Aalto University
School of Science

Using Unfoldings in Automated Testing of Multithreaded Programs

Kari Kähkönen, Olli Saarikivi, Keijo Heljanko

The Problem

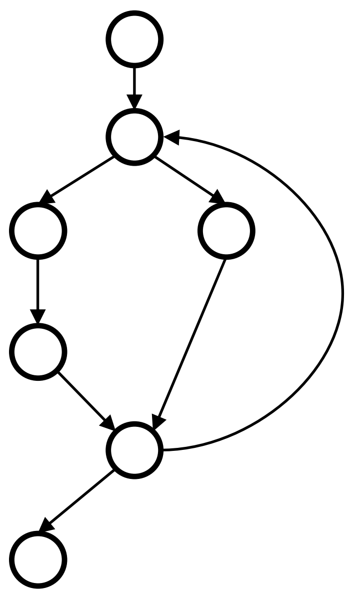
- How to automatically test the local state reachability in multithreaded programs
  - E.g., find assertion violations, uncaught exceptions, etc.
- The main challenge: path explosion and numerous interleavings of threads
- One approach: dynamic symbolic execution (DSE) + partial order reduction
- New approach: DSE + unfoldings

Dynamic Symbolic Execution

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

```
x = input
x = x + 5

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



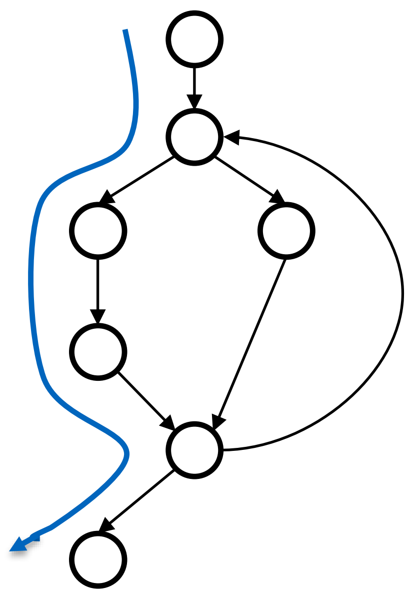
Control flow graph

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

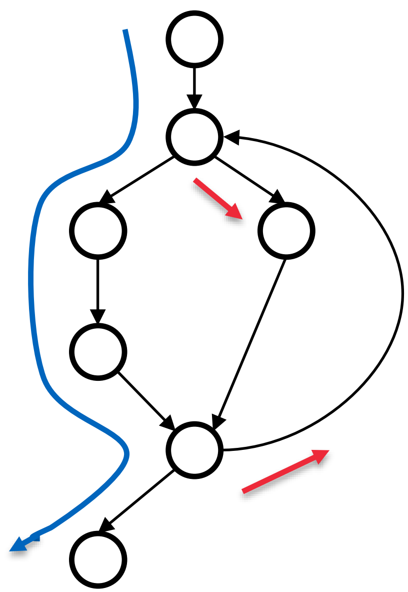
Aalto University
School of Science

Dynamic Symbolic Execution

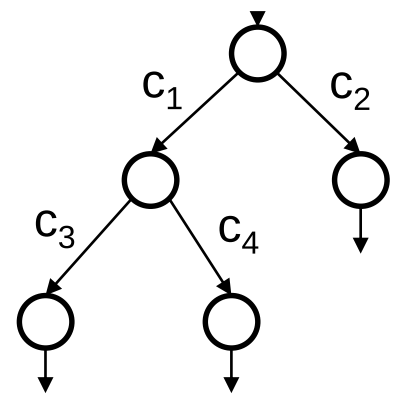
- Symbolic execution generates constraints that can be solved to obtain new test inputs for unexplored paths

```
x = input
x = x + 5

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



Control flow graph

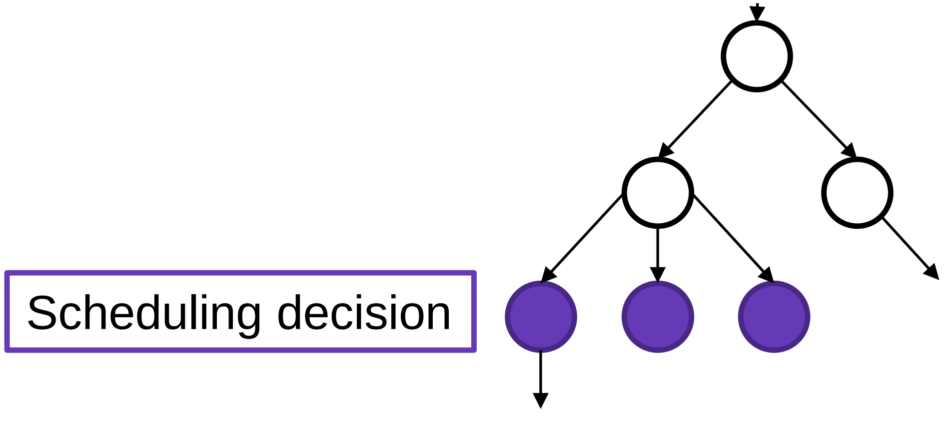


$c_1 = input_1 + 5 > 10$

$c_2 = input_1 + 5 \leq 10$

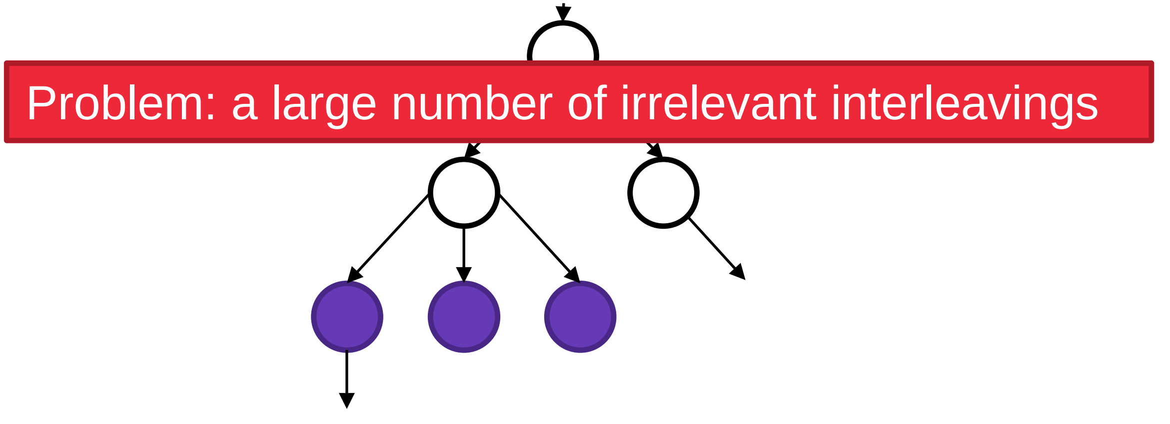
What about Multithreaded Programs?

- Take control of the scheduler
- Execute threads one by one until a global operation (e.g., access shared variable) is reached
- Branch the execution tree for each enabled operation



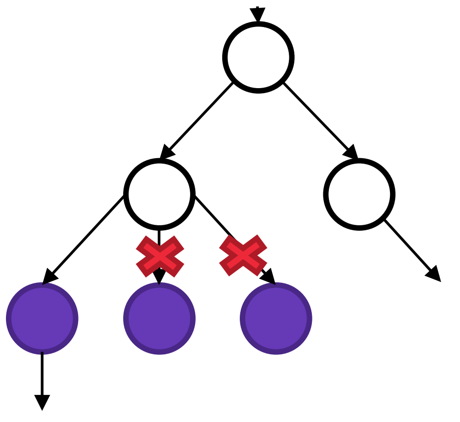
What about Multithreaded Programs?

- Take control of the scheduler
- Execute threads one by one until a global operation (e.g., access shared variable) is reached
- Branch the execution tree for each enabled operation



One Solution: Partial-Order Reduction

- Ignore provably irrelevant parts of the symbolic execution tree



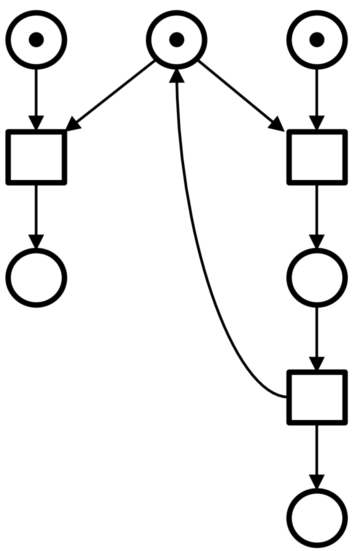
- Existing algorithms:
  - dynamic partial-order reduction
  - race detection and flipping

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

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

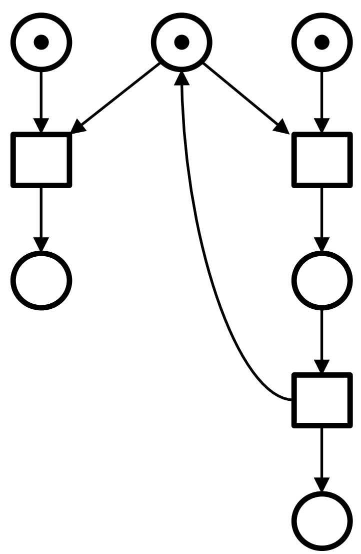


Petri net

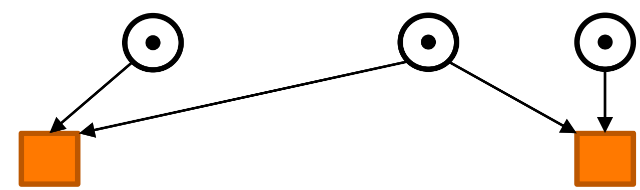
Initial unfolding

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



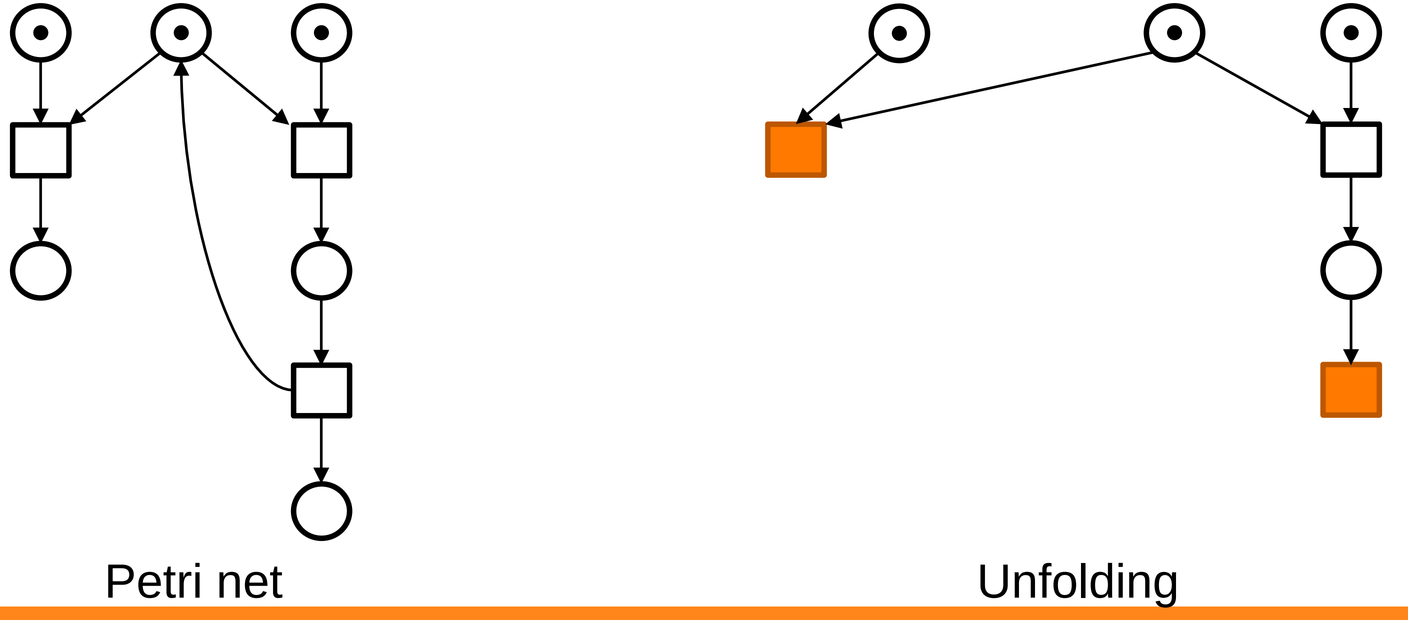
Petri net



Unfolding

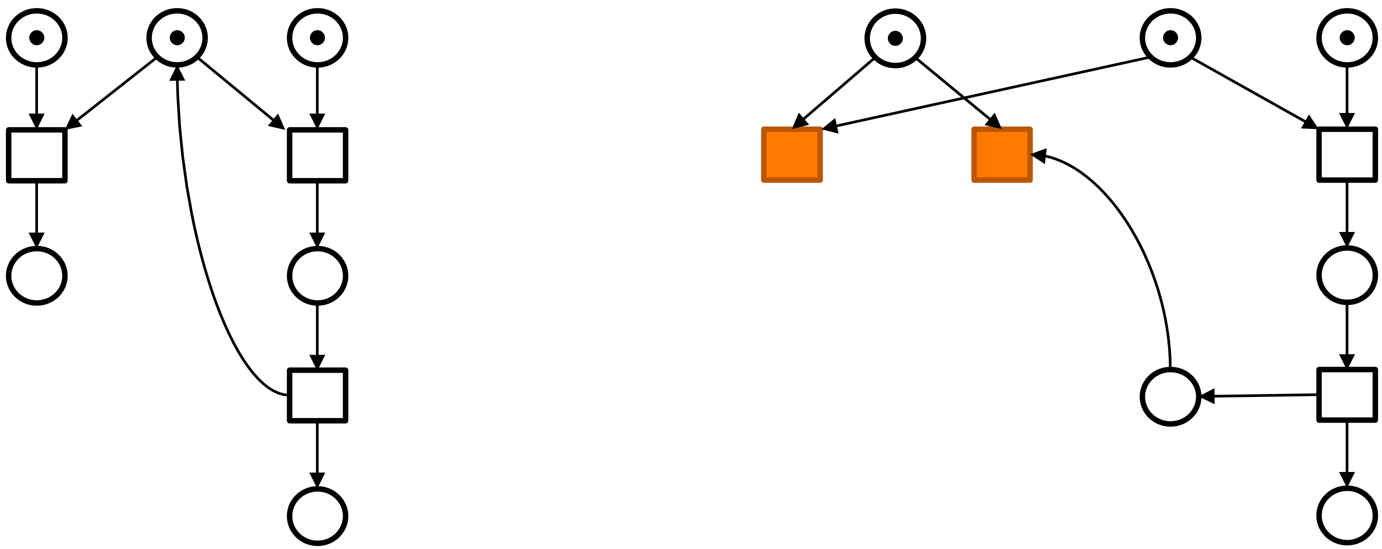
What Are Unfoldings?

- 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



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

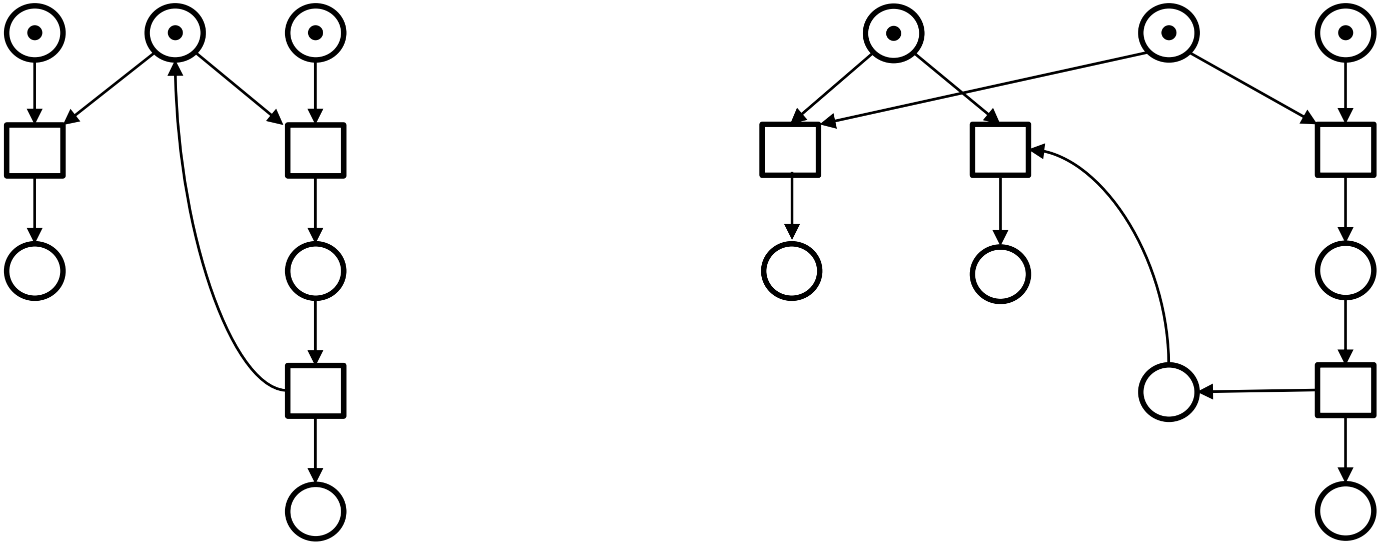


Petri net

Unfolding

What Are Unfoldings?

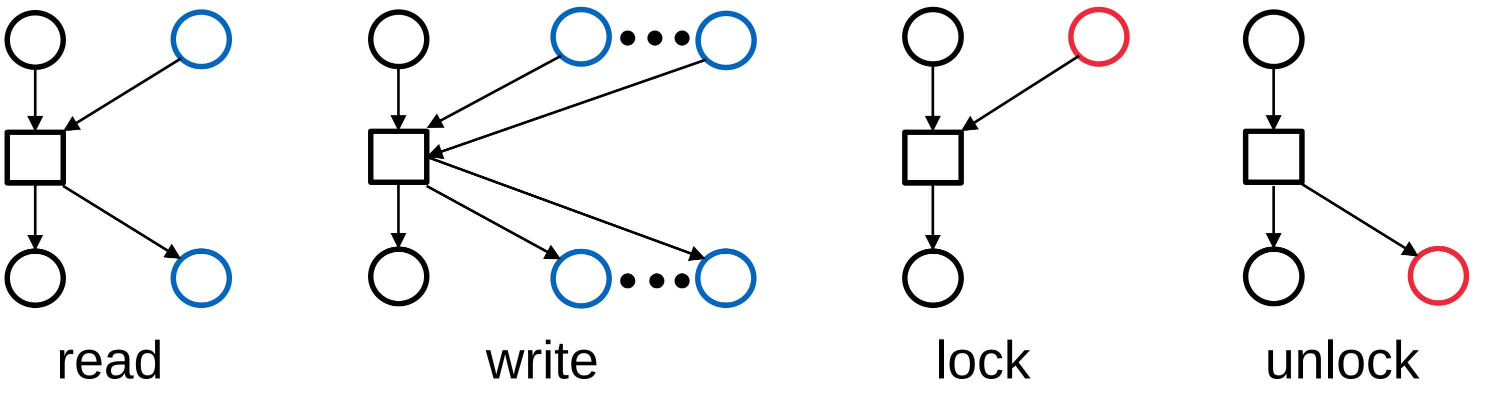
- Unwinding of a control flow graph is an execution tree
- Unwinding of a Petri net is an unfolding
- Can be exponentially more compact than exec. trees



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

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();
```

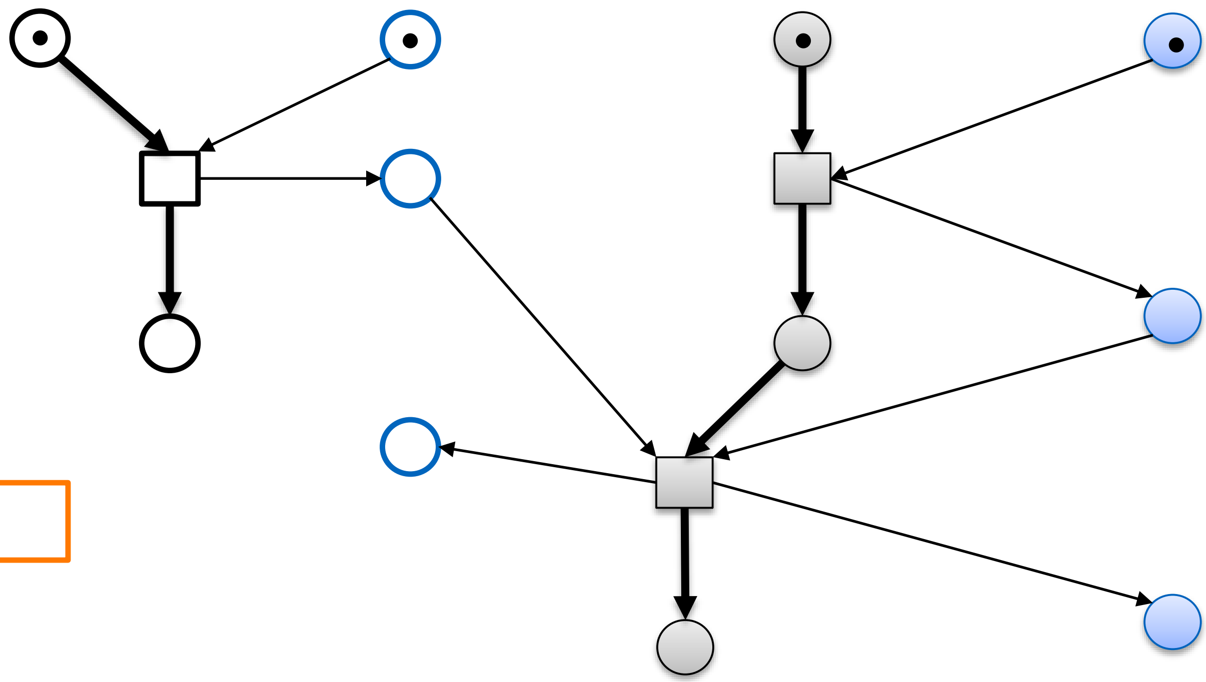
Initial unfolding

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();



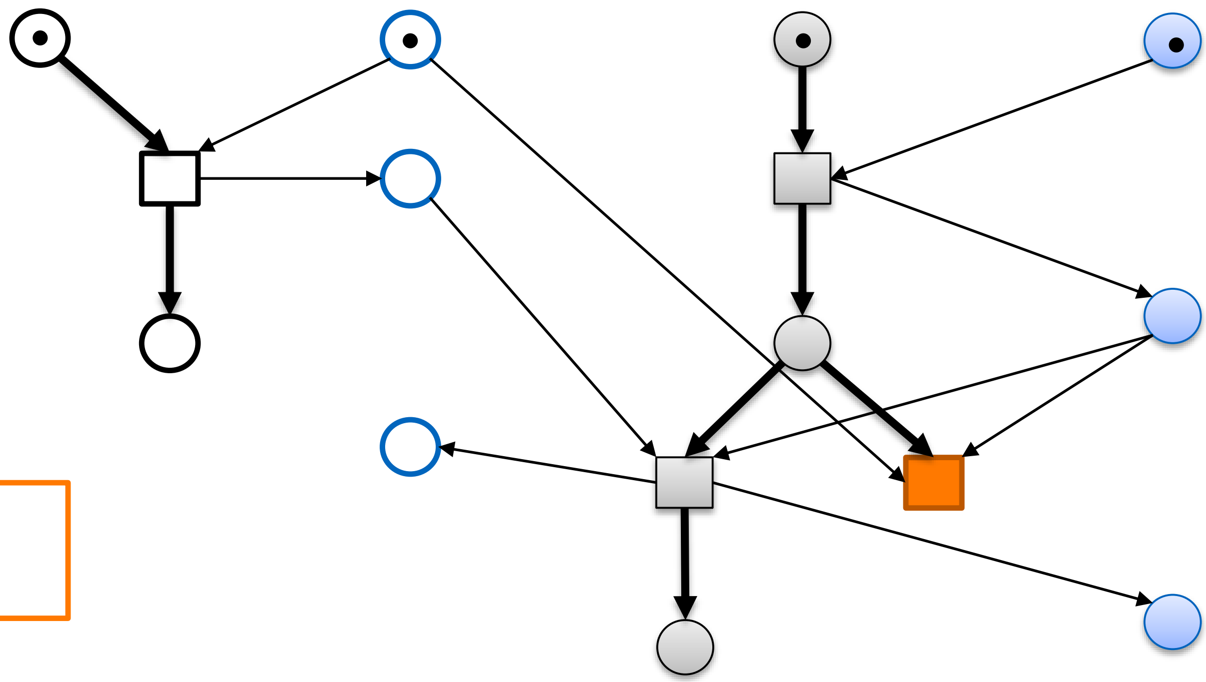
First test run

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();



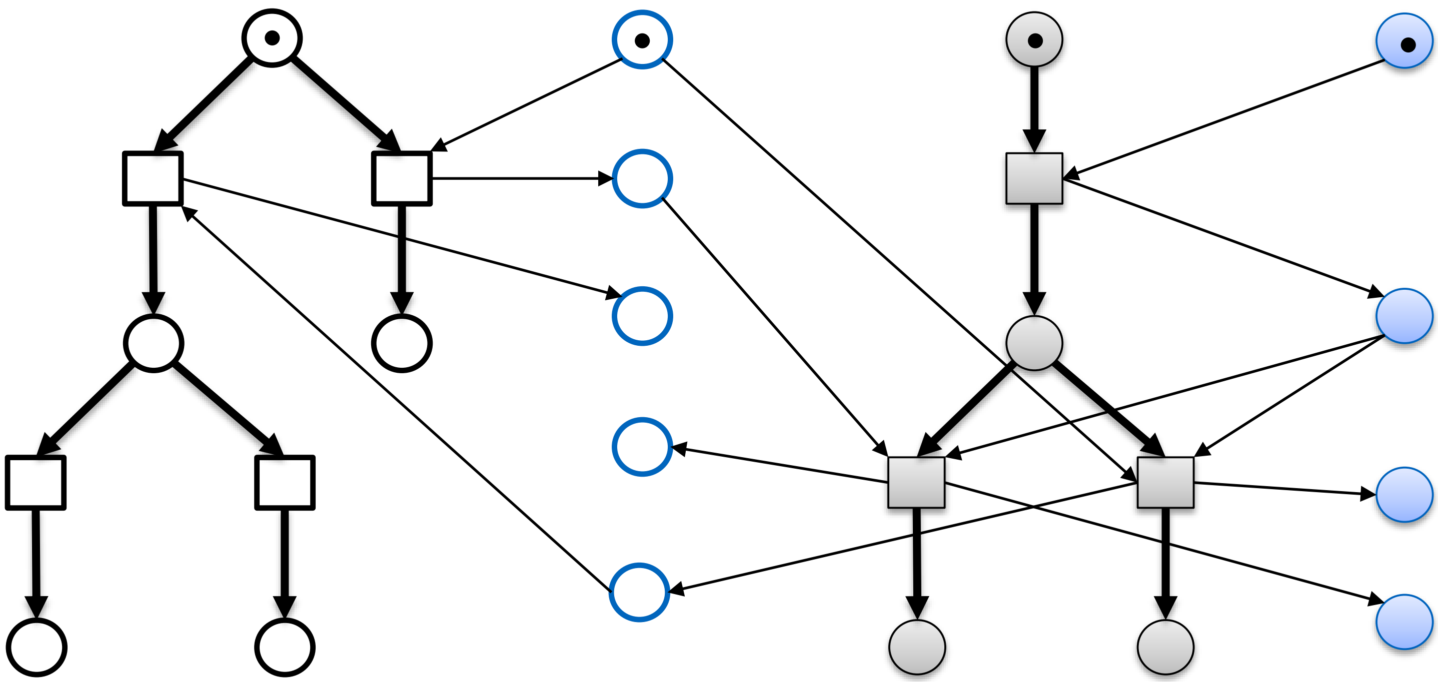
Find possible extensions

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
- It is possible to use existing potential extension algorithms with DSE
  - Designed for arbitrary Petri nets
  - Very expensive
- Key contribution: 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

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 is computationally more expensive per test run but typically requires less test runs
  - With threads that contains high amount of independence, the reduction to the number of test runs can be even exponential

Experiments

	Unfolding		DPOR (ACSD '12)		jCUTE
program	**paths**	**time**	**paths**	**time**	**paths**
Indexer (12)	**8**	**2**	85	10	**8**
Filesystem (16)	**3**	**0**	16	2	31
Filesystem (18)	**4**	**0**	97	6	2026
Parallel pi (5)	**120**	**3**	2698	17	**120**
Test selector (3)	**65**	**2**	87	**2**	**65**
Test selector (4)	**2576**	**70**	8042	97	**2576**
Pairs (6)	**7**	**0**	512	8	580
Locking (4)	**2520**	42	**2520**	**13**	**2520**
Synthetic-1 (3)	**984**	15	3716	**10**	2430
Synthetic-2 (3)	**1943**	**54**	7768	56	4860
Synthetic-3 (4)	**682**	**14**	8550	52	1757

Conclusions

- A new approach to test multithreaded programs by combining DSE and unfoldings
- The restricted form of the unfoldings allows efficient implementation of the algorithm
- The new algorithm offers competitive performance to existing approaches
  - In some cases it can be substantially faster