Aalto University
School of Science

Lightweight State Capturing for Automated Testing of Multithreaded Programs

Kari Kähkönen and Keijo Heljanko

The Main Goal

- How to cover reachable local states in multithreaded programs that read input values (e.g., find assertion violations)
- In principle easy: test each input value combination together with all thread interleavings

```
Thread 1:
a = input();
if (a > 100) {
   X = 7;
   …
}
…
```

```
Thread 2:
b = X;
c = input();
while (b != 7) {
   …
}
…
```

The Main Goal

- How to cover reachable local states in multithreaded programs that read input values (e.g., find assertion violations)
- In principle easy: test each input value combination together with all thread interleavings

```
Thread 1:                    Thread 2:
a = input();                 b = X;
if (a > 100) {               c = input();
   …                            …
}                            }
…                            …
```

Infeasible to explicitly test all combinations

One Approach

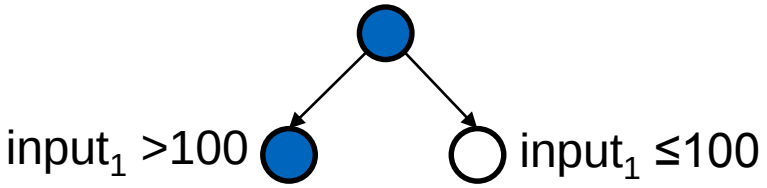
- Use dynamic symbolic execution to avoid testing irrelevant input values
- Use partial order reduction methods to avoid exploring irrelevant interleavings of threads

Example

```
Thread 1:
a = input();
if (a > 100) {
   b = X;
…

Thread 2:
c = input();
if (c != 5)
    d = X;
…
```

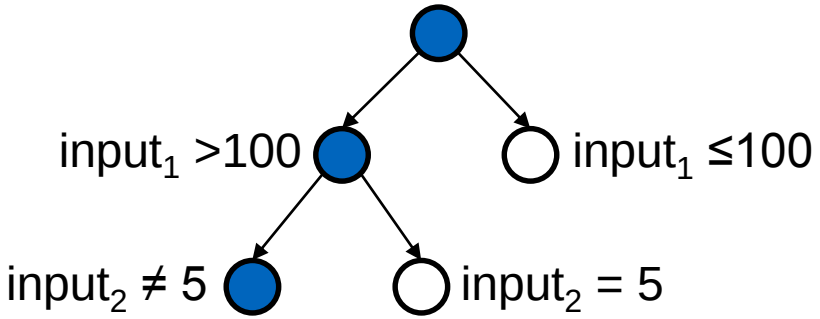
Example

```
Thread 1:
a = input(); // = 412
if (a > 100) {
    b = X;
…

Thread 2:
c = input();
if (c != 5)
     d = X;
…
```



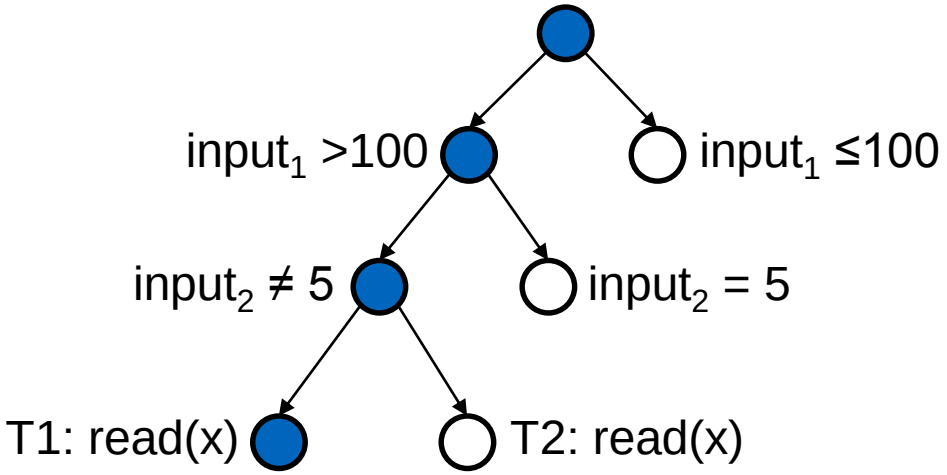
Example

```
Thread 1:
a = input(); // = 412
if (a > 100) {
    b = X;
…

Thread 2:
c = input(); // = 0
if (c != 5)
     d = X;
…
```



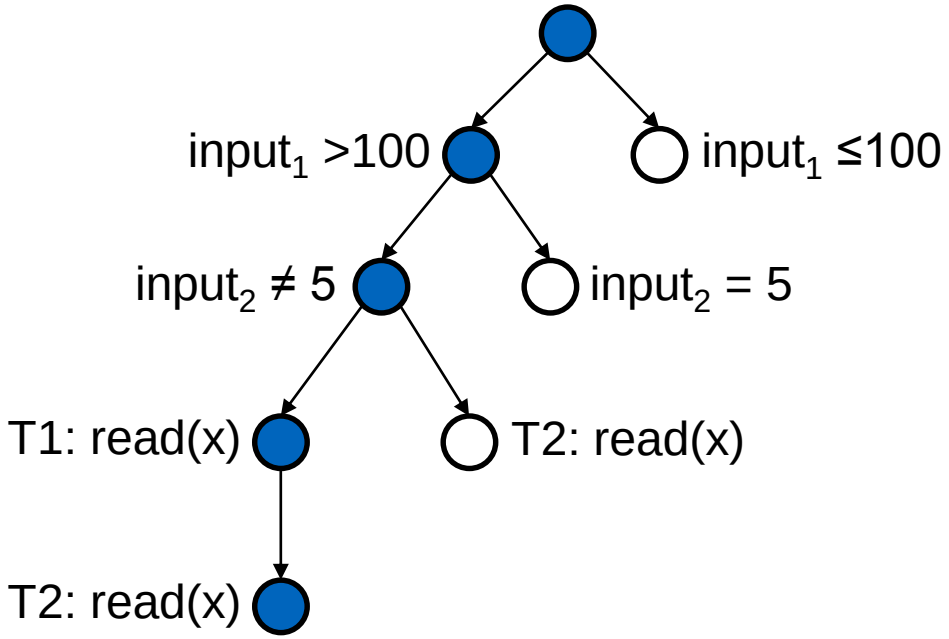
Example

```
Thread 1:
a = input(); // = 412
if (a > 100) {
   b = X;
…

Thread 2:
c = input(); // = 0
if (c != 5)
    d = X;
…
```



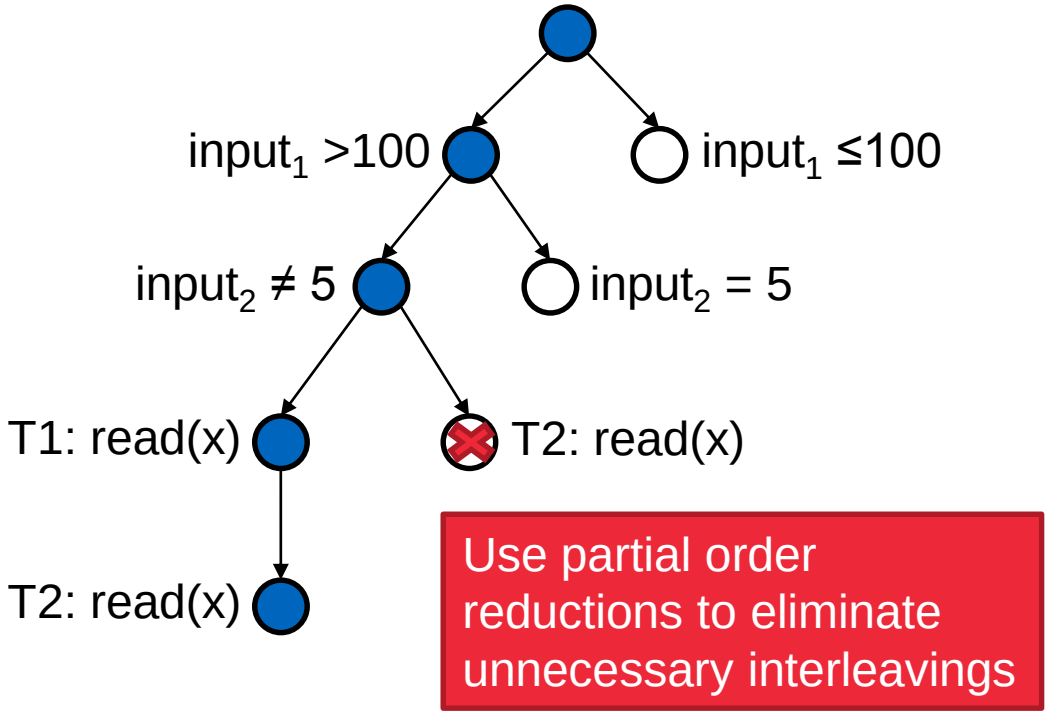
Example

```
Thread 1:
a = input(); // = 412
if (a > 100) {
   b = X;
…

Thread 2:
c = input(); // = 0
if (c != 5)
    d = X;
…
```



Example

```
Thread 1:
a = input(); // = 412
if (a > 100) {
   b = X;
…

Thread 2:
c = input(); // = 0
if (c != 5)
    d = X;
…
```



The Problem

- Typical partial order reduction approaches explore all interleavings of dependent state transitions
  - Can sometimes lead to unnecessary test executions

```
Thread 1:
acquire(lock1);
X = 1;
release(lock1);
…
```

```
Thread 2:
acquire(lock1);
Y = 1;
release(lock1);
…
```

The Problem

- Typical partial order reduction approaches explore all interleavings of dependent state transitions
  - Can sometimes lead to unnecessary test executions

```
Thread 1:                Thread 2:
acquire(lock1);          acquire(lock1);
X = 1;                   Y = 1;
release(lock1);          release(lock1);
…                        …
```

Both ways to interleave the executions lead to the same state

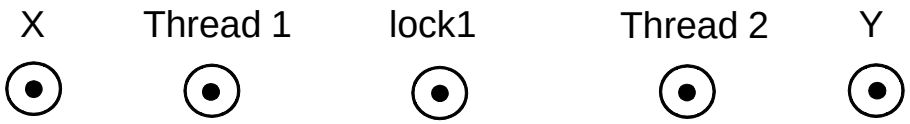
Solution: Capture and Match States

- Capturing concrete states of programs can be expensive
- Symbolic state matching can require expensive solver calls
  - E.g., symbolic states resulting from dynamic symbolic execution
- The approach in this paper:
  - Model test executions as a Petri net
  - Use the model to determine when a previously visited state is encountered

Model Example

```
Thread 1:
acquire(lock1);
X = 1;
release(lock1);
```

```
Thread 2:
acquire(lock1);
Y = 1;
release(lock1);
```

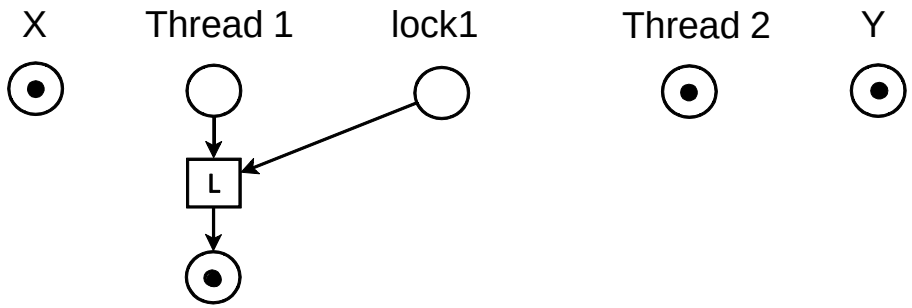


- Local states, shared variables and locks are represented as places
- A marking == an abstract representation of a program state
- The initial state of the program is illustrated above

Model Example

```
Thread 1:
acquire(lock1);
X = 1;
release(lock1);
```

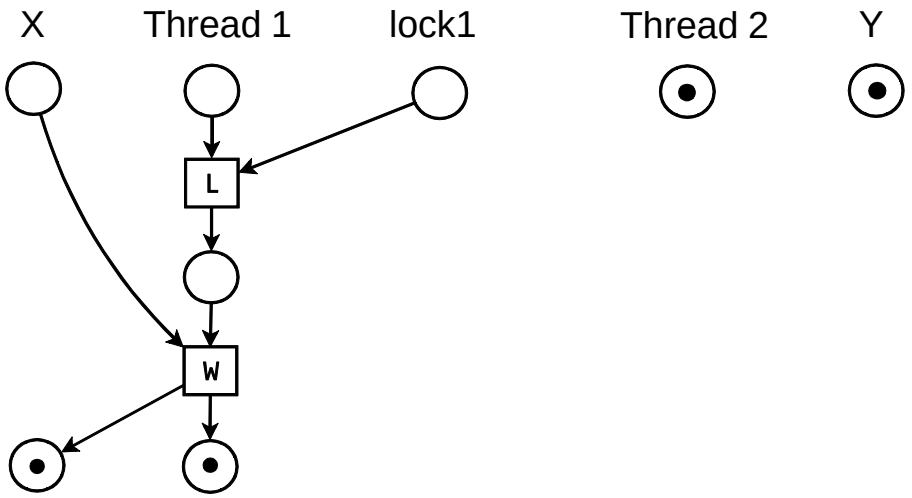
```
Thread 2:
acquire(lock1);
Y = 1;
release(lock1);
```



Model Example

```
Thread 1:
acquire(lock1);
X = 1;
release(lock1);
```

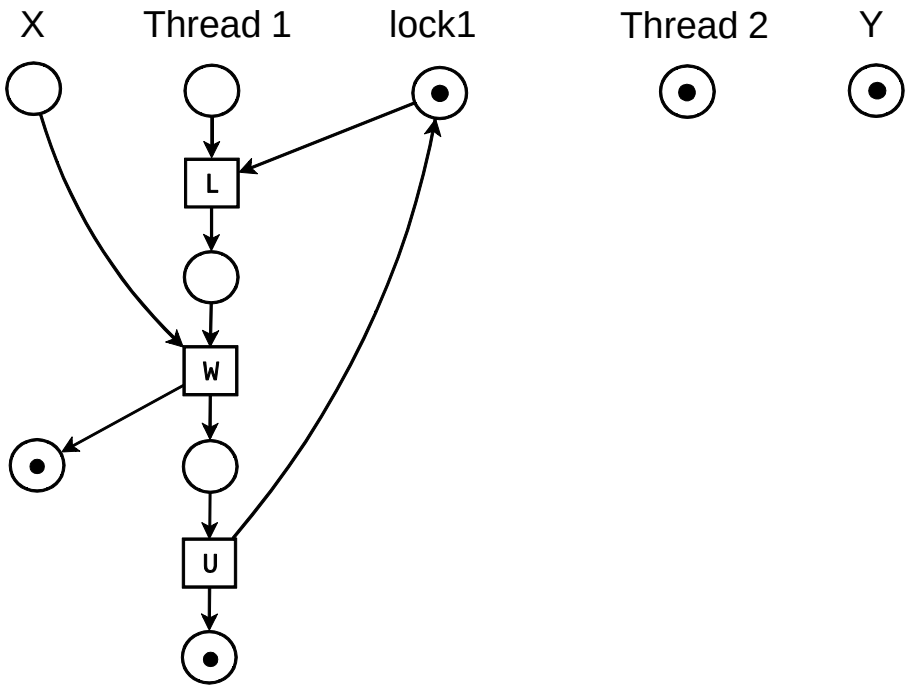
```
Thread 2:
acquire(lock1);
Y = 1;
release(lock1);
```



Model Example

```
Thread 1:
acquire(lock1);
X = 1;
release(lock1);
```

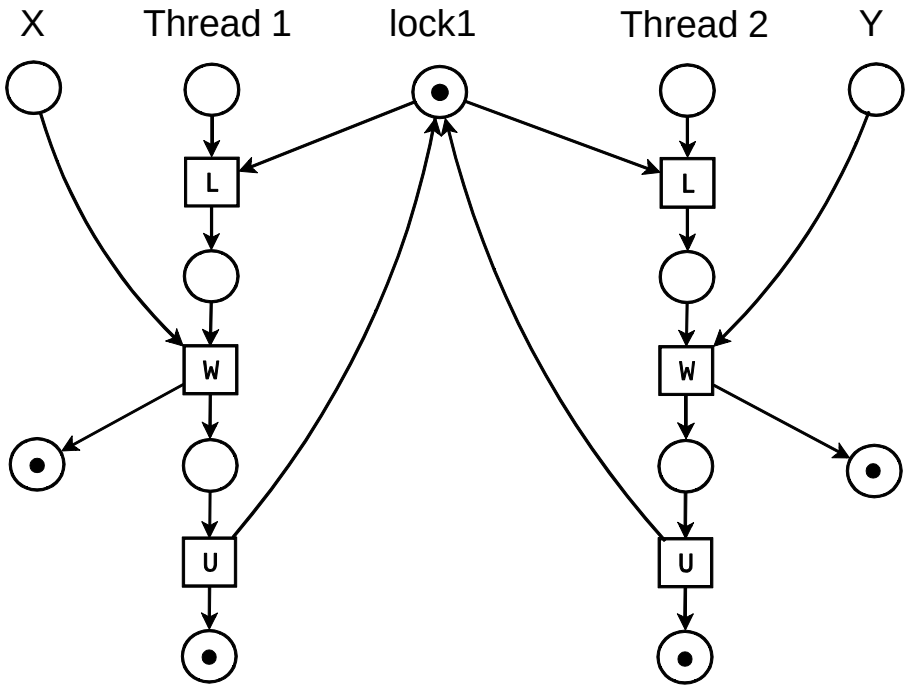
```
Thread 2:
acquire(lock1);
Y = 1;
release(lock1);
```



Model Example

```
Thread 1:
acquire(lock1);
X = 1;
release(lock1);
```

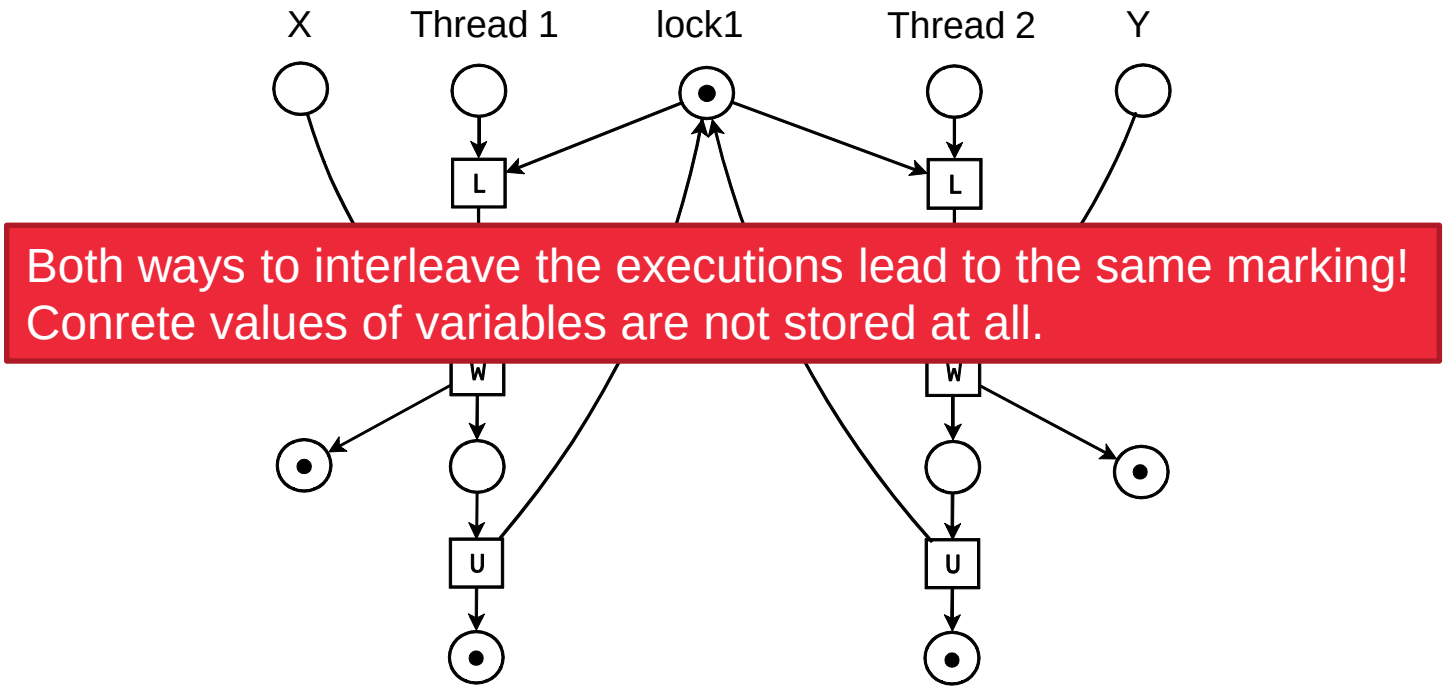
```
Thread 2:
acquire(lock1);
Y = 1;
release(lock1);
```



Model Example

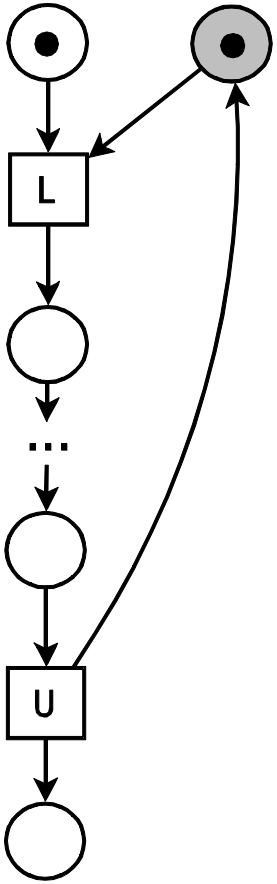
```
Thread 1:
acquire(lock1);
X = 1;
release(lock1);
```

```
Thread 2:
acquire(lock1);
Y = 1;
release(lock1);
```

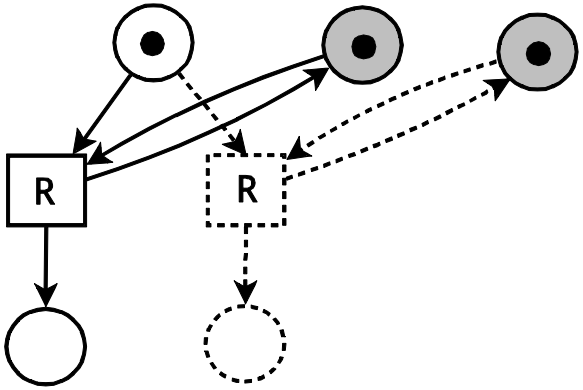


Modeling Constructs

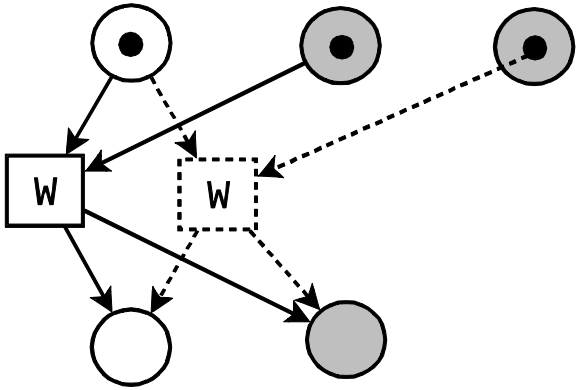
Locking and unlocking



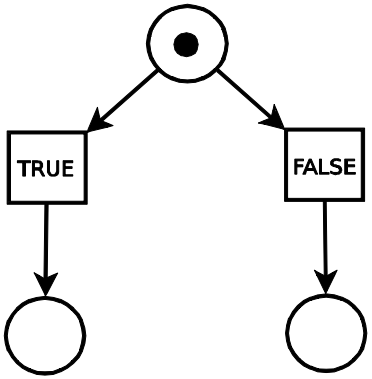
Reading a shared variable



Writing to a shared variable



Symbolic branching

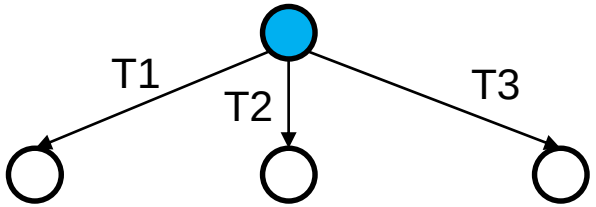


Automated Testing

- Model a random test execution
- Systematically explore the states of the model by unwinding it (into a tree or an acyclic Petri net)
- Store visited markings and cut the state space exploration if the same marking is encountered again
- If the model is incomplete at some state, perform a test execution to extend the model and return to step 2

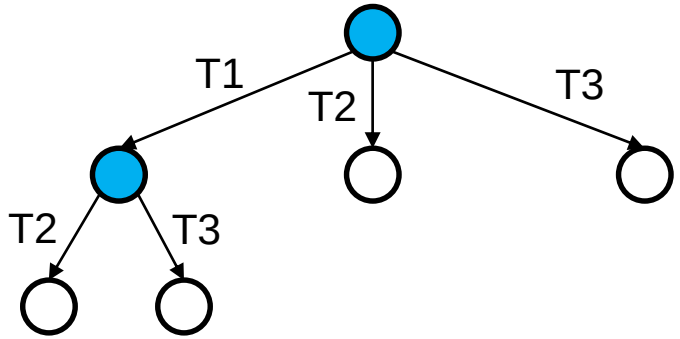
Example

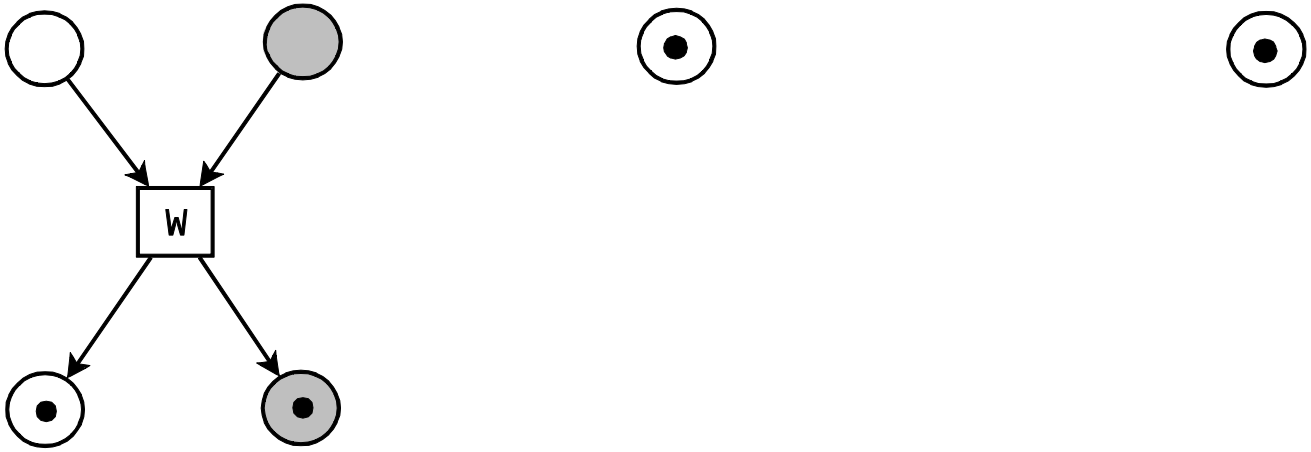
```
Thread 1:   Thread 2:   Thread 3:
X = 1;      X = 2;      X = 3;
```



Example

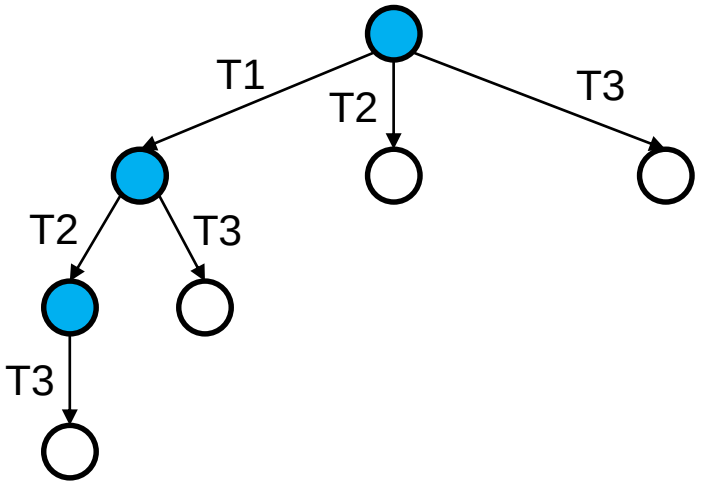
```
Thread 1:   Thread 2:   Thread 3:
X = 1;      X = 2;      X = 3;
```

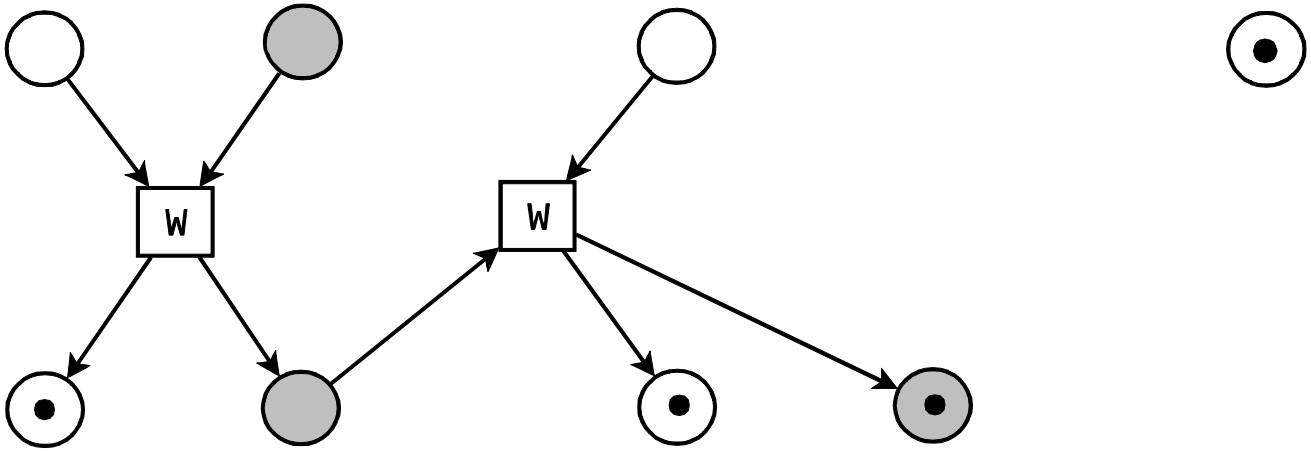




Example

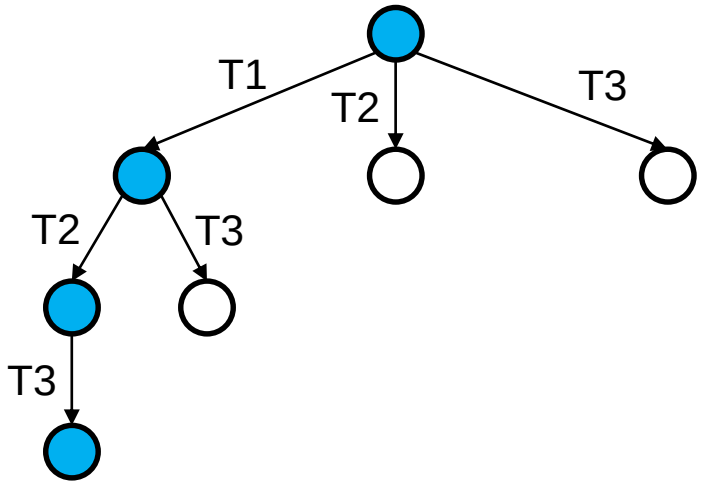
```
Thread 1:   Thread 2:   Thread 3:
X = 1;      X = 2;      X = 3;
```

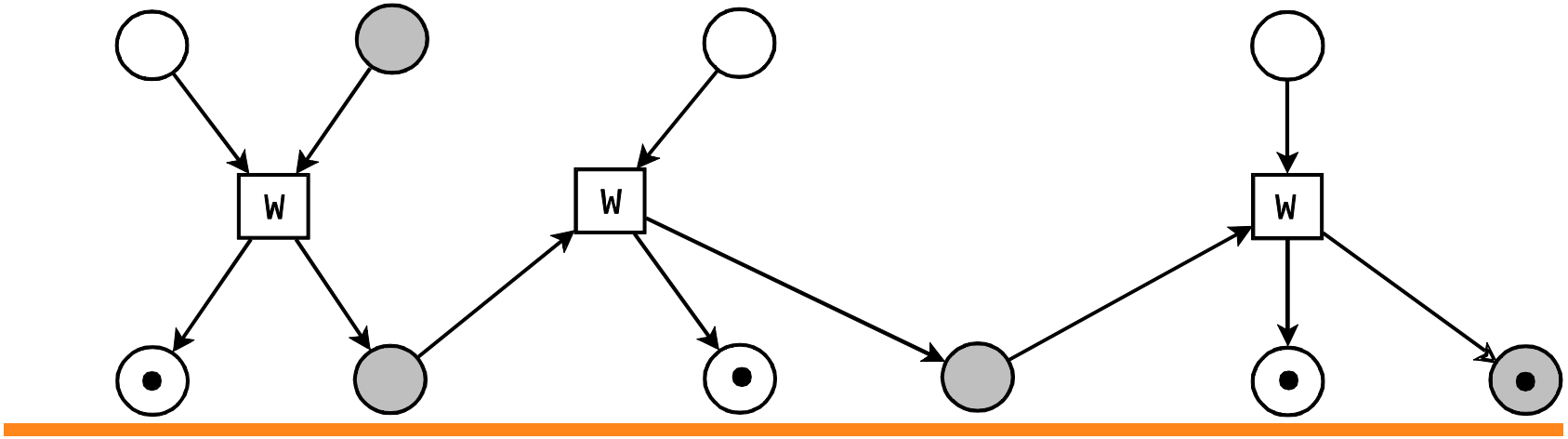




Example

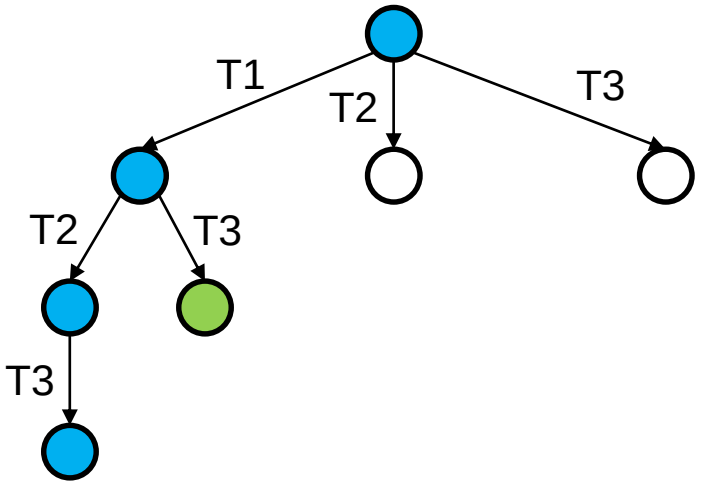
```
Thread 1:   Thread 2:   Thread 3:
X = 1;      X = 2;      X = 3;
```



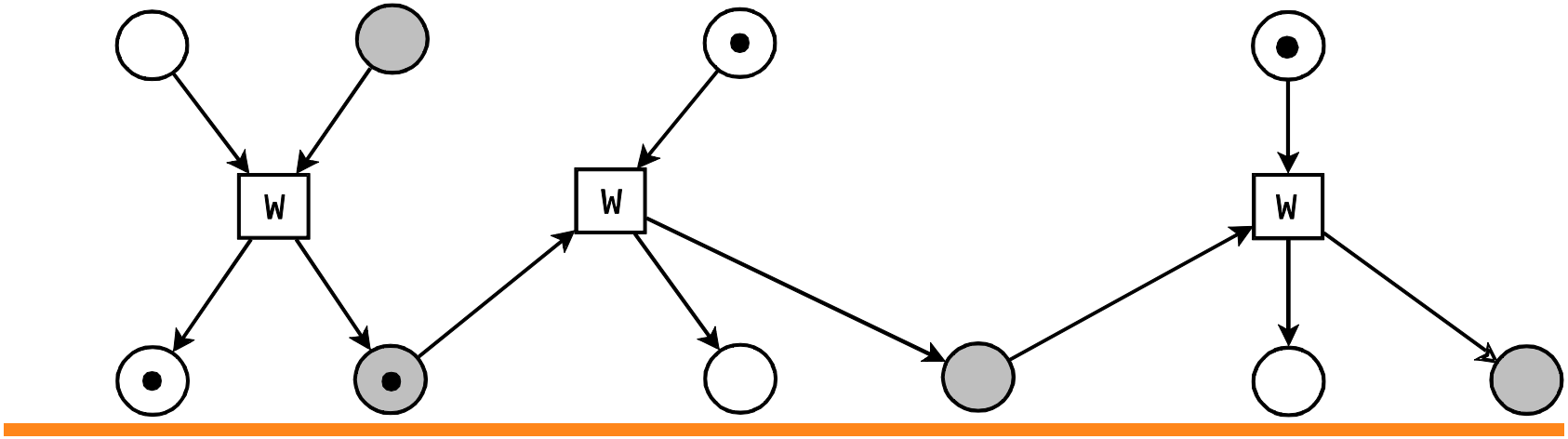


Example

```
Thread 1:   Thread 2:   Thread 3:
X = 1;      X = 2;      X = 3;
```

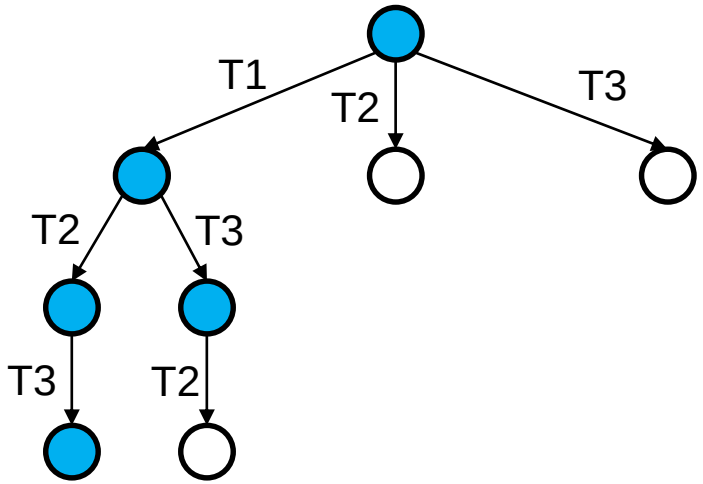


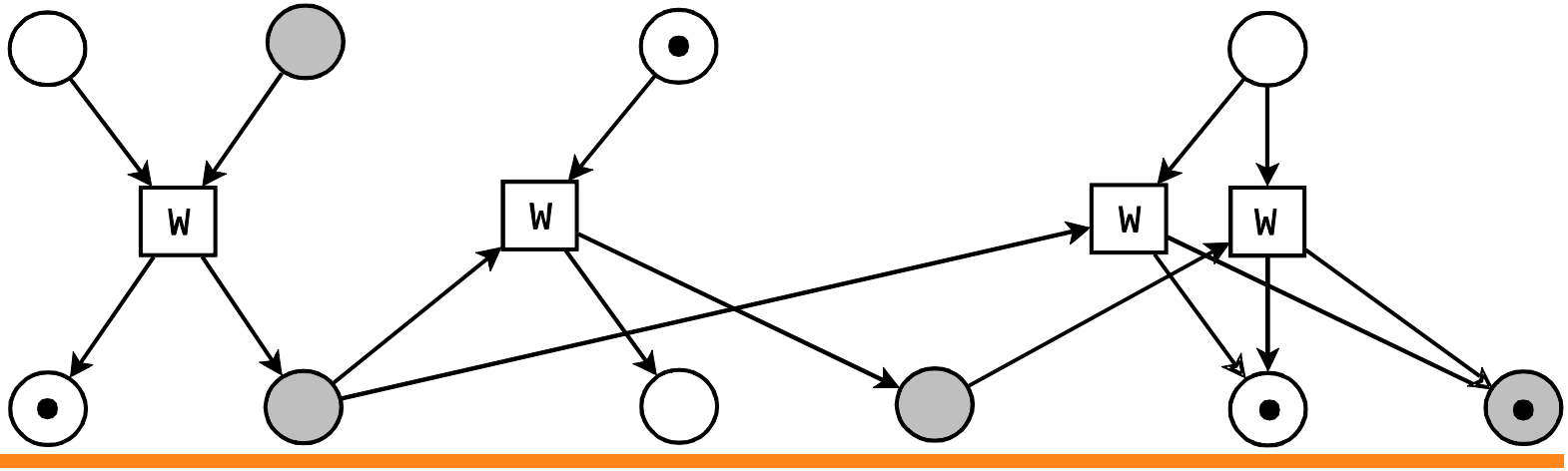
- Transition for thread 3 is missing
- Model can be extended by performing test execution (T1, T3, …)
- In this case the missing transition can also be predicted from the model!



Example

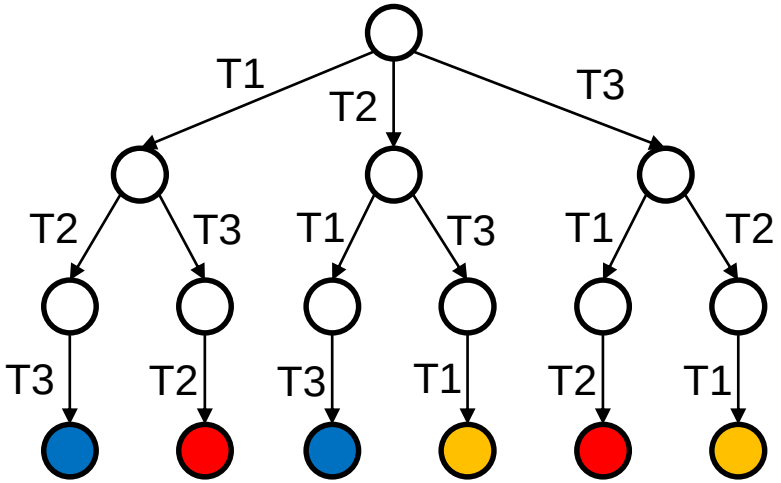
```
Thread 1:   Thread 2:   Thread 3:
X = 1;      X = 2;      X = 3;
```

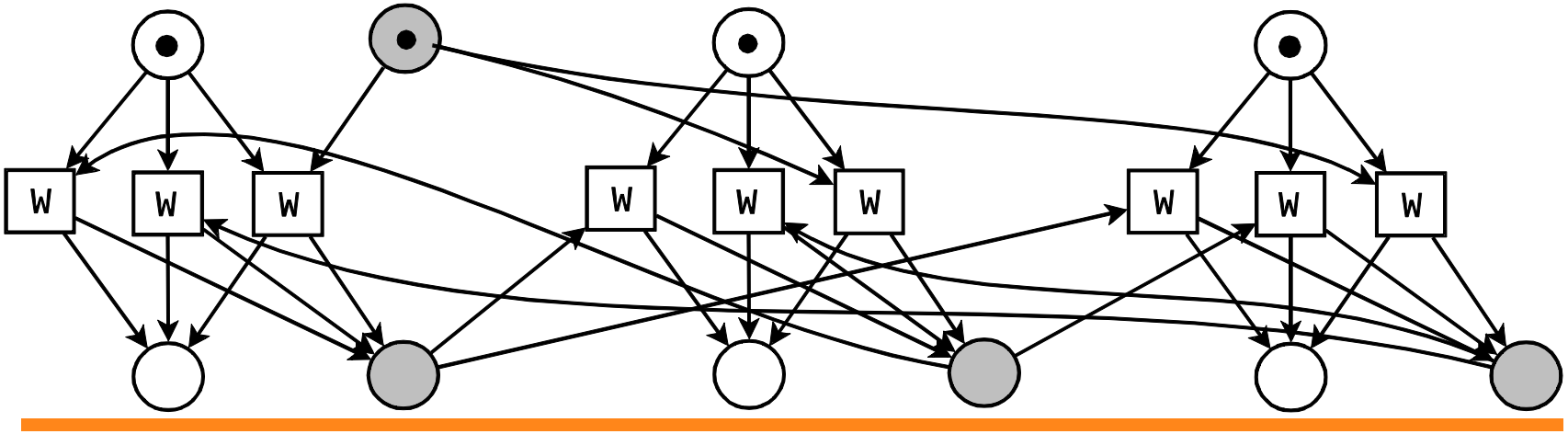




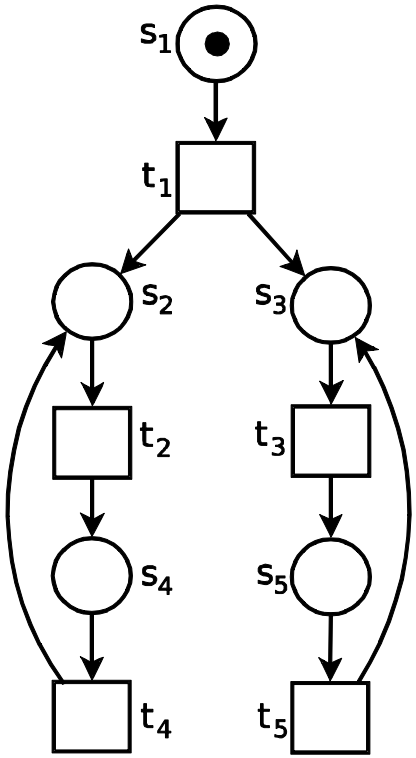
Example

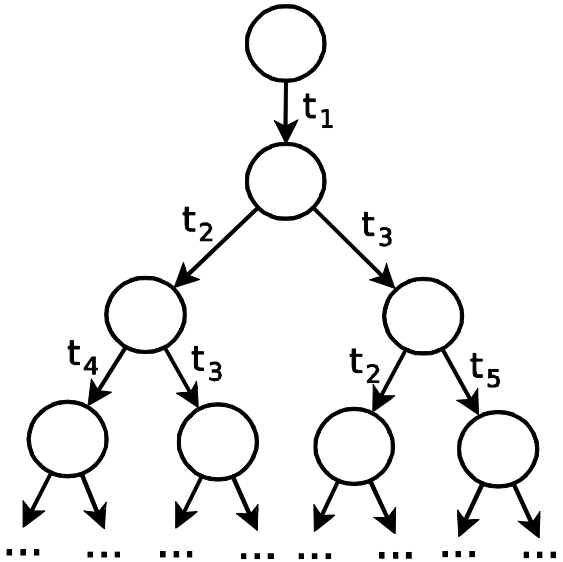
```
Thread 1:   Thread 2:   Thread 3:
X = 1;      X = 2;      X = 3;
```

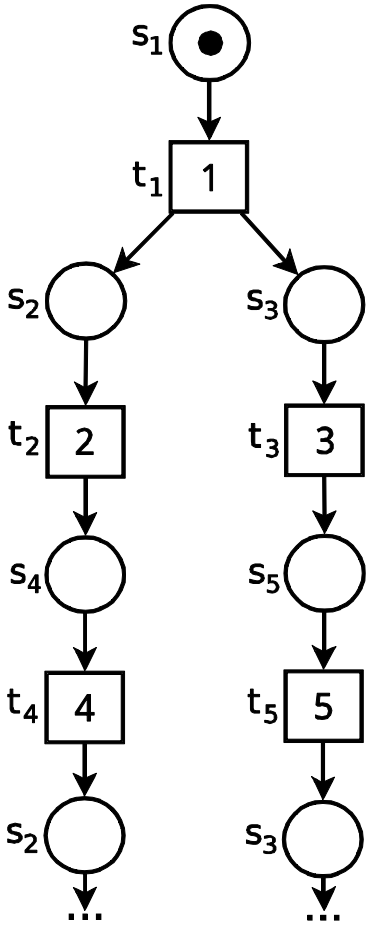




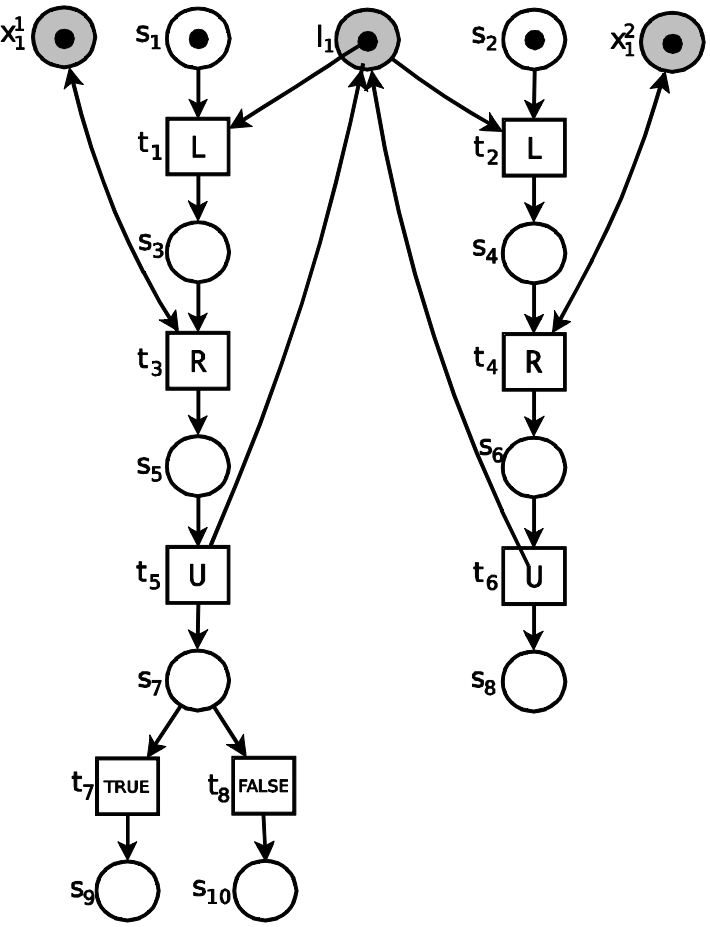
An Alternative to Trees: Unfoldings

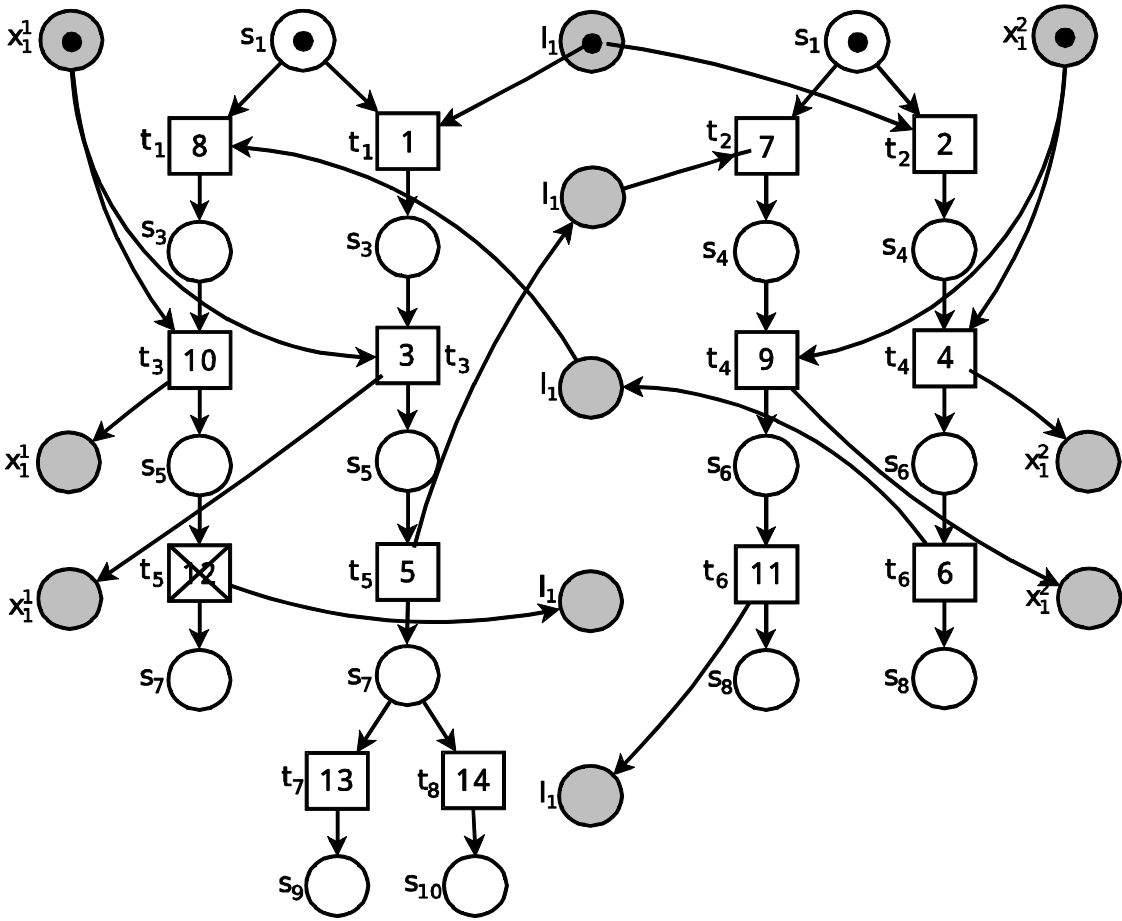






Unfolding Example





Experiments

	Stateless unf.		DPOR		Stateful tree		Stateful unf.	
program	**tests**	**time**	**tests**	**time**	**tests**	**time**	**tests**	**time**
Fib 1	19605	0m 17s	21102	0m 21s	5746	**0m 11s**	4946	0m 15
Fib 2	218243	4m 18s	232531	4m 2s	53478	3m 45s	46829	**3m 15s**
File 2	3	**0m 0s**	2227	0m 46s	-	> 30m	3	**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**
Locking 2	22680	0m 56s	22680	0m 47s	29	**0m 2s**	26	0m 9s
Locking 3	-	> 30m	-	> 30m	115	**0m 21s**	89	3m 32s
Szymanski	65138	2m 3s	65138	**0m 30s**	50264	0m 43s	46679	2m 35s
Writes	**-**	> 30m	-	> 30m	1	**0m 0s**	1	**0m 0s**

Conclusions and Future Work

- Lightweight state capturing based on modeling behaviour encountered during test executions
  - Additional tests are used to extend the model
- Can be combined with dynamic symbolic execution and partial order reduction approaches
- Future: it is possible to make the model more succint
  - Track concrete values of shared variables
  - Model special cases such as wait/notify loops