = 0;$		
	Thread 3:	Thread 4:
	3: $b = Y;$	4: $Y = 2;$

Example Unfolding Grows Linear in N

- Unfolding of the example is $O(N)$, see below, while the DPOR reduced execution tree is $O(2^N)$



- Unfoldings will be exponentially more compact than any deadlock (Mazurkiewicz trace) preserving POR method!

What is Preserved by Unfoldings

- The unfolding of the previous example can be covered by two test runs
- The unfolding preserves the reachability of local states and executability of statements
- Thus asserts on local state can be checked
- Reachability of global states e.g., deadlocks, is symbolic in the unfolding – allows unfoldings to be smaller
- Reachability of a global state is present in the unfolding – it is a symbolic representation of the system behavior
- Checking any global state reachability question can be done in NP in the unfolding size using an SAT solver



Unfolding based Testing Algorithm

Input: A program P

- 1: $unf :=$ initial unfolding
- 2: $extensions :=$ events enabled in the initial state
- 3: **while** $extensions \neq \emptyset$ **do**
- 4: **choose** $target \in extensions$
- 5: **if** $target \notin unf$ **then**
- 6: $statement_sequence := EXECUTE(P, target, k)$
- 7: $M =$ initial marking
- 8: **for all** $stmt \in statement_sequence$ **do**
- 9: $e = CORRESPONDINGEVENT(stmt, M)$
- 10: **if** $e \notin unf$ **then**
- 11: add e and its output conditions to unf
- 12: $extensions := extensions \setminus \{e\}$
- 13: $pe := POSSIBLEEXTENSIONS(e, unf)$
- 14: $extensions := extensions \cup pe$
- 15: $M = FIREEVENT(e, M)$



Computing Possible Extensions

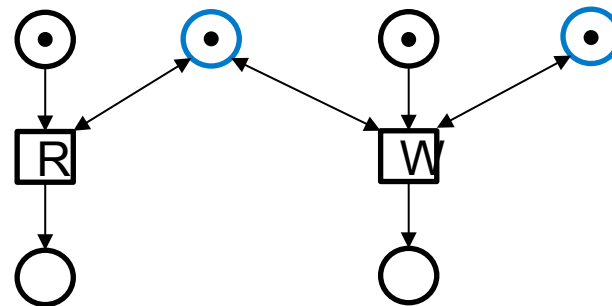
- Finding *possible extensions* is the most computationally expensive part of unfolding (**NP-complete** [Heljanko'99])
- It is possible to use existing backtracking search based potential extension algorithms with DSE
 - Designed for arbitrary Petri nets
 - Can be very expensive in practice
- Key observation: It is possible to limit the search space of potential extensions due to restricted form of unfoldings generated by the algorithm
 - Same worst case behavior, but in practice very efficient, see ASE'2012 paper for details

Specialized Algorithm for Possible Extensions

- In a Petri net representation of a program under test (not constructed explicitly in our algorithm) the places for shared variables are always marked
- This results in a tree like connection of the unfolded shared variable places and allows very efficient potential extension search, see [ASE'12]

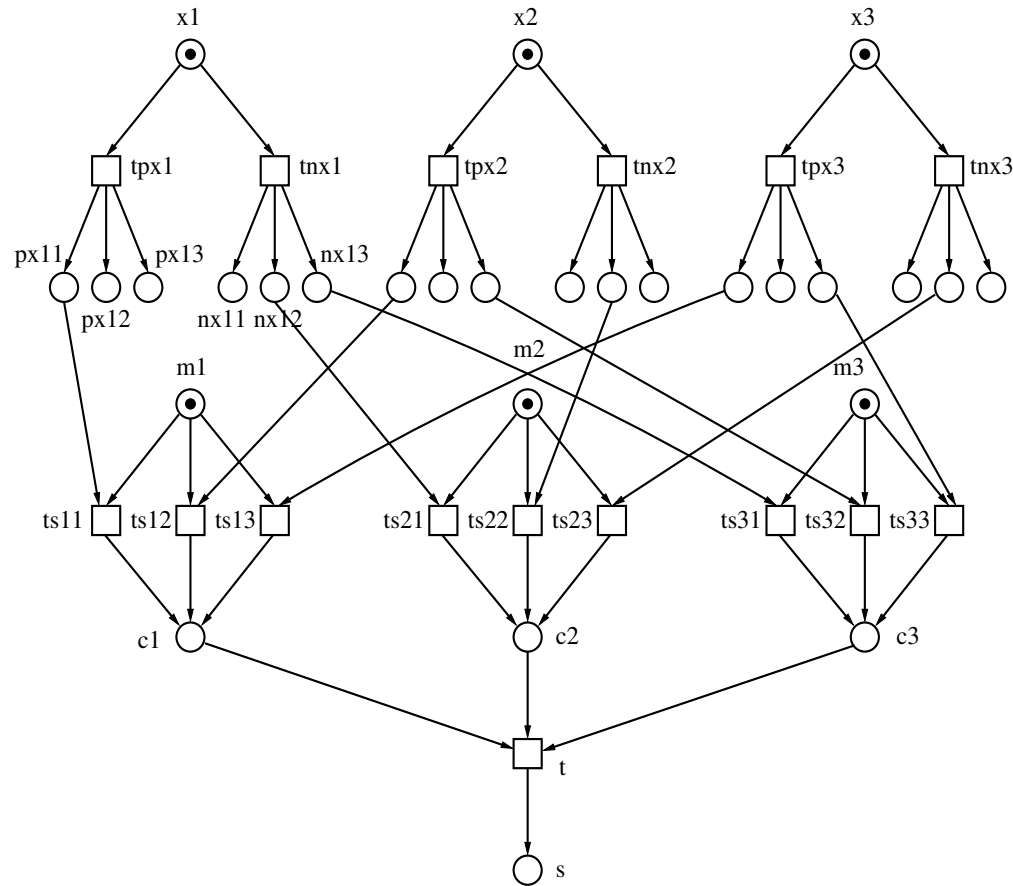
Thread 1:
local int a = x; (read)

Thread 2:
x = 5; (write)



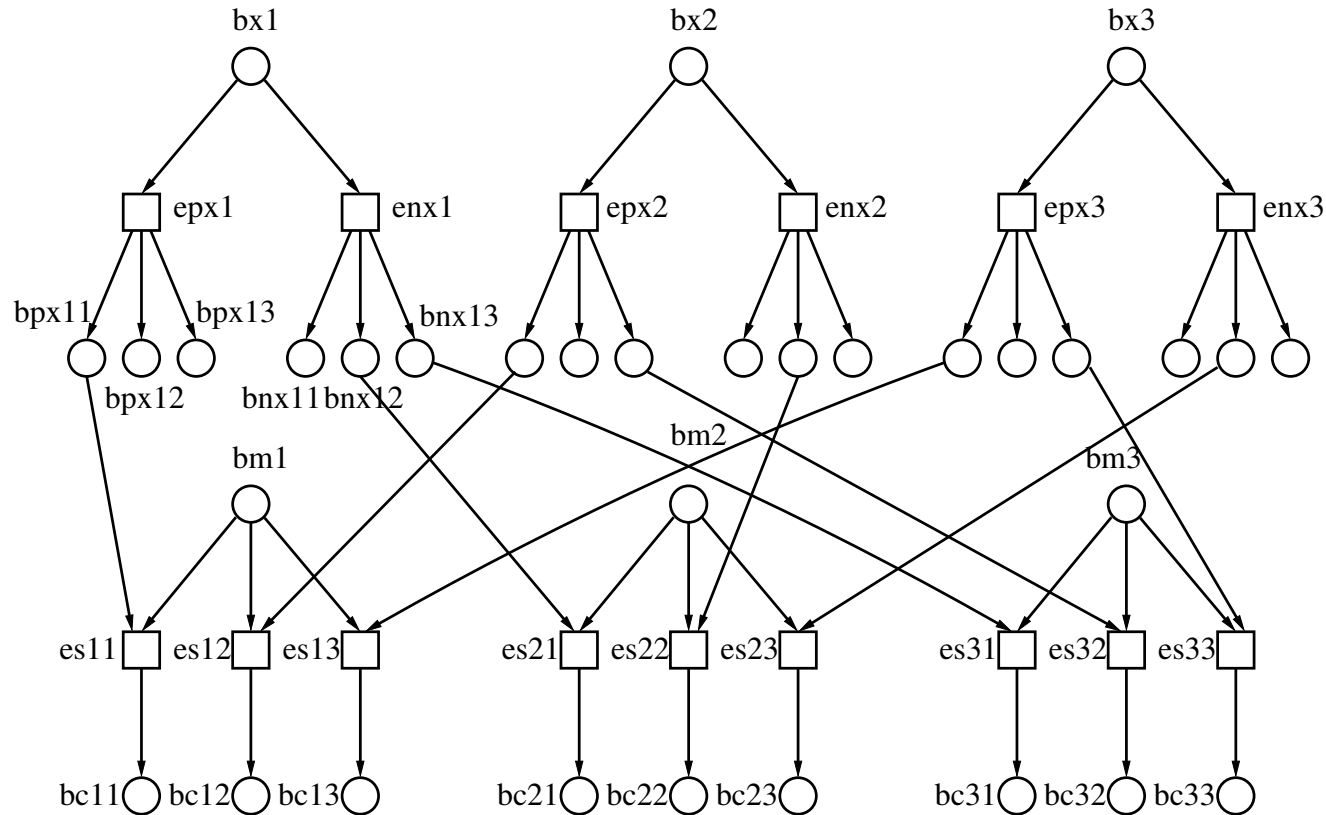
NP-Hardness of Possible Extensions

Consider the 3-SAT Formula below turned into a Petri net:
 $(x1 \vee x2 \vee v3) \wedge (!x1 \vee !x2 \vee !x3) \wedge (!x1 \vee x2 \vee x3)$



NP-Hardness of Possible Extensions

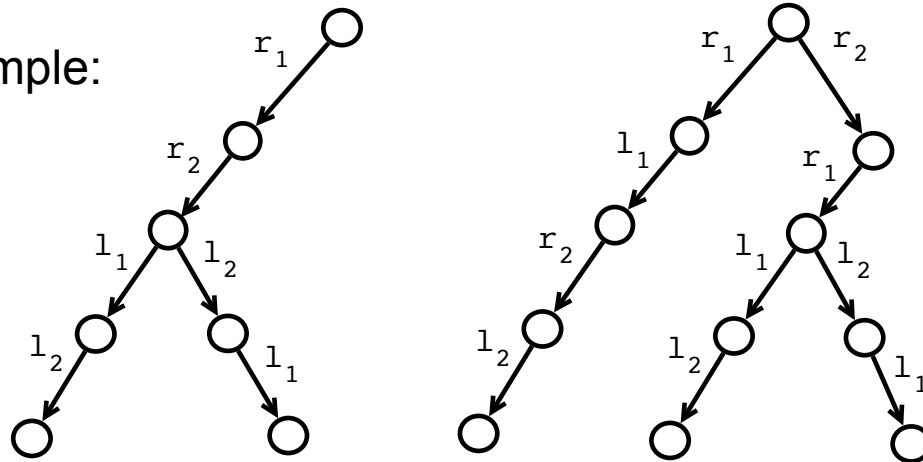
- The formula is satisfiable iff transition t is a possible extension of the following prefix of the unfolding:



Comparison with DPOR and Race Detection and Flipping

- The amount of reduction obtained by dynamic partial-order approaches depend on the order events are added to the symbolic execution tree
 - Unfolding approach always generates **canonical representation** regardless of the execution order

DPOR example:



Comparison with DPOR and Race Detection and Flipping

- Unfolding approach is computationally more expensive per test run than DPOR but requires less test runs
 - The reduction to the number of test runs can be exponential
 - Recall the system with $2N$ threads and N shared variables, which consist of a thread reading and writing a variable X_i .
 - It has an exponential number of Mazurkiewics traces but a linear size unfolding

Additional Observations

- The unfolding approach is especially useful for programs whose control depends heavily on input values
 - DPOR might have to explore large subtrees generated by DSE multiple times if it does not manage to ignore all irrelevant interleavings of threads
- One limitation of ASE'12 algorithm is that it does not cleanly support dynamic thread creation
 - Fixed in our ACSD'14 paper by using contextual nets

Using Contextual Unfoldings (ACSD'14)

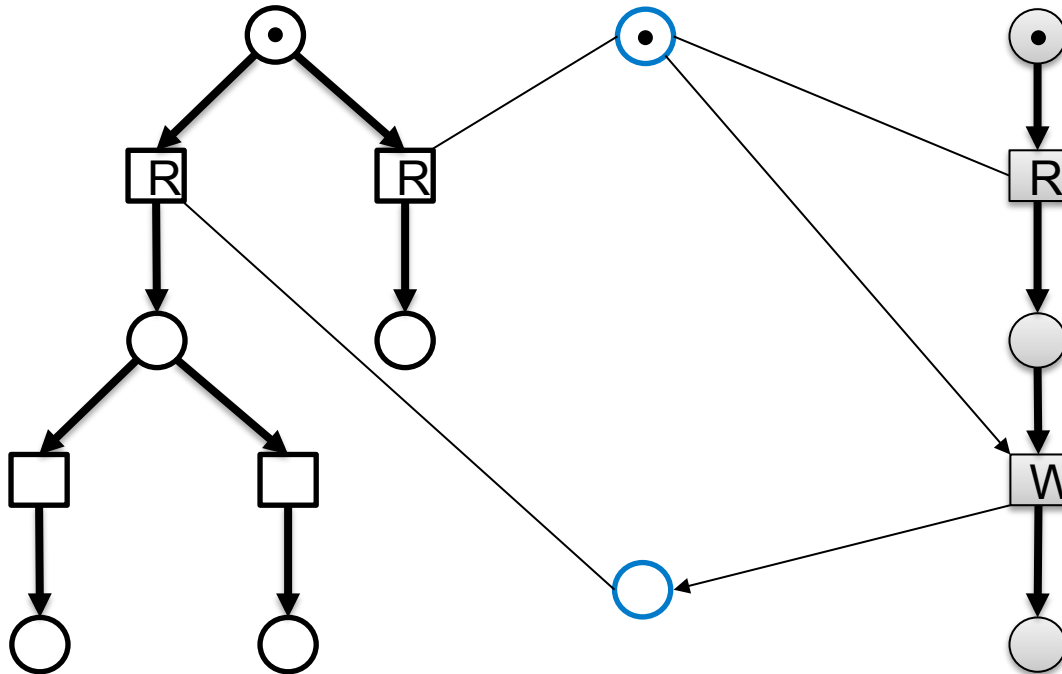
- *Contextual nets* (Petri nets with read arcs) allow an even more compact representation of the control and data flow
- *Read arcs* are denoted with lines instead of arrows
- A read arc requires a token to be present for transition to be enabled but firing it does not consume the token
- A more compact representation using read arcs can potentially be covered with less test executions
- However, computing potential extensions becomes computationally more demanding in practice (not in theory)

Example with Read Arcs

Global variables:
`int x = 0;`

Thread 1:
`local int a = x;`
`if (a > 0)`
`error();`

Thread 2:
`local int b = x;`
`if (b == 0)`
`x = input();`



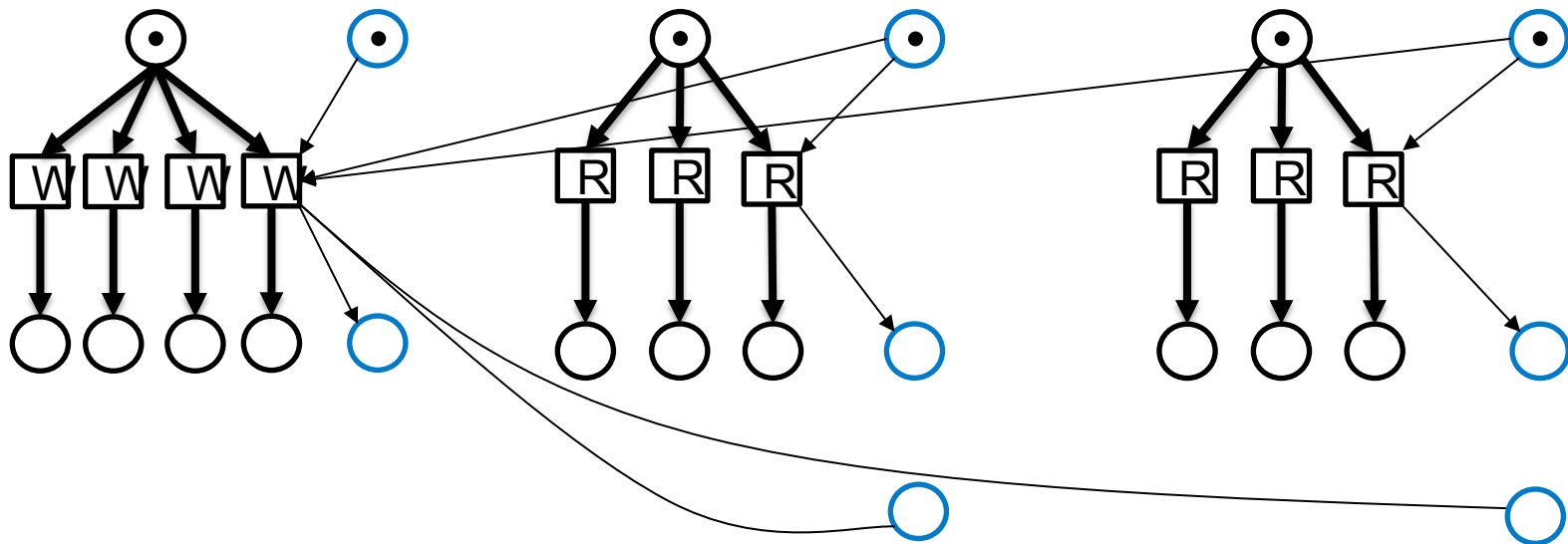
Another Example (Place Replication)

Global variables:
int x = 0;

Thread 1:
x = 5;

Thread 2:
local int a = x;

Thread 3:
local int b = x;



- Requires 4 test executions to explore all subsets of two reads before write

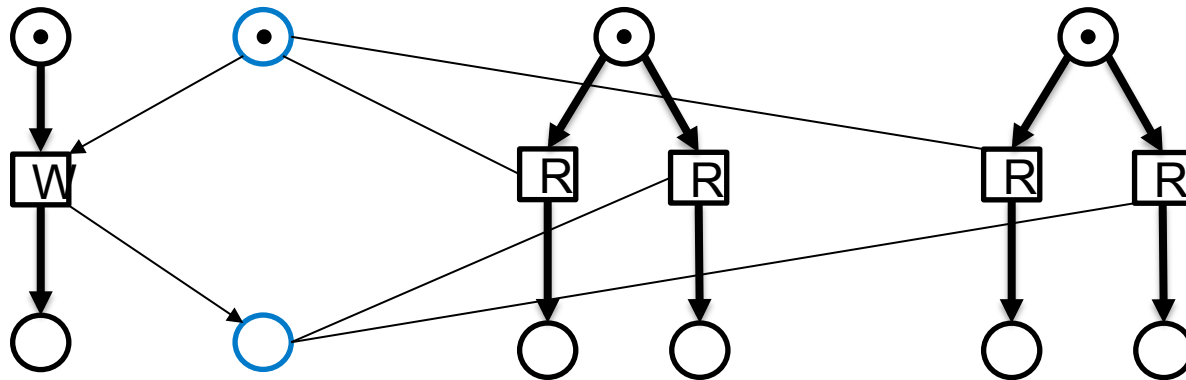
Another Example (Read Arcs)

Global variables:
int x = 0;

Thread 1:
x = 5;

Thread 2:
local int a = x;

Thread 3:
local int b = x;



- Contextual unfoldings can be exponentially more compact (just add more reads)!
- Only requires two test to be covered

Almost the Same Example Program

Thread 1:

```
a = input();
```

```
if (a == 10)
```

```
    a = 0;
```

Thread 2:

```
b = X;
```

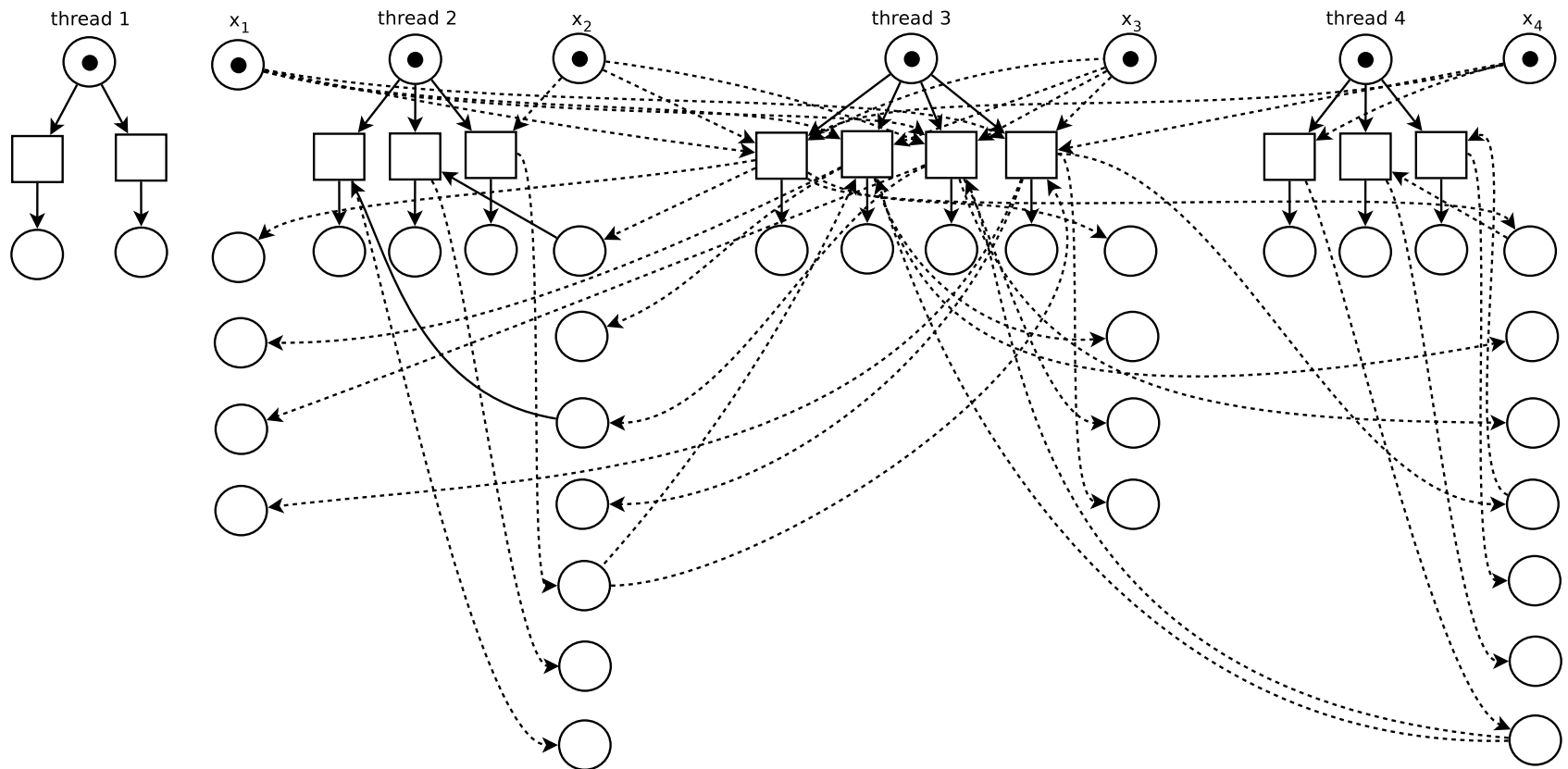
Thread 3:

```
X = 5;
```

Thread 4:

```
c = x;
```

Unfolding Using Petri Net – All arcs!

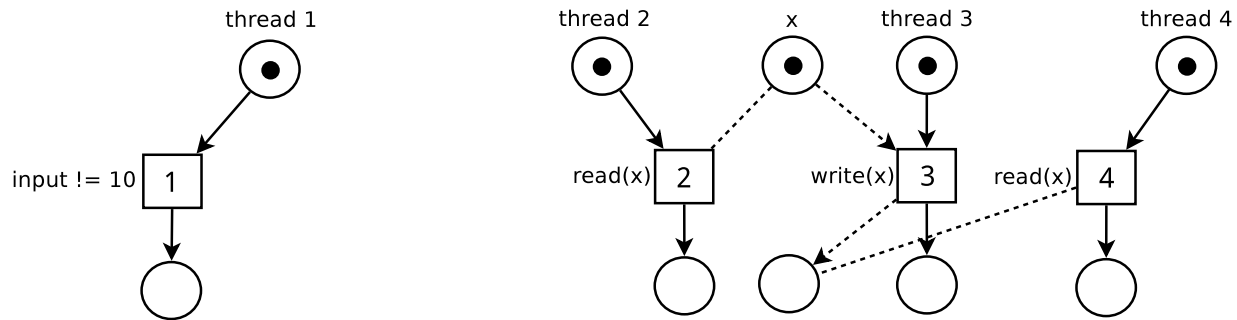


Unfolding with Contextual Nets – Natural

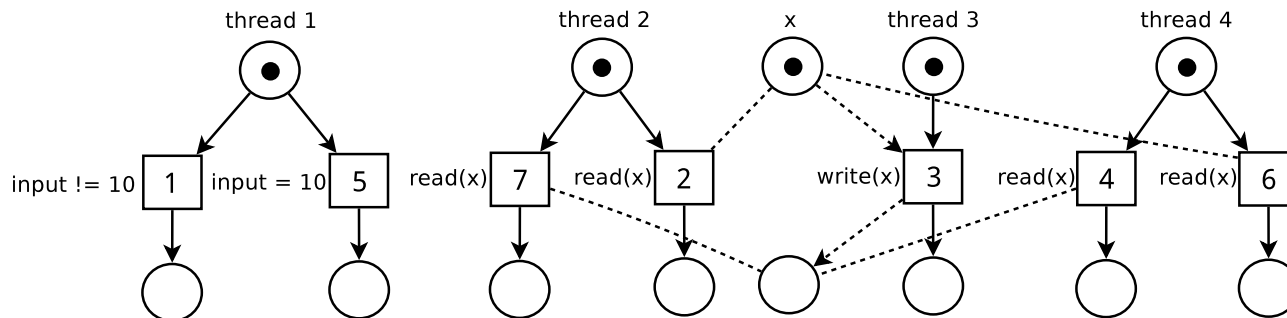
initial unfolding



unfolding after one test execution



complete unfolding



Additional Example

Global variables:

$X = 0;$

Thread 1:

1: $X = 5;$

2: $a = X;$

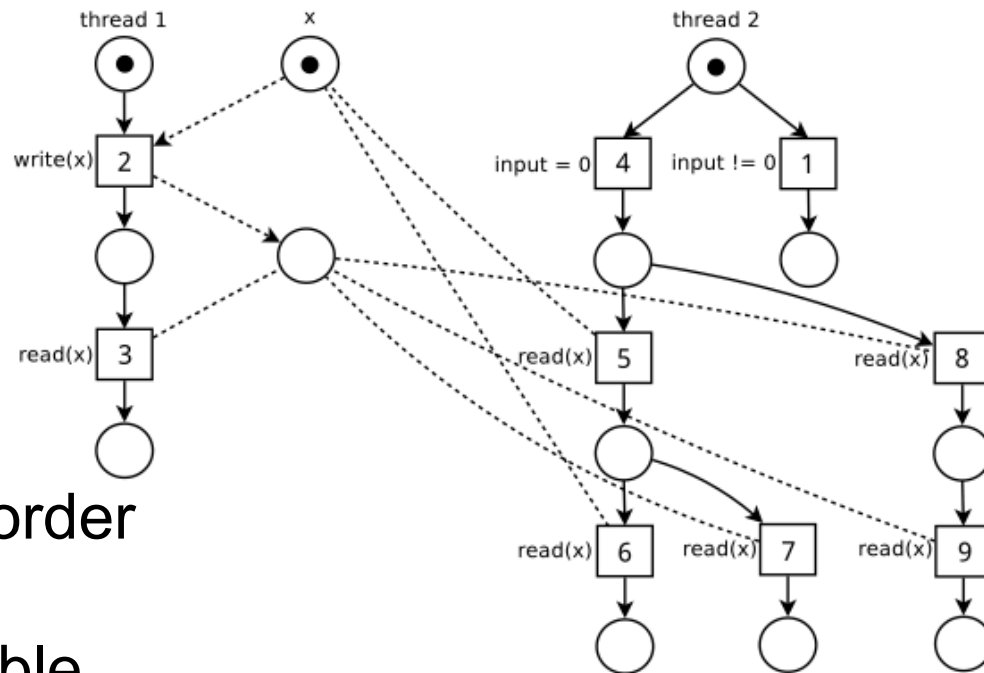
Thread 2:

3: $b = \text{input}();$

4: **if** ($b == 0$)

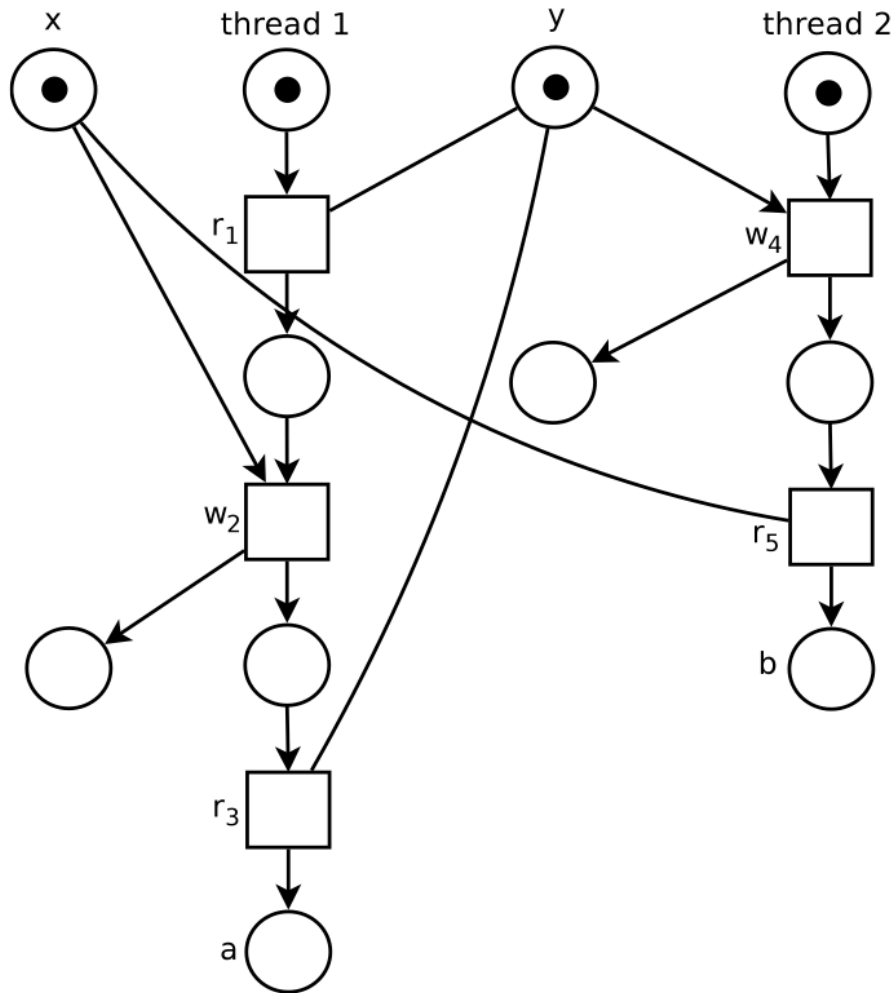
5: $c = X;$

6: $d = X;$



- Execution order
2,4,5
is impossible

Read Arcs Induce Cycles of Causality



- No way to fire both r_3 and r_5 !
- w_2 before r_3
- r_3 before w_4
- w_4 before r_5
- r_5 before w_2
- No ordering of $\{w_2, r_3, w_4, r_5\}$ possible!

Petri nets vs Contextual nets

- Contextual nets are sometimes much more (exponentially) compact (but not for all systems!)
- Contextual nets often have also less test runs to explore for the same system
- Contextual nets have a more complex theory
- Algorithms for contextual nets are more complex (need to check for acyclicity of ordering relation), and thus sometimes slower per generated test case
- Dynamic thread creation is easy with contextual nets

Experiments – Unfoldings vs DPOR

program	Unfolding		DPOR	
	paths	time	paths	time
Szymanski	65138	2m 3s	65138	0m 30s
Filesystem 1	3	0m 0s	142	0m 4s
Filesystem 2	3	0m 0s	2227	0m 46s
Fib 1	19605	0m 17s	21102	0m 21s
Fib 2	218243	4m 18s	232531	4m 2s
Updater 1	33269	2m 22s	33463	2m 6s
Updater 2	33497	2m 24s	34031	2m 13s
Locking	2520	0m 8s	2520	0m 6s
Synthetic 1	926	0m 3s	1661	0m 4s
Synthetic 2	8205	0m 41s	22462	1m 20s

Experiments – Petri vs Contextual nets

program	Unfolding		Contextual unfolding	
	paths	time	paths	time
Szymanski	65138	2m 3s	65138	2m 37s
Fib 1	19605	0m 17s	4959	0m 6s
Fib 2	218243	4m 18s	46918	0m 54s
Updater 1	33269	2m 22s	33269	3m 24s
Synthetic 1	926	0m 3s	773	0m 3s
Synthetic 2	8205	0m 41s	3221	0m 18s
Locking 2	22680	0m 55s	22680	1m 3s

Summary for Unfoldings in Testing

- A new approach to test multithreaded programs
- The restricted form of the unfoldings allows efficient implementation of the algorithm, crucial for performance!
- Unfoldings are competitive with existing approaches and can be substantially faster in some cases
- Can be exponentially smaller than any persistent set algorithm – Only preserves local state reachability
- Global state reachability is more complex:
 - Encode the unfolding as SMT formula in order to check global properties of the program under test

Lightweight State Capturing for Automated Testing of Multithreaded Programs (TAP'14)

- DSE testing tools usually do not store the reached states explicitly
- It can be very expensive to test whether a set of states reached by one branch is a subset of a set of states reached by another branch
- Usually such a subsumption check involves SMT solver proving inclusion between two symbolic path constraints, which can be expensive
- What can be done without using SMT solver based inclusion checks?
- Our Solution: Use a Petri net abstraction to capture some paths with identical sets of reachable states – cut-off testing when such states are found

Motivating Example

- The two interleavings will result in the same final state:

Global variables:

```
int x = 0;
```

```
int y = 0;
```

Thread 1:

```
1: acquire(lock);
```

```
2: x = 1;
```

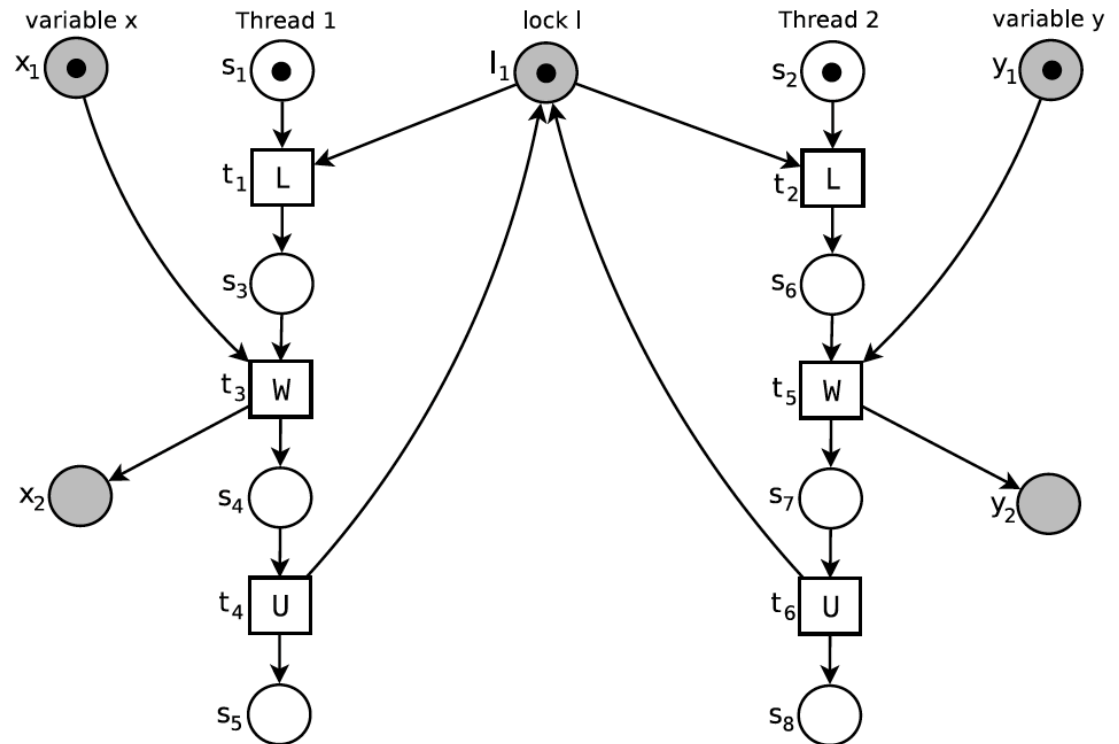
```
3: release(lock);
```

Thread 2:

```
4: acquire(lock);
```

```
5: y = 1;
```

```
6: release(lock);
```



Solution – Lightweight State Matching

- Generate from the tested system a state matching abstraction such that:
 - If the abstraction reaches a state M in two different test executions, then the symbolically represented sets of states reached by the two executions will be identical
 - Note: The converse is not true: If two different test executions reach the same symbolically represented sets of states, the abstraction might be in two different states M and M'

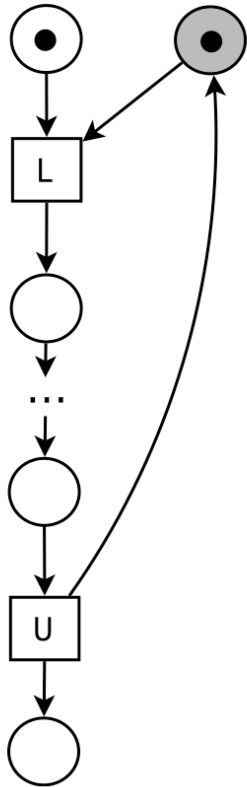
Solution – Lightweight State Matching

- We will use a Petri net based abstraction:
 - Locks have no internal state, lock+unlock combination will return the system into the same internal state
 - Reads will not modify the global variables read, just the local state of the process
 - Writes will modify both the local state of the process and the global variable

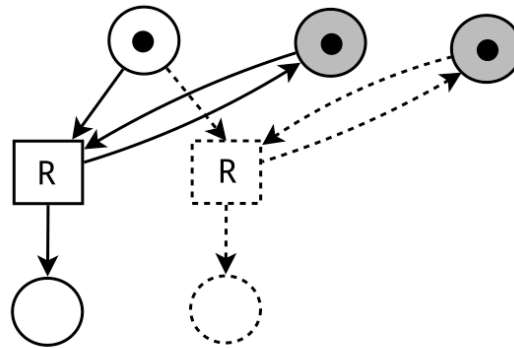
Abstraction Modeling Constructs

- Lock&unlock will always use the same lock place, read modifies only local state, write modifies local state&var:

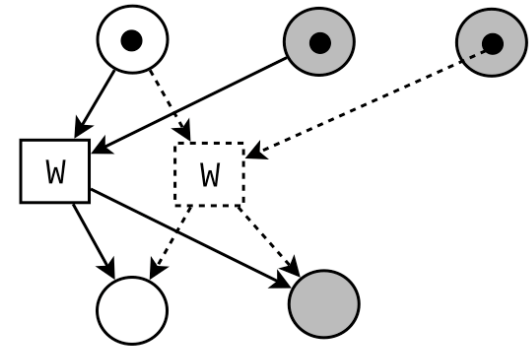
Locking and unlocking



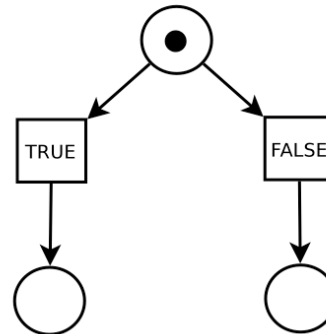
Reading a shared variable



Writing to a shared variable



Symbolic branching



Naïve Lightweight State Matching

Input: A program P

1: $model :=$ empty Petri net

2: $visited := \emptyset$

3: extend model with a random test execution

4: EXPLORE(M_0, \emptyset)

5: **procedure** EXPLORE(M, S)

6: **if** $M \notin visited$ **then**

7: $visited := visited \cup \{M\}$

8: PREDICTTRANSITIONSFROMMODEL(M)

9: **if** model is incomplete at M **then**

10: EXTENDMODEL(P, S, k)

11: **for all** transitions t enabled in M **do**

12: $M' :=$ FIRE(t, M)

13: $S' := S$ appended with t

14: EXPLORE(M', S')

- Just basic DFS with cut-off if the same abstract state M is seen twice



Unfolding based Lightweight State Matching

Input: A program P

```
1: model := empty Petri net
2: unf := initial unfolding
3: visited :=  $\emptyset$ 
4: extend model with a random test execution
5: extensions := events enabled in the initial state
6: while extensions  $\neq \emptyset$  do
7:   choose  $\leftarrow$  minimal event  $e$  from extensions
8:    $M := \mathbf{St}(e)$ 
9:   PREDICTTRANSITIONSFROMMODEL( $M$ )
10:  if model is incomplete at  $M$  then
11:    EXTENDMODEL( $P, e, k$ )
12:  else
13:    add  $e$  to unf
14:    extensions := extensions  $\setminus \{e\}$ 
15:    if  $M \notin \textit{visited}$  then //  $e$  is not a terminal
16:      visited := visited  $\cup \{M\}$ 
17:      extensions := extensions  $\cup$  POSSIBLEEXTENSIONS( $e, unf$ )
```

- Unfolding with cut-off if the same abstract state M is seen twice
- Note search order imposed on line 7!

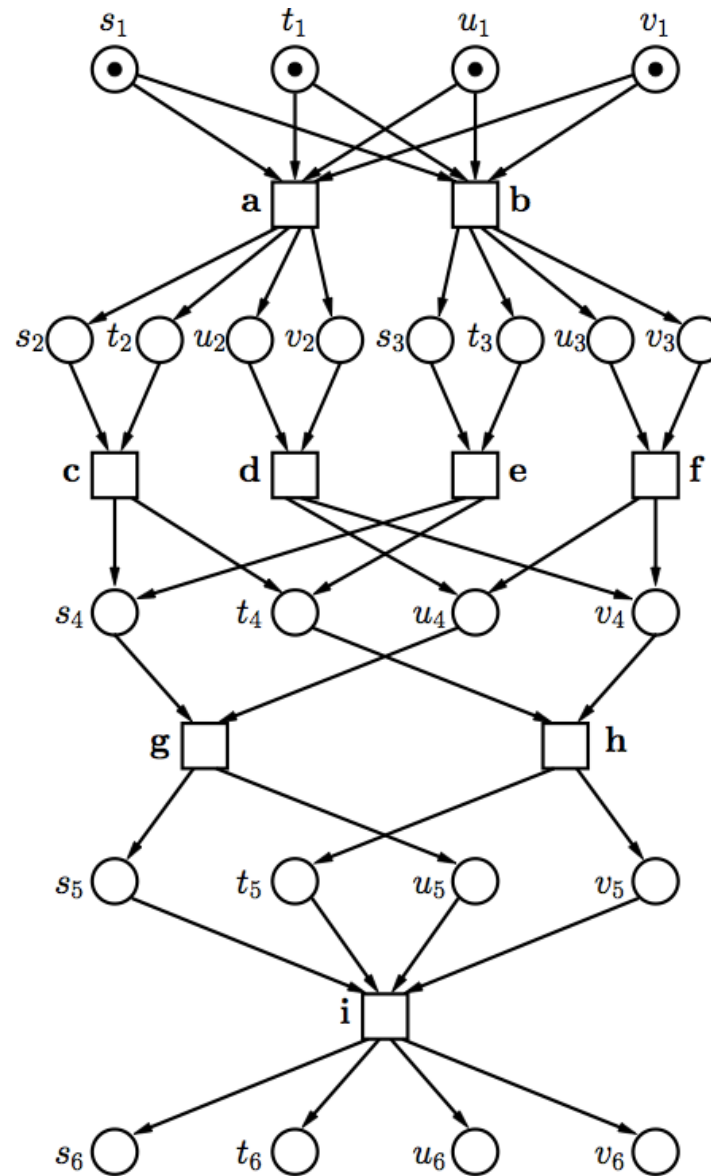


Why use Order $<$ on Line 7 when Extending Unfolding?

- Note that on line 7 a minimal event according to an [Adequate Order](#) $<$ on events to be added to the unfolding is selected
- Such an order was first presented by Esparza, J., Römer, S., Vogler, W.: [An improvement of McMillan's unfolding algorithm](#). Formal Methods in System Design 20(3), 285–310 (2002)
- **If an arbitrary order would be used on line 7, the algorithm would become unsound!**
- For more information, see Chapter 4.4 of: Esparza, J. and Heljanko, K.: Unfoldings - A Partial-Order Approach to Model Checking.

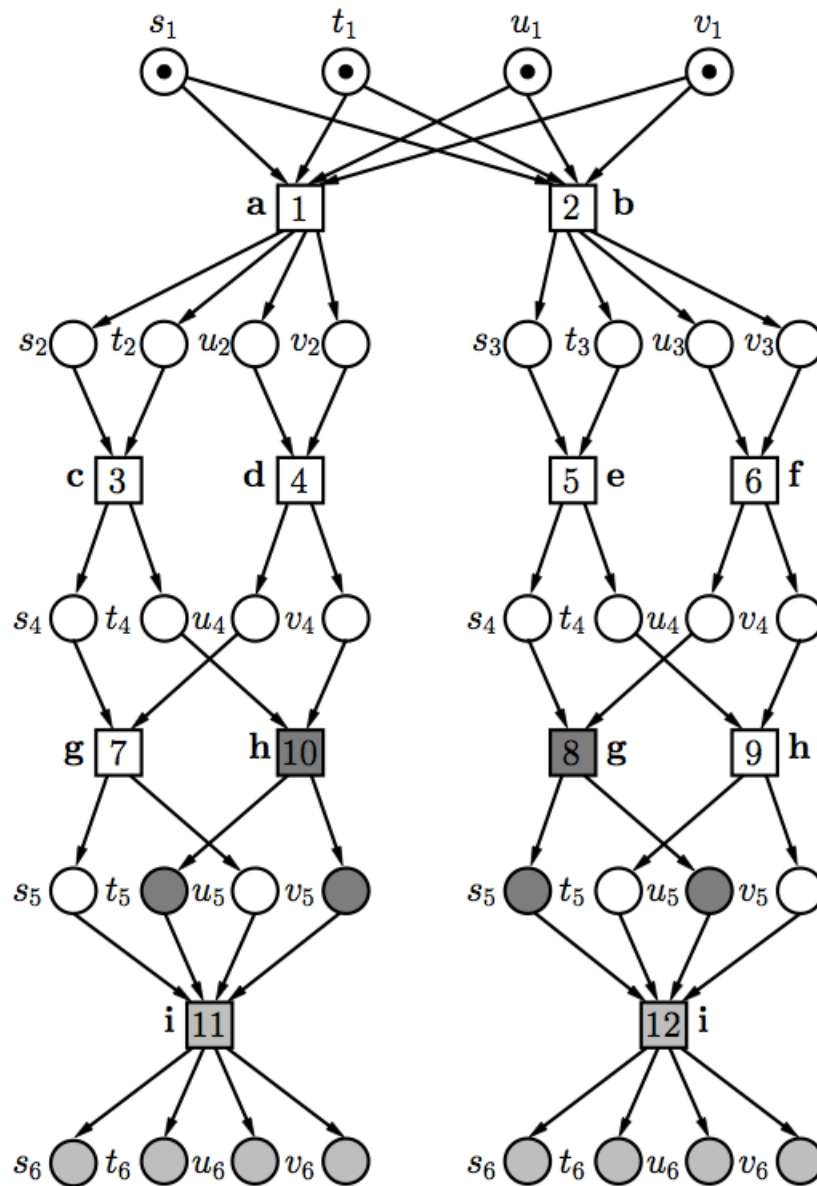
Unsoundness:

- Smallest known example where unsoundness occurs



Unsound Cuts:

- Using the order given by numbers will cut both 8 and 9, leaving both 11 and 12 undiscovered!
- This can be fixed!



Adequate Orders

- The adequate order $<$ on traces needs to satisfy the following:
 1. $<$ is well-founded (contains no infinite descending chain); and
 2. If $[w] < [w']$ then $[w w''] < [w' w'']$.
- Surprisingly Chatain and Khomenko were able to prove that (1) above implies (2), and thus a sufficient condition on an adequate order $<$ is that it is well founded
- Using property (2) repeatedly it is possible to show all reachable local states have a representative in the unfolding also when cut-offs are used to cut branches away
- Thus our algorithm is sound when $<$ is an adequate order!

Experimental Results

	Stateless unfolding		Stateless DPOR		Stateful naive		Stateful unfolding	
Benchmark	tests	time	tests	time	tests	time	tests	time
Fib 1	19605	0m 17s	21102	0m 21s	5746	0m 11s	4946	0m 15s
Fib 2	218243	4m 18s	232531	4m 2s	53478	3m 45s	46829	3m 15s
Filesystem 1	3	0m 0s	142	0m 4s	-	(> 30m)	3	0m 0s
Filesystem 2	3	0m 0s	2227	0m 46s	-	(> 30m)	3	0m 0s
Dining 1	798	0m 3s	1161	0m 3s	3	0m 0s	4	0m 0s
Dining 2	5746	0m 14s	10065	0m 22s	3	0m 1s	3	0m 1s
Dining 3	36095	1m 29s	81527	3m 29s	2	0m 7s	4	0m 1s
Dining 4	205161	12m 55s	-	(> 30m)	-	(> 30m)	2	0m 3s
Szymanski	65138	2m 3s	65138	0m 30s	50264	0m 43s	46679	2m 35s
Locking 1	2520	0m 8s	2520	0m 6s	20	0m 1s	18	0m 3s
Locking 2	22680	0m 56s	22680	0m 47s	29	0m 2s	26	0m 9s
Locking 3	-	(> 30m)	-	(> 30m)	115	0m 21s	89	3m 32s
Updater	33269	2m 22s	33463	2m 6s	13586	1m 23s	12259	1m 52s
Writes	-	(> 30m)	-	(> 30m)	1	0m 0s	1	0m 0s
Synthetic 1	926	0m 3s	1661	0m 4s	68	0m 1s	62	0m 1s
Synthetic 2	8205	0m 41s	22462	1m 20s	123	0m 7s	97	0m 11s
Synthetic 3	11458	1m 12s	37915	2m 18s	326	1m 8s	298	0m 30s

Summary – Lightweight State Matching

- Lightweight state matching can yield dramatic reductions in the number of explored test runs
- This holds despite that no SMT solver is being used for state matching
- The naïve state matching testing algorithm can sometimes beat advanced DPOR and unfolding approaches
- Adding lightweight state matching to DPOR is straightforward future work
- Adding state matching to unfoldings requires the use of adequate orders from unfolding theory to remain sound

Conclusions

- When testing multithreaded programs **partial order reductions should be used**
 - Partial order reduction combines nicely with dynamic symbolic execution (DSE)
 - **DPOR together with sleep sets can be implemented in a fairly straightforward fashion** based e.g., on our ACSD'12 paper for pseudocode and PDMC'12 for its parallelization
 - **Unfoldings can be exponentially more compact than DPOR**
 - More complex theory and algorithms needed
 - **State matching is essential** to improve performance
 - **TODO: Explore border between Testing and Model Checking**
-



References for this talk

- Kari Kähkönen, Olli Saarikivi, Keijo Heljanko: [Using unfoldings in automated testing of multithreaded programs](#). [ASE 2012](#): 150-159
- Kari Kähkönen and Keijo Heljanko: [Testing Multithreaded Programs with Contextual Unfoldings and Dynamic Symbolic Execution](#). [ACSD 2014](#), to appear.
- Kari Kähkönen and Keijo Heljanko. [Lightweight State Capturing for Automated Testing of Multithreaded Programs](#). [TAP 2014](#), to appear.

References for this talk (cnt.)

- Olli Saarikivi, Kari Kähkönen, Keijo Heljanko: [Improving Dynamic Partial Order Reductions for Concolic Testing](#). [ACSD 2012](#): 132-141
- Kähkönen, K., Saarikivi, O., and Heljanko, K.: [LCT: A Parallel Distributed Testing Tool for Multithreaded Java Programs](#). [PDMC 2012](#): Electronic Notes in Theoretical Computer Science (ENTCS), Volume 296, 2013, p. 253-259
- Kari Kähkönen: [Automated Systematic Testing Methods for Multithreaded Programs](#), PhD Thesis Draft, June 5, 2014.