Using Unfoldings in Automated Testing of Multithreaded Programs

Kari Kähkönen, Olli Saarikivi, Keijo Heljanko
(firstname.lastname@aalto.fi)
Department of Computer Science and Engineering,
Aalto University &
Helsinki Institute for Information Technology

Published in ASE'12

25th Nordic Workshop on Programming Theory, NWPT '13
Tallinn, Estonia, 20th November 2013

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

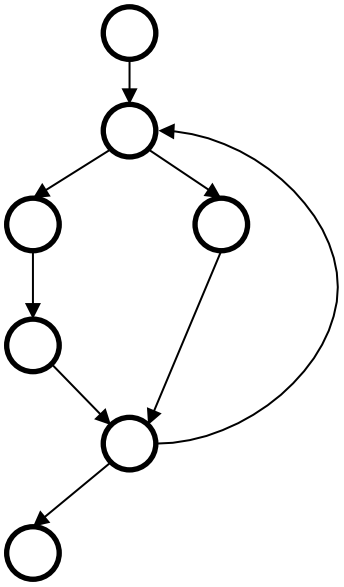
- How to automatically test the local state reachability in multithreaded programs that read input values
  - E.g., find assertion violations, uncaught exceptions, etc.
- Our tools use a subset of Java as its input language
- The main challenge: path explosion and numerous interleavings of threads
- 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) {
  ...
}
...
```



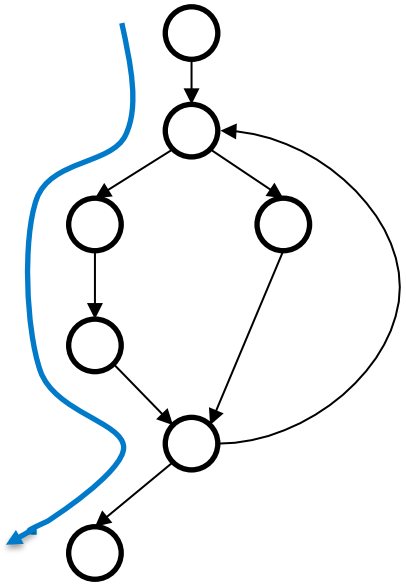
Control flow graph

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) {
  ...
}
...
```



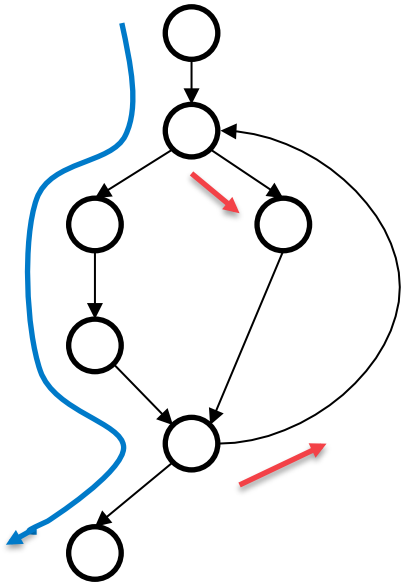
Control flow graph

Dynamic Symbolic Execution

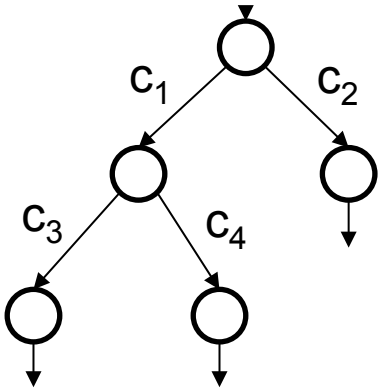
- Symbolic execution generates constraints at branch points that define input values leading to true and false branches

```
x = input
x = x + 5

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



Control flow graph

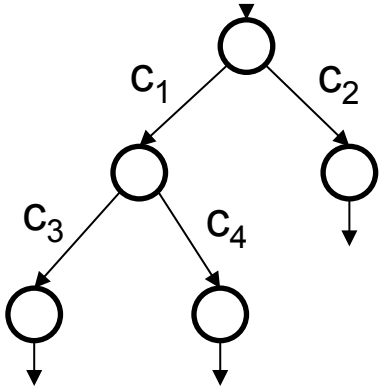


$c_1 = input_1 + 5 > 10$

$c_2 = input_1 + 5 \leq 10$

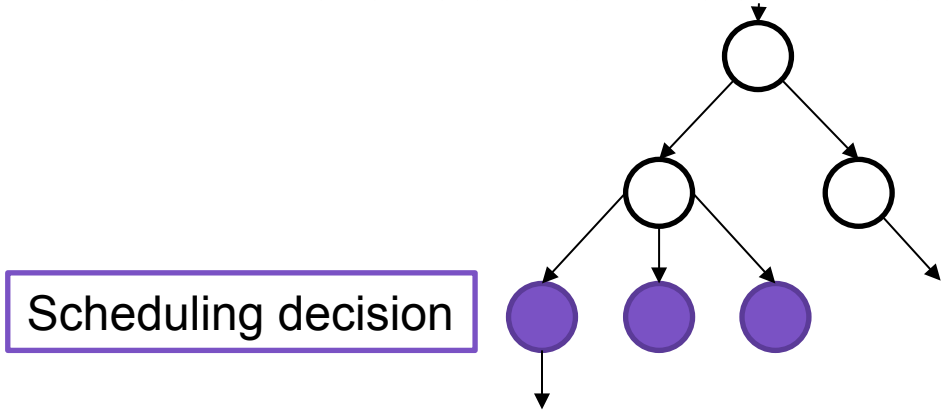
Dynamic Symbolic Execution

- 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: $\text{input}_1 = 10 \text{ and } \text{input}_2 = 5$



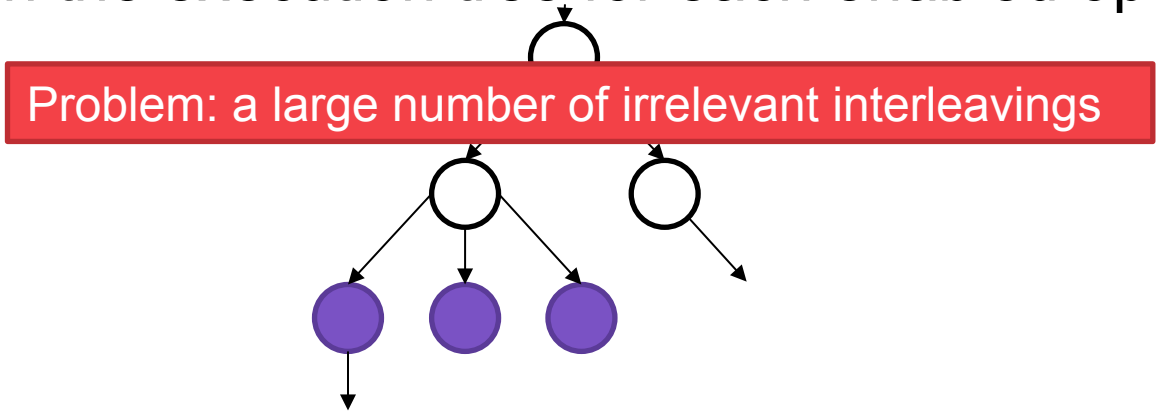
What about Multithreaded Programs?

- 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



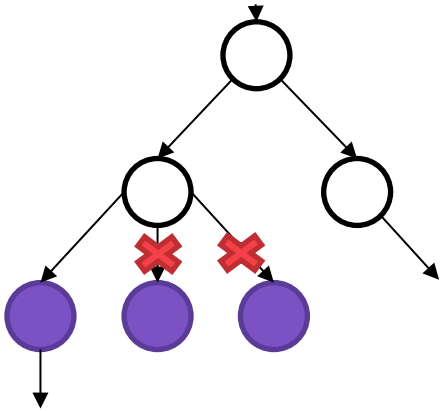
What about Multithreaded Programs?

- 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



One Solution: Partial-Order Reduction

- Ignore provably irrelevant parts of the symbolic execution tree



- Existing algorithms:
  - dynamic partial-order reduction (DPOR) [FlaGod05]
  - race detection and flipping [SenAgh06]

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 assertion violations and deadlocks
- As any persistent set approach, preserves one interleaving from each Mazurkiewicz trace

Identifying Backtracking Points in DPOR

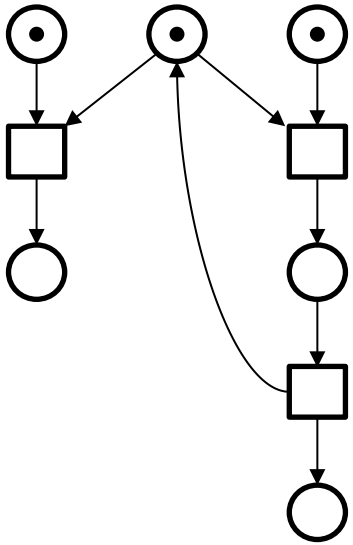
- When a race is identified during execution, DPOR adds a backtracking point is added to be explored later
- To do so, DPOR tracks the causal relationships of global operations in order to identify backtracking points
- In typical implementations the causal relationships are tracked by using vector clocks
- An optimized DPOR approach can be found from:
  - Saarikivi, O., Kähkönen, K., and Heljanko, K.: Improving Dynamic Partial Order Reductions for Concolic Testing. In ACSD 2012.

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
- When the executed tests cover the symbolic representation completely, the testing process can be stopped

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

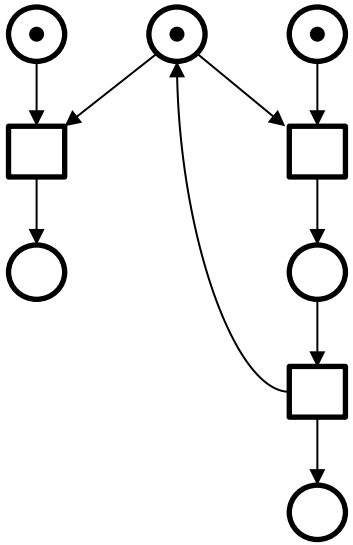


Petri net

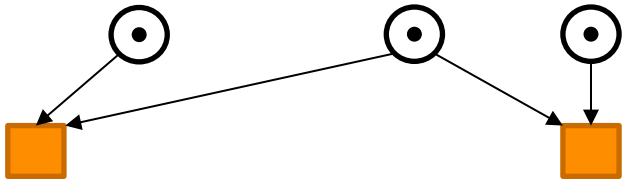
Initial unfolding

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



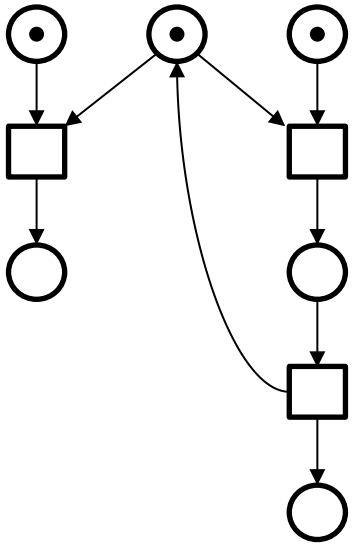
Petri net



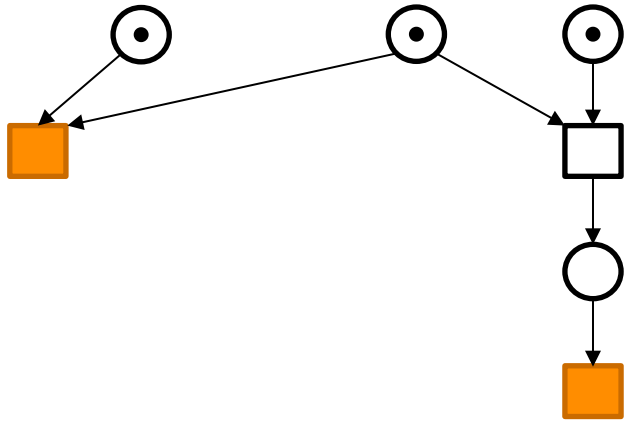
Unfolding

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



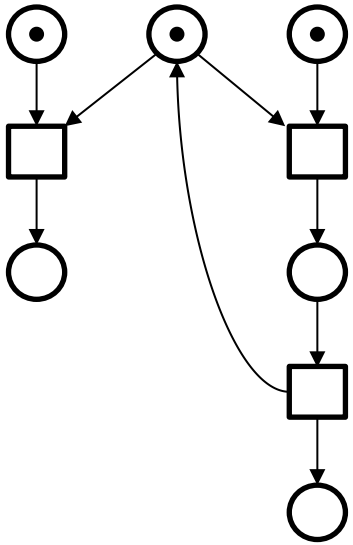
Petri net



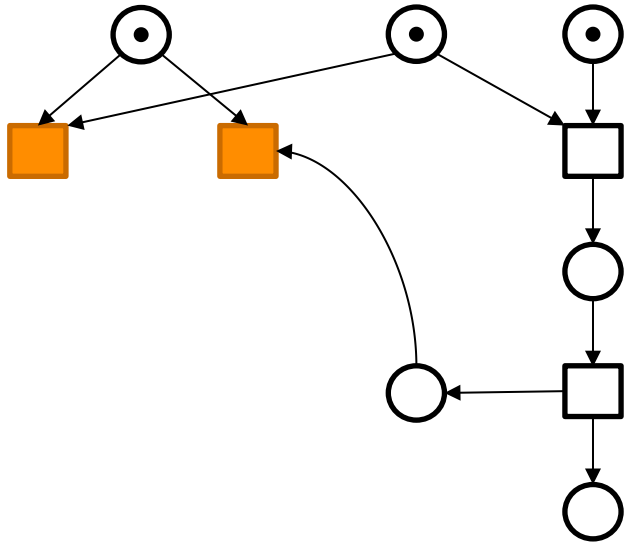
Unfolding

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



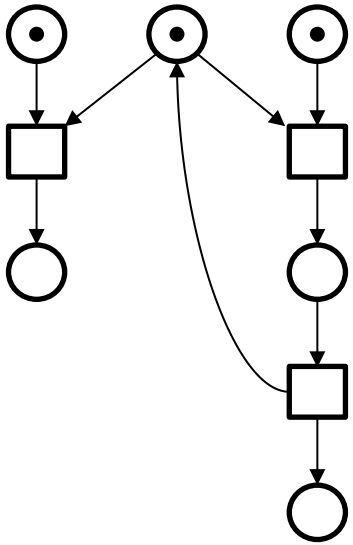
Petri net



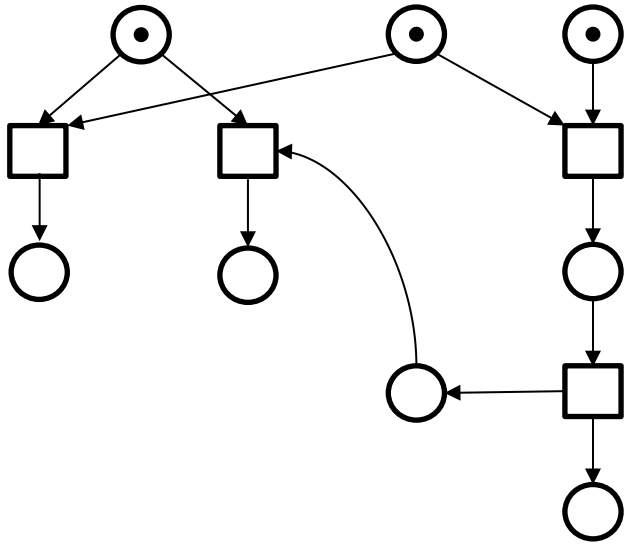
Unfolding

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



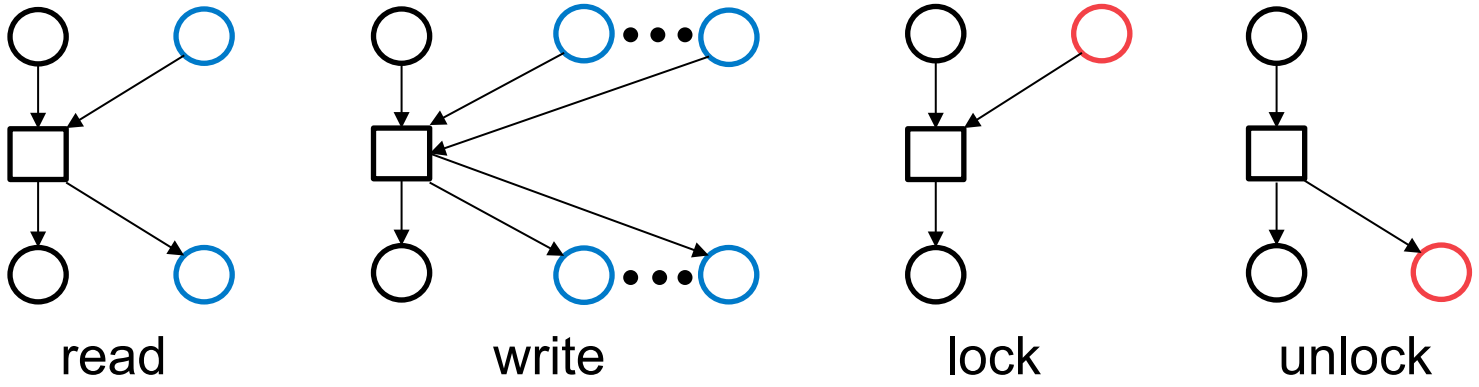
Petri net



Unfolding

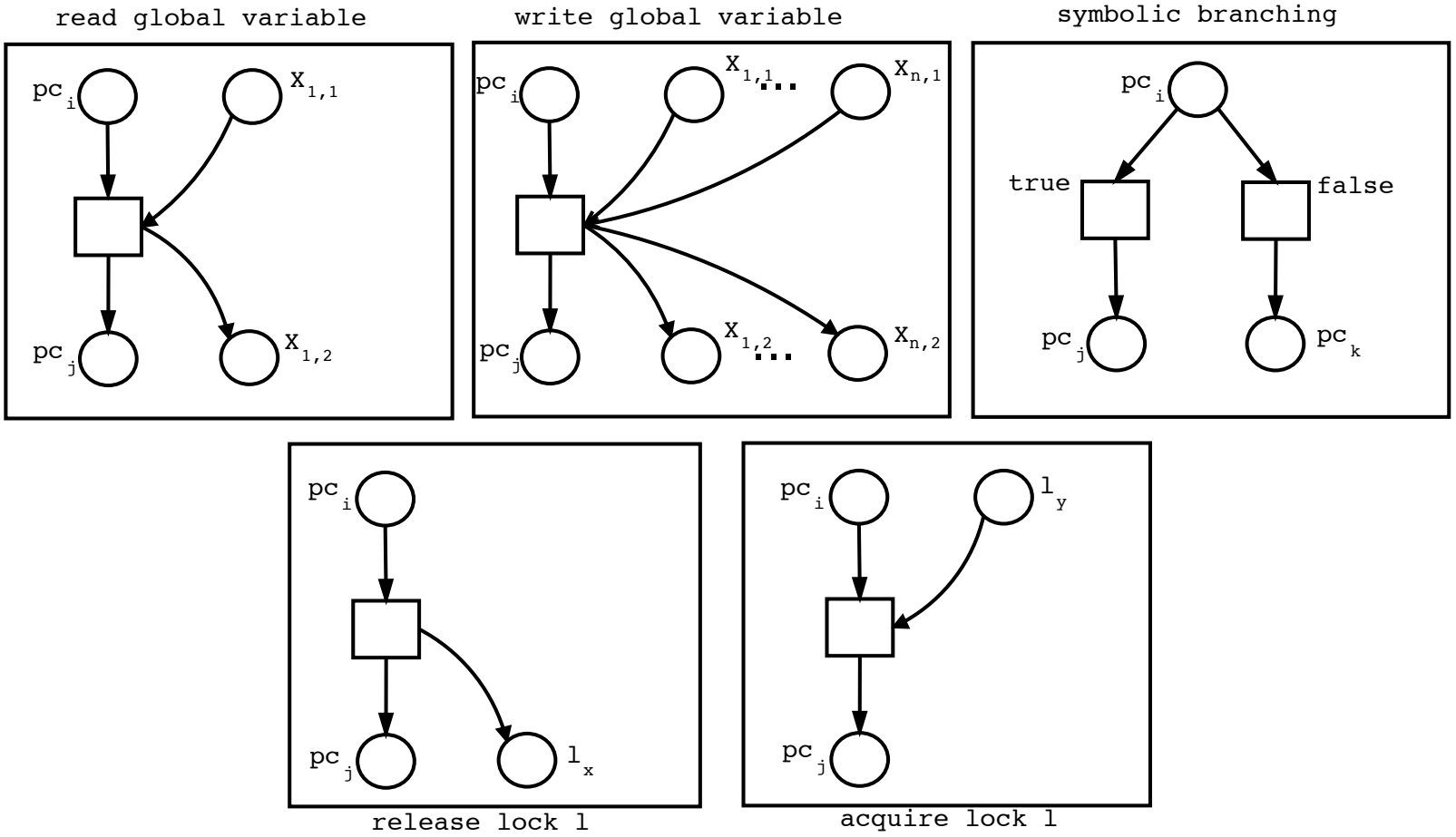
Using Unfoldings with DSE

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



- Potential extensions for the added event are new test targets

Shared Variables have Local Copies



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, 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 are pruned when found

Example

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();

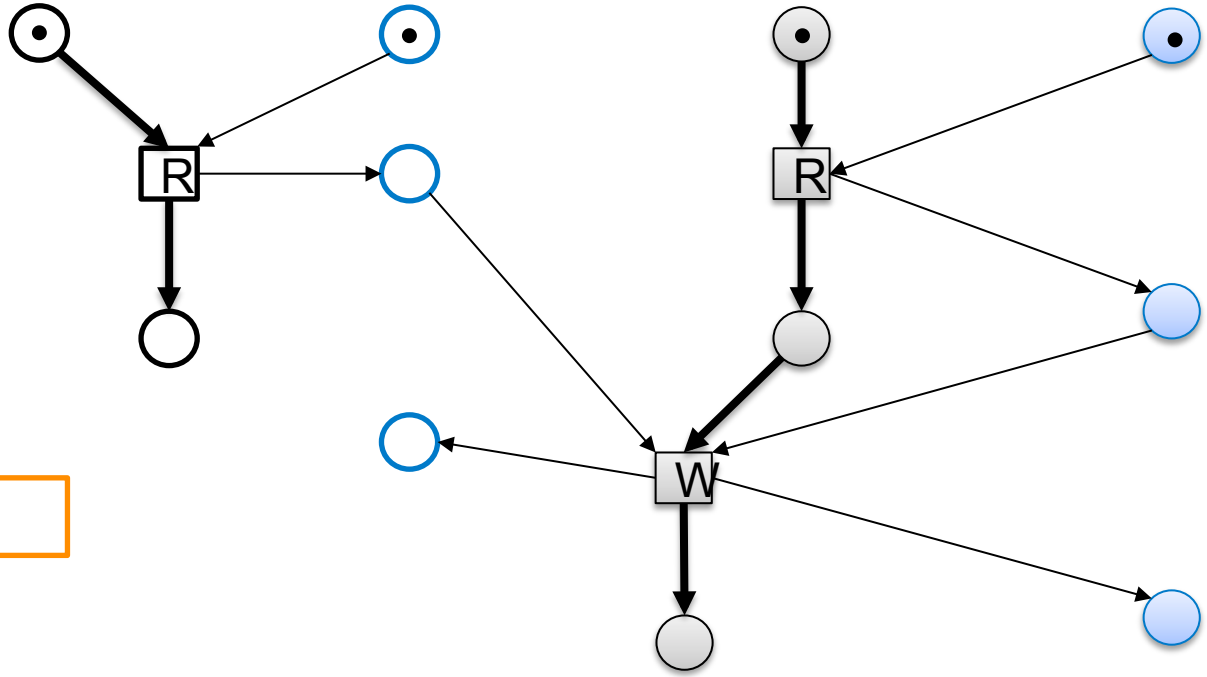
Initial unfolding

Example

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();



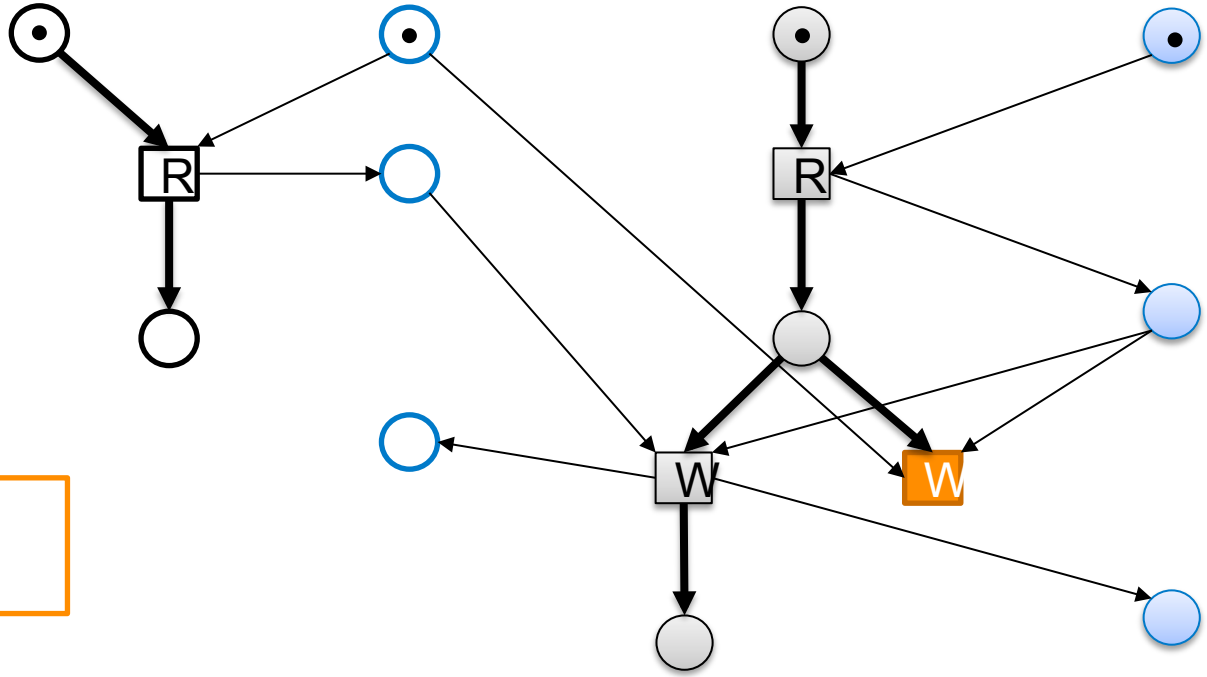
First test run

Example

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();



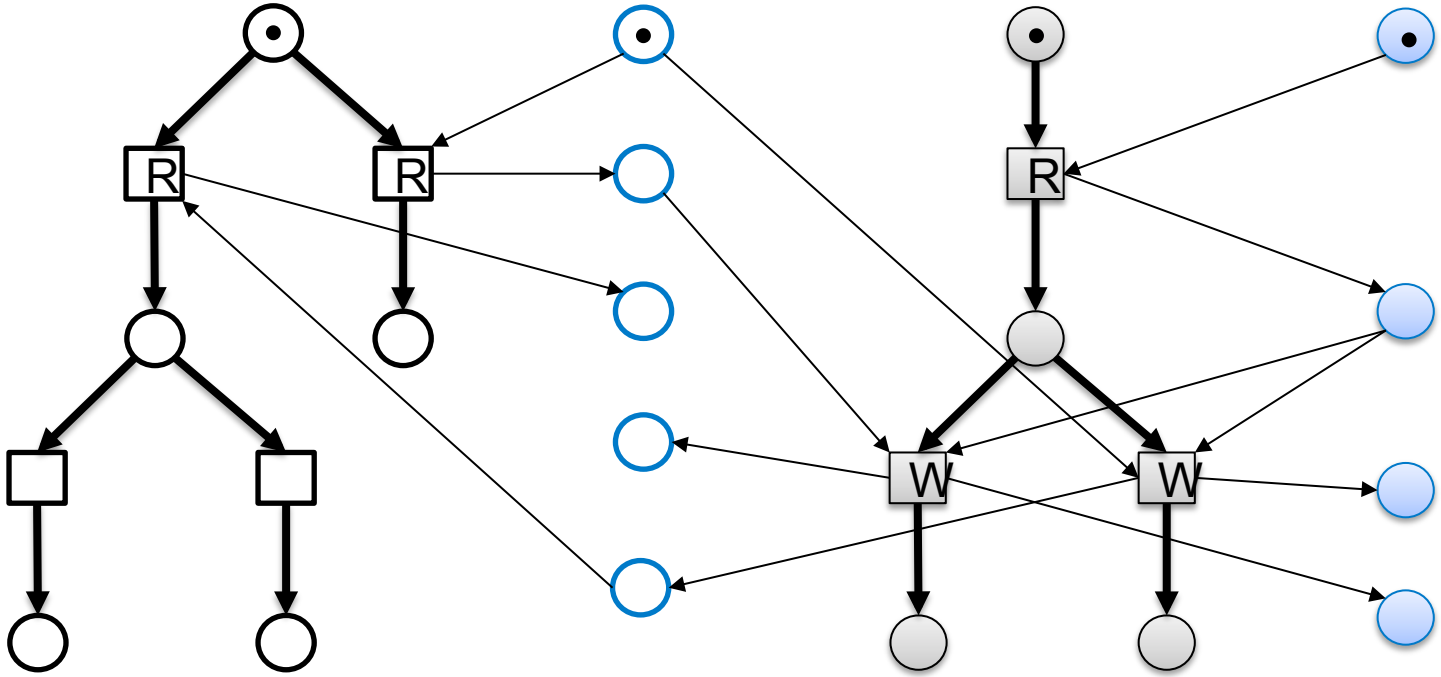
Find possible extensions

Example

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();

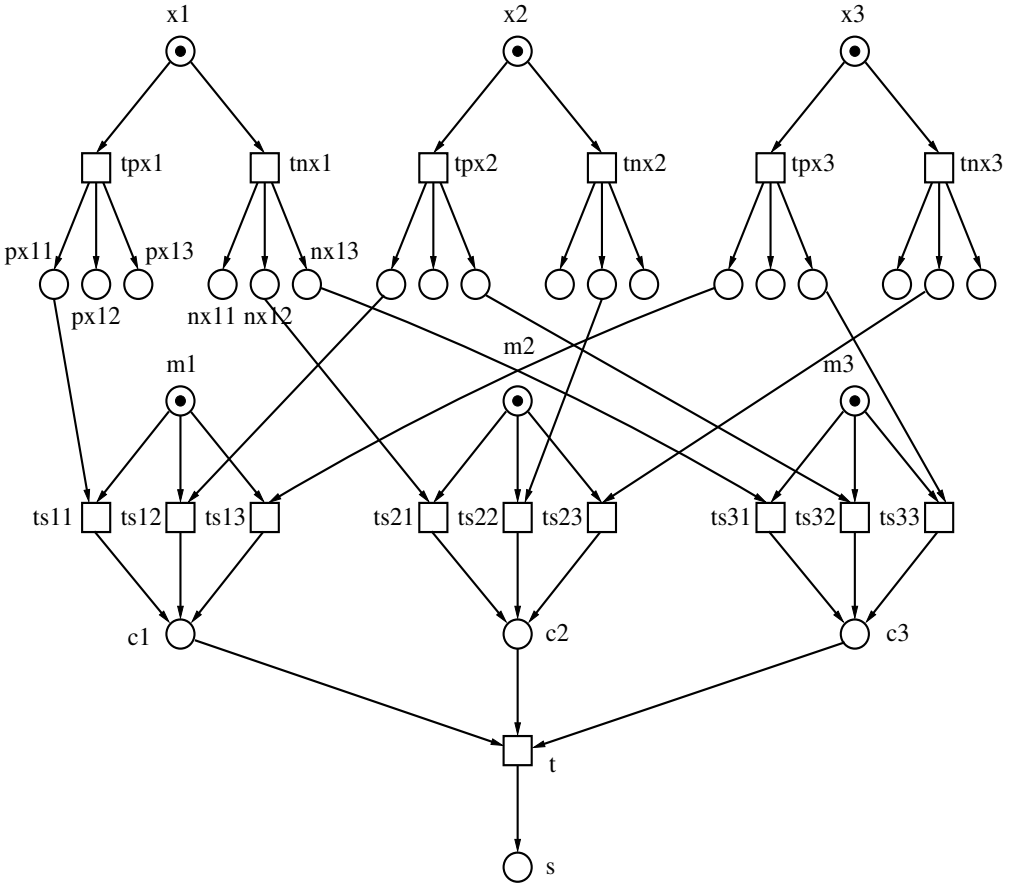


Computing Potential Extensions

- Finding potential extensions is the most computationally expensive part of unfolding (NP-complete [Heljanko'99])
- It is possible to use existing 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

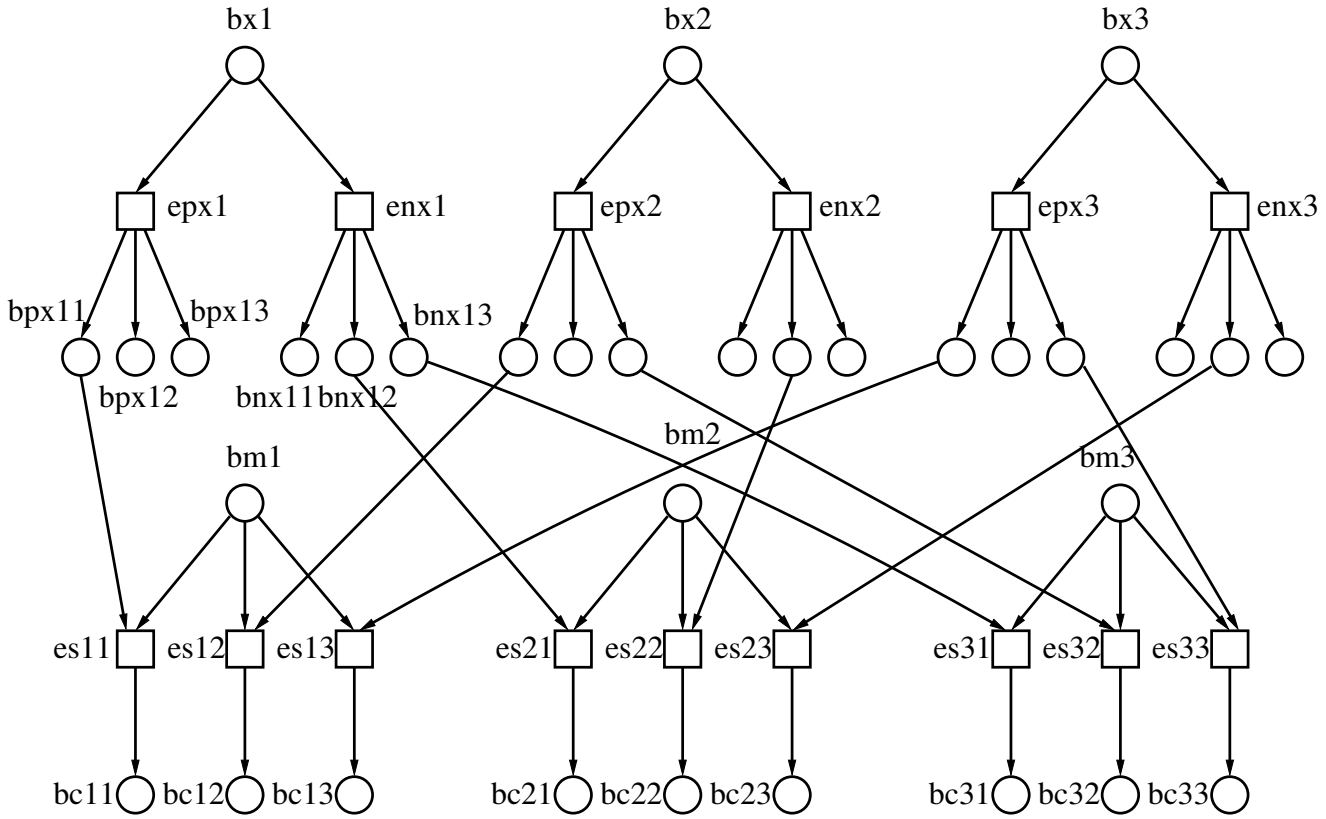
NP-Hardness of Possible Extensions

Consider the 3-SAT Formula below turned into a Petri net:
(x1 ∨ x2 ∨ v3) ∧ (!x1 ∨ !x2 ∨ !x3) ∧ (!x1 ∨ x2 ∨ x3)



NP-Hardness of Possible Extensions

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

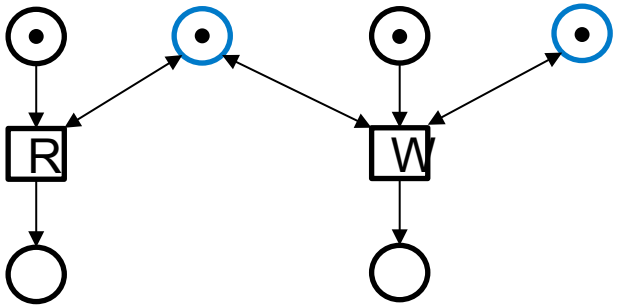


Computing Potential Extensions

- 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 computations in practice

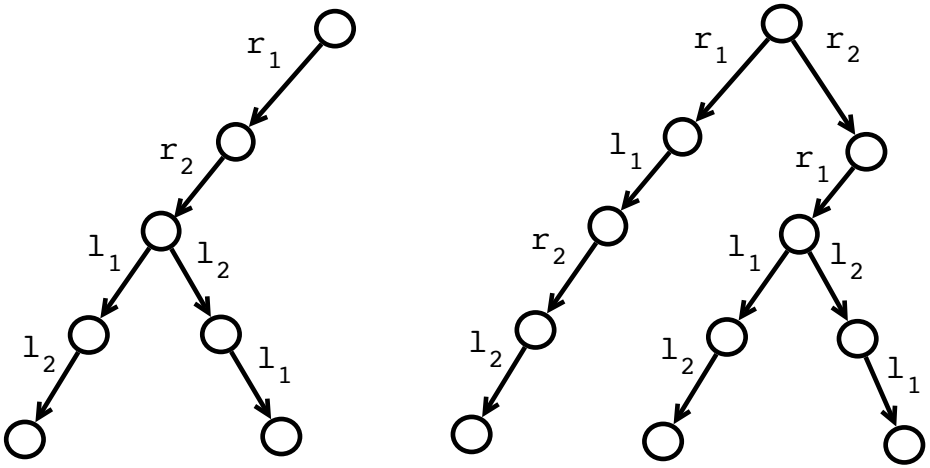
Thread 1:
local int a = x; (read)

Thread 2:
x = 5; (write)



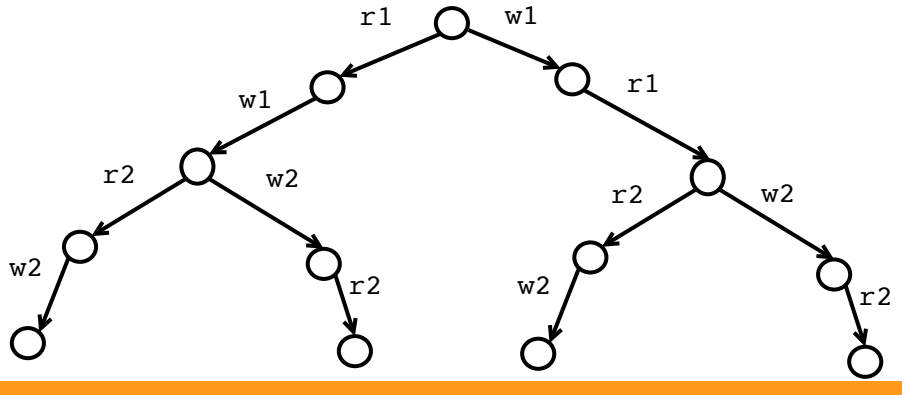
Comparison with DPOR and Race Detection and Flipping

- 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



Comparison with DPOR and Race Detection and Flipping

- Unfolding approach is computationally more expensive per test run but requires less test runs
  - The reduction to the number of test runs can be exponential
  - Consider a system with 2n threads and n shared variables, which consist of a thread reading (ri) and writing (wi) variable 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
  - Suggested solution to explore in ASE'12 paper: Contextual nets, i.e. Petri nets with read-arcs

New: Using Contextual Unfoldings

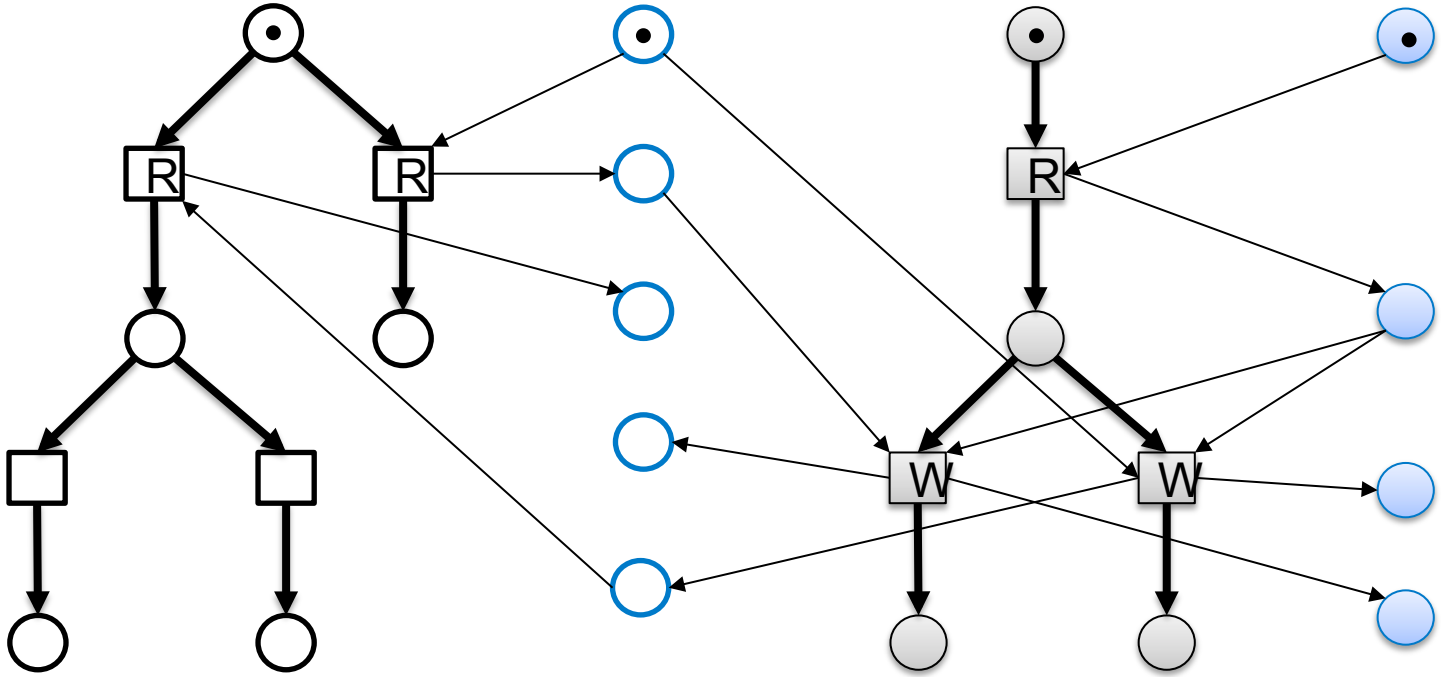
- Contextual nets (nets with read arcs) allow an even more compact representation of the control and data flow
- A more compact representation can potentially be covered with less test executions
- However, computing potential extensions becomes computationally more demanding in practice (not in theory)

Recap: Example as Ordinary Petri net

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();

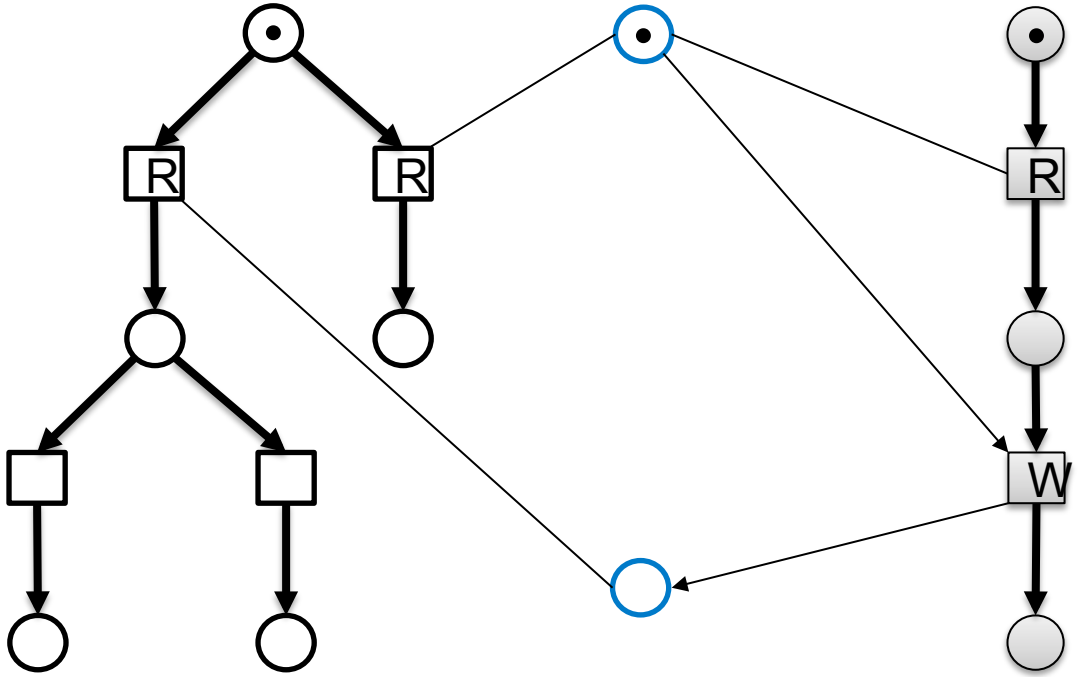


Example with Read Arcs

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();



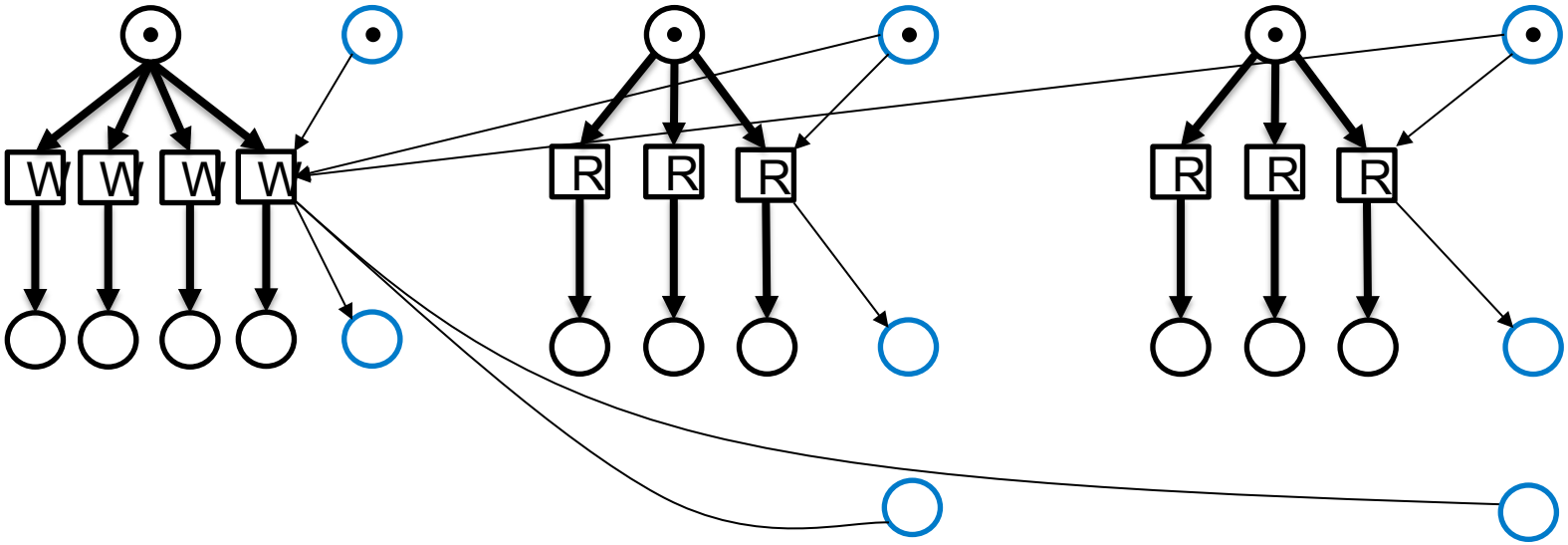
Another Example (Place Replication)

Global variables:
int x = 0;

Thread 1:
x = 5;

Thread 2:
local int a = x;

Thread 3:
local int b = x;



Requires four test executions to cover, as all writes are in conflict!
(most of the arcs and conditions are not shown to simplify the picture)

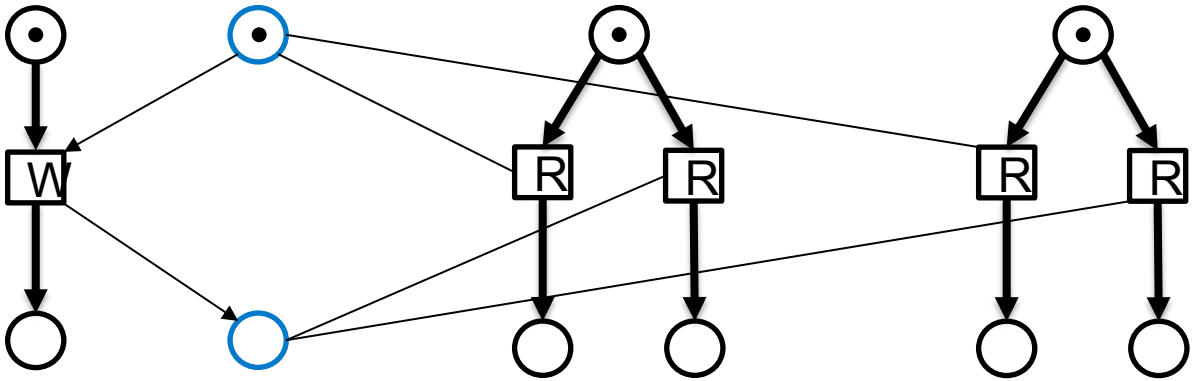
Another Example (Read Arcs)

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 subtantially more compact.
Only requires two test to be covered!

Experiments

	Unfolding		DPOR	
program	**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

	Unfolding		Contextual unfolding	
program	**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

Conclusions

- 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
- Ongoing work:
  - Encoding the unfolding as SMT formulas in order to check global properties of the program under test
  - Even more compact representations with read-arcs