Concurrency Strategy Adaptation using Learning State Machines

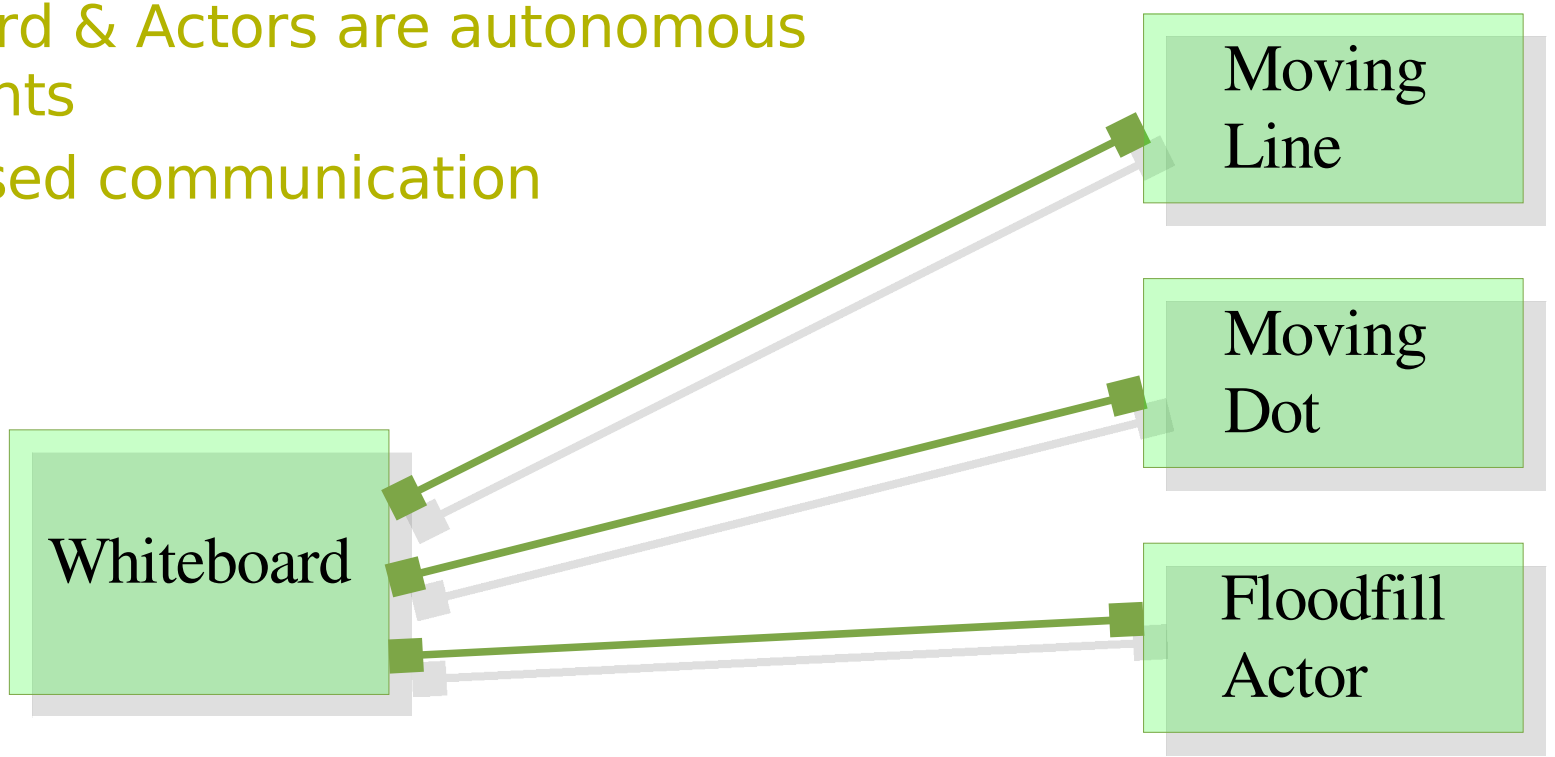
Werner Van Belle
werner.van.belle@gmail.com
werner@onlinux.be

Concurrency Strategy Conflicts

Whiteboard Example

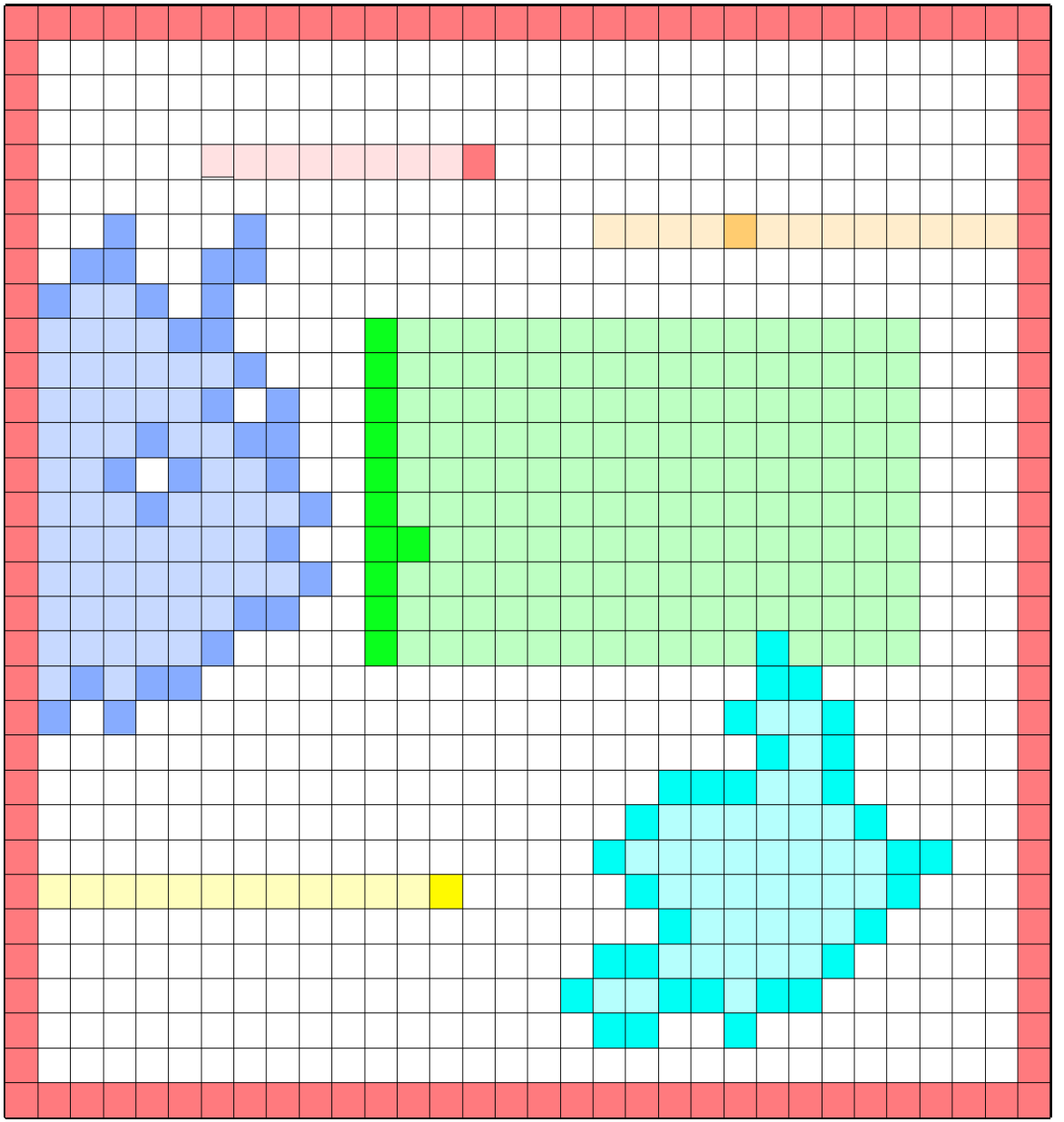
- Open Systems
  - The Internet
  - Peer to peer computational networks
  - Mobile embedded systems

- Whiteboard & Actors are autonomous components
- Event Based communication



Case Study

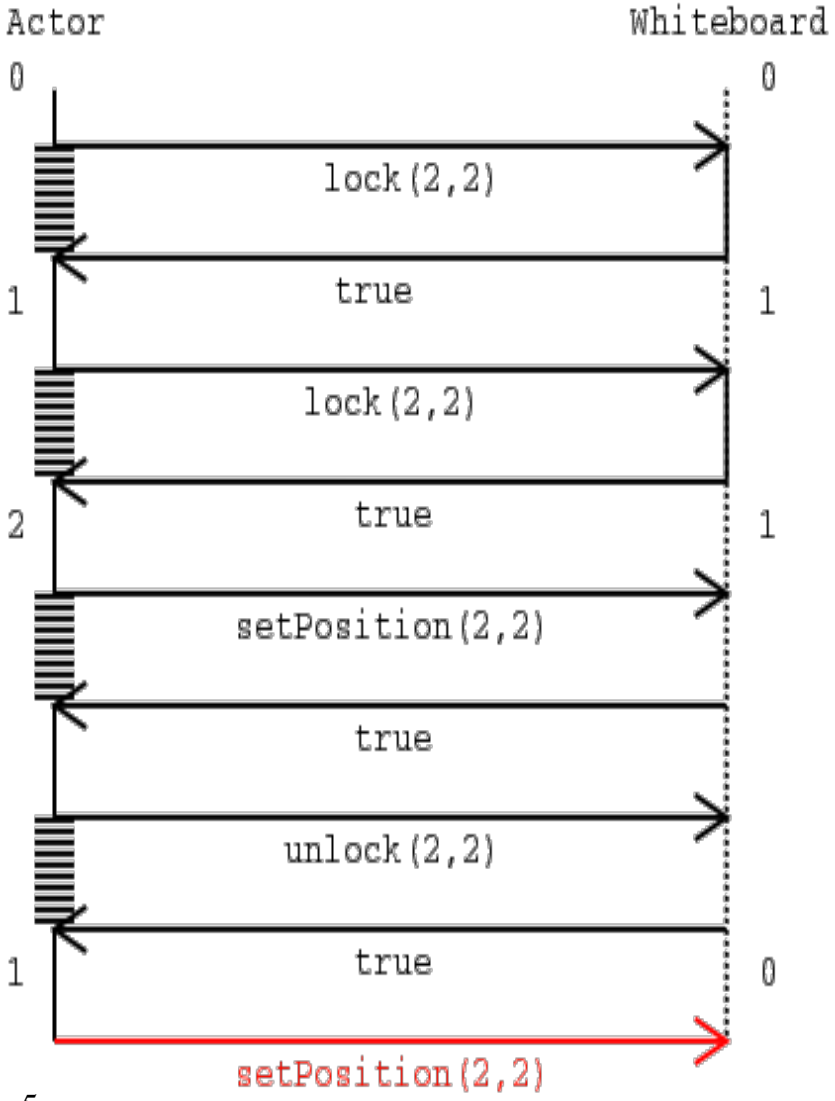
- Whiteboard
- Implicit Agreement
  - Different actors can set positions on board
  - Actors should not cross each others boundaries



Requires
Concurrency Strategy

Concurrency Interface Conflict

- Components/Agents/Processes/Actors can be manufactured by different authorities



Whiteboard Provides

bool lock(x, y)

-- binary semaphore

Client Requires/Assumes

bool lock(x, y)

-- counting semaphore

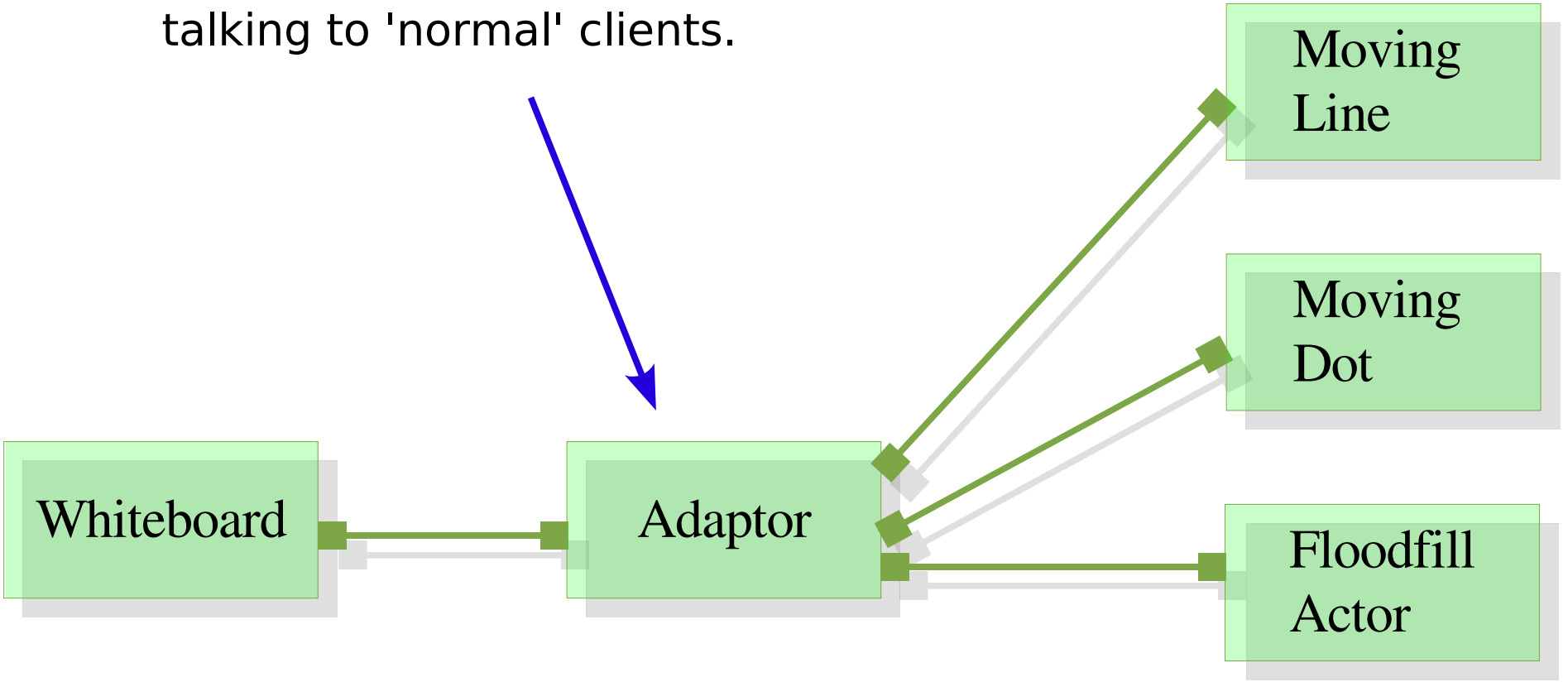
Results in

Concurrency Strategy Conflicts

Whiteboard Example

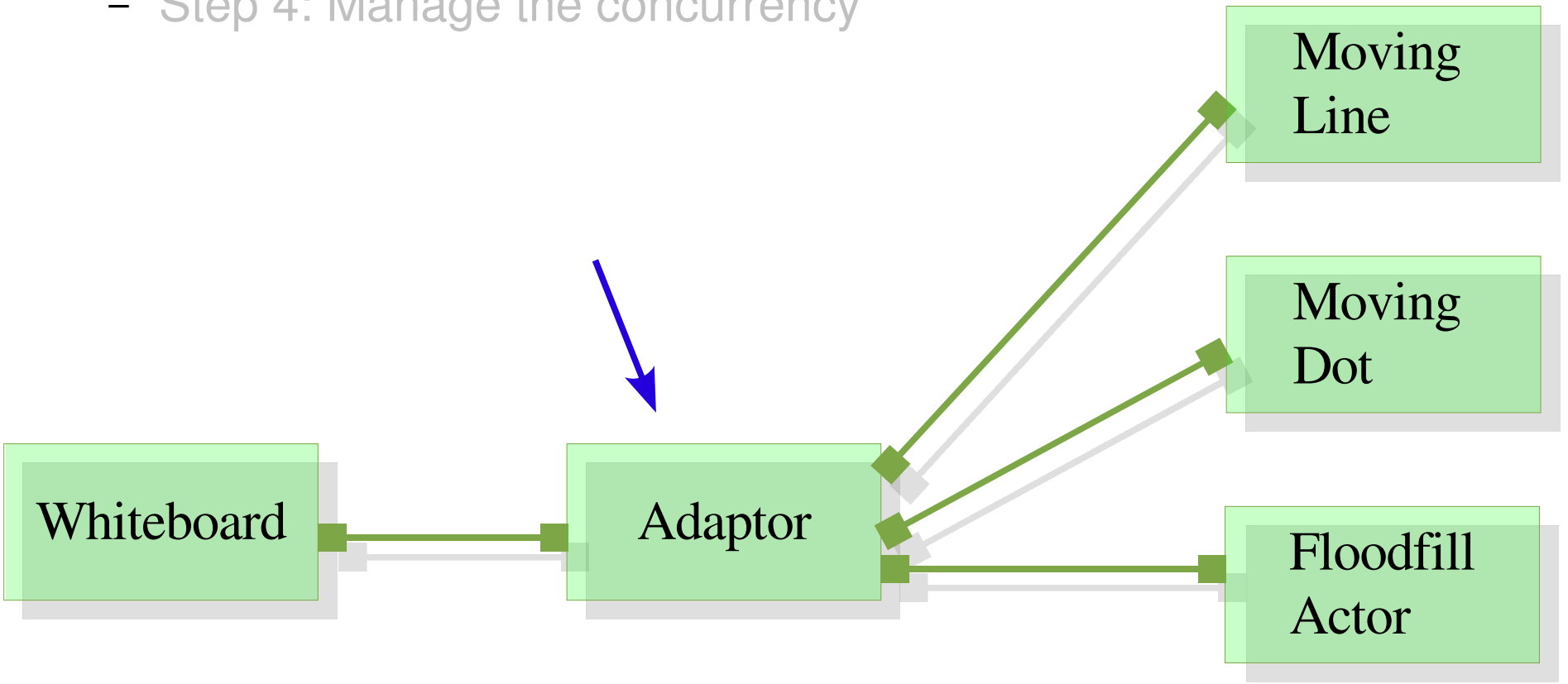
- Insertion of an Adapter

Mediates differences between components such that clients thinks they are talking to a 'normal' server & the server thinks it is talking to 'normal' clients.



Whiteboard Example

- Insertion of an Adapter
  - Step 1: Formal documented API
  - Step 2: Learn what the client wants to achieve
  - Step 3: Figure out how to bring the server in a specific state
  - Step 4: Manage the concurrency



Step 1 - **Formal API Description using Colored Petri-nets**

An API

Provided interface

```
bool Lock(int resource);
void unlock(int resource);
```

Required interface

??

Events

Provided interface

```
in Lock(int resource)
out LockTrue
out LockFalse
in Unlock(int resource)
out UnlockDone
```

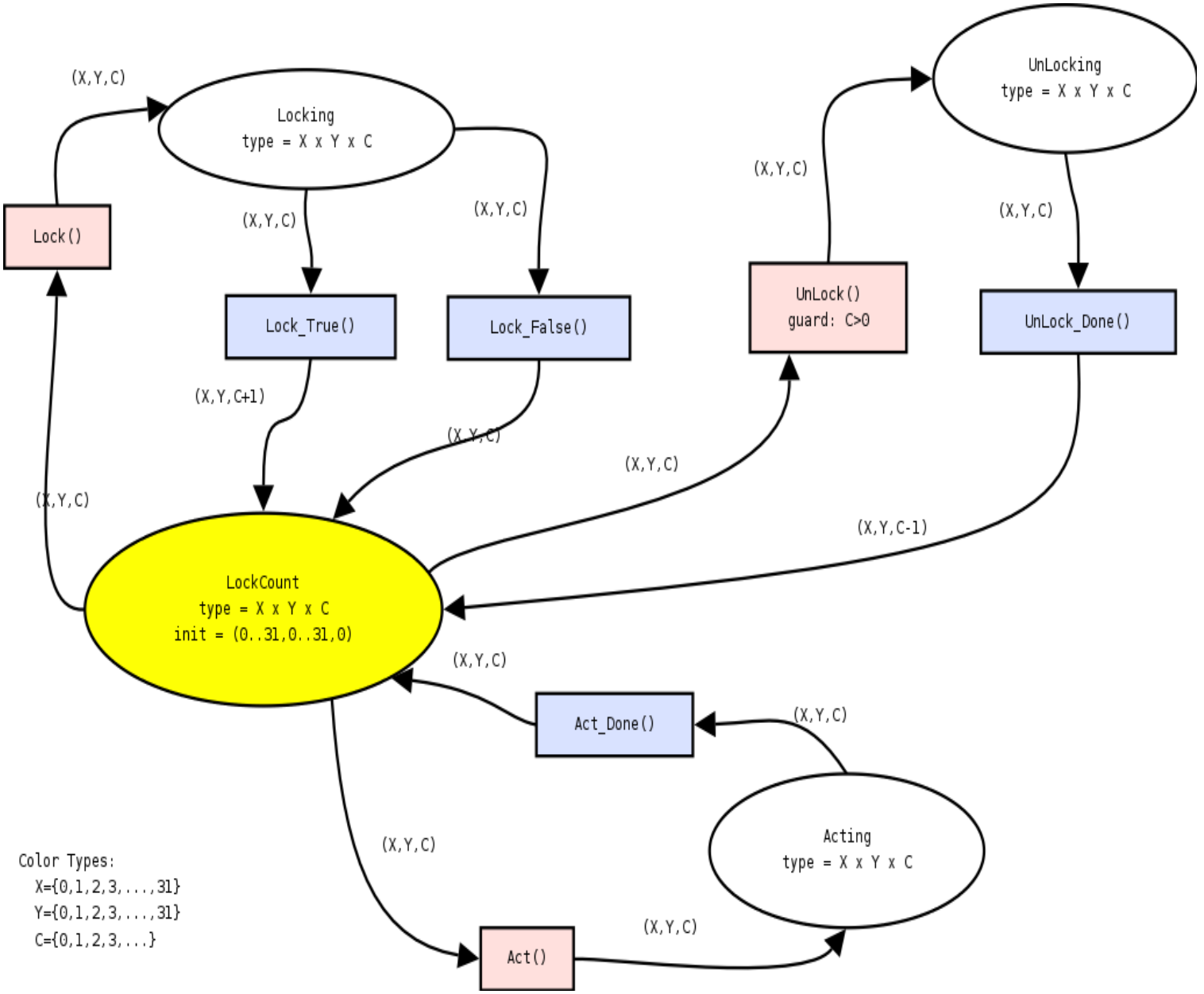
Required interface

```
out Lock(int resource)
in LockTrue
in LockFalse
out Unlock(int resource)
in UnlockDone
```

Relation between events ?

Petri-net description of API

```
out Lock()
in LockTrue()
in LockFalse()
out Unlock()
in UnlockDone()
```



Petri-Nets

$$N = (\Sigma, P, T, A, C, G, E, I)$$

- **Tokens**: a piece of data, a message

O

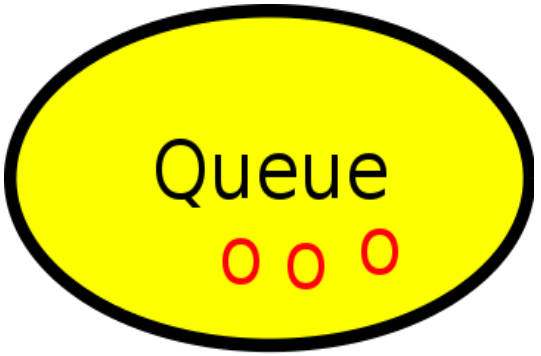
$\sum$ is a non empty set of *types*, called color sets.

$$\{XY = \{0, 1, 2, ..., 31\},$$
$$C = \{0, 1, 2, ...\},$$
$$B = \{busyL, busyU, avail\}\}$$

Petri-Nets

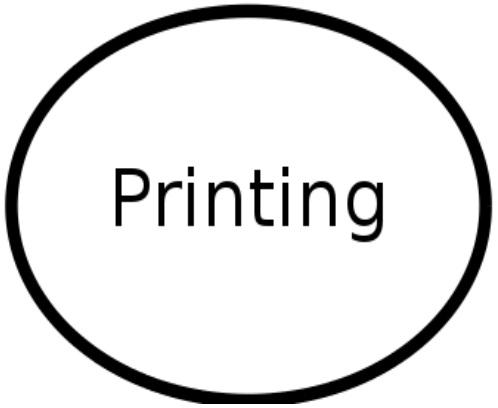
$$N = (\Sigma, P, T, A, C, G, E, I)$$

- **Places**: contain zero, one or more tokens



P are the *places*

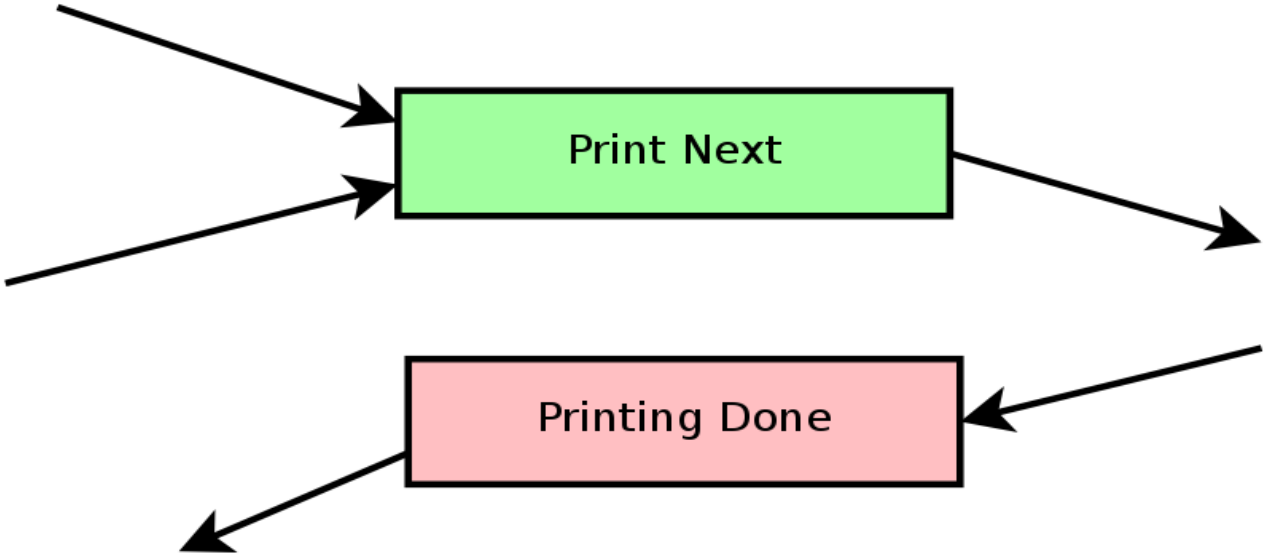
$C : P \rightarrow \Sigma$ is a color function



Petri-Nets

$$N = (\Sigma, P, T, A, C, G, E, I)$$

- **Transitions:** can move tokens between places
  - can fire when **all** inputs provide a suitable token (pre-condition / guarding)
  - when fired, removes all input tokens and creates new output tokens (post-condition)
  - fire atomically (no concurrent transitions)



Petri-Nets

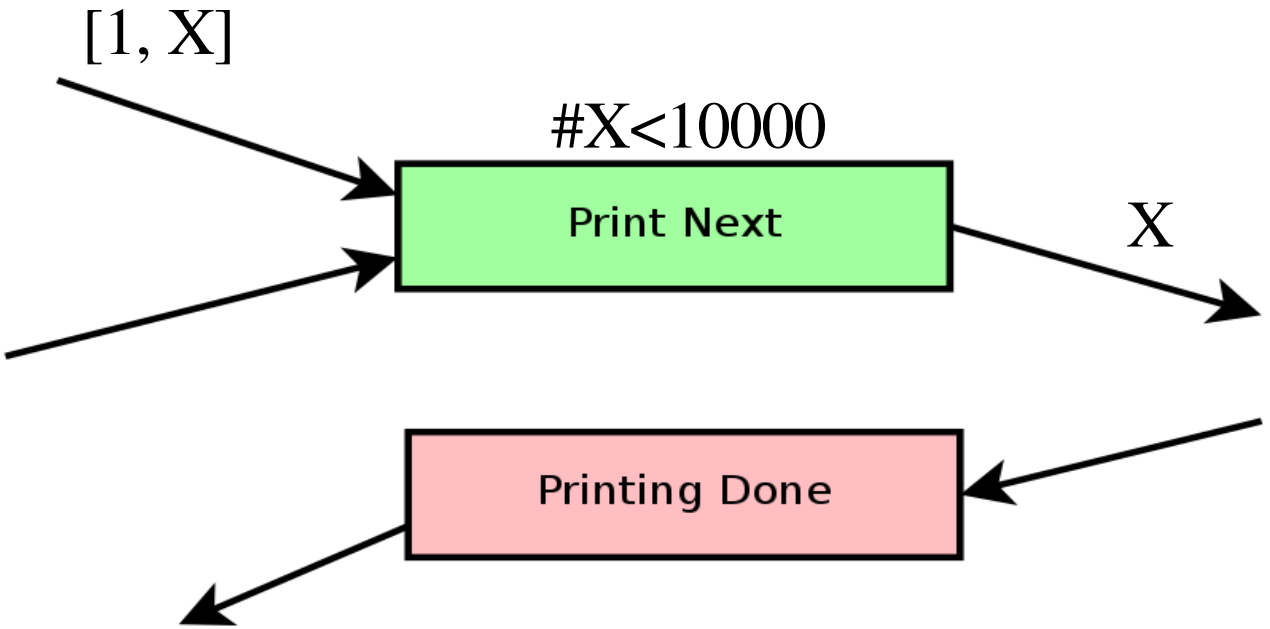
$$N = (\Sigma, P, T, A, C, G, E, I)$$

- **Transitions:** can move tokens between places

T, A are the *transitions* and the *flow* relation

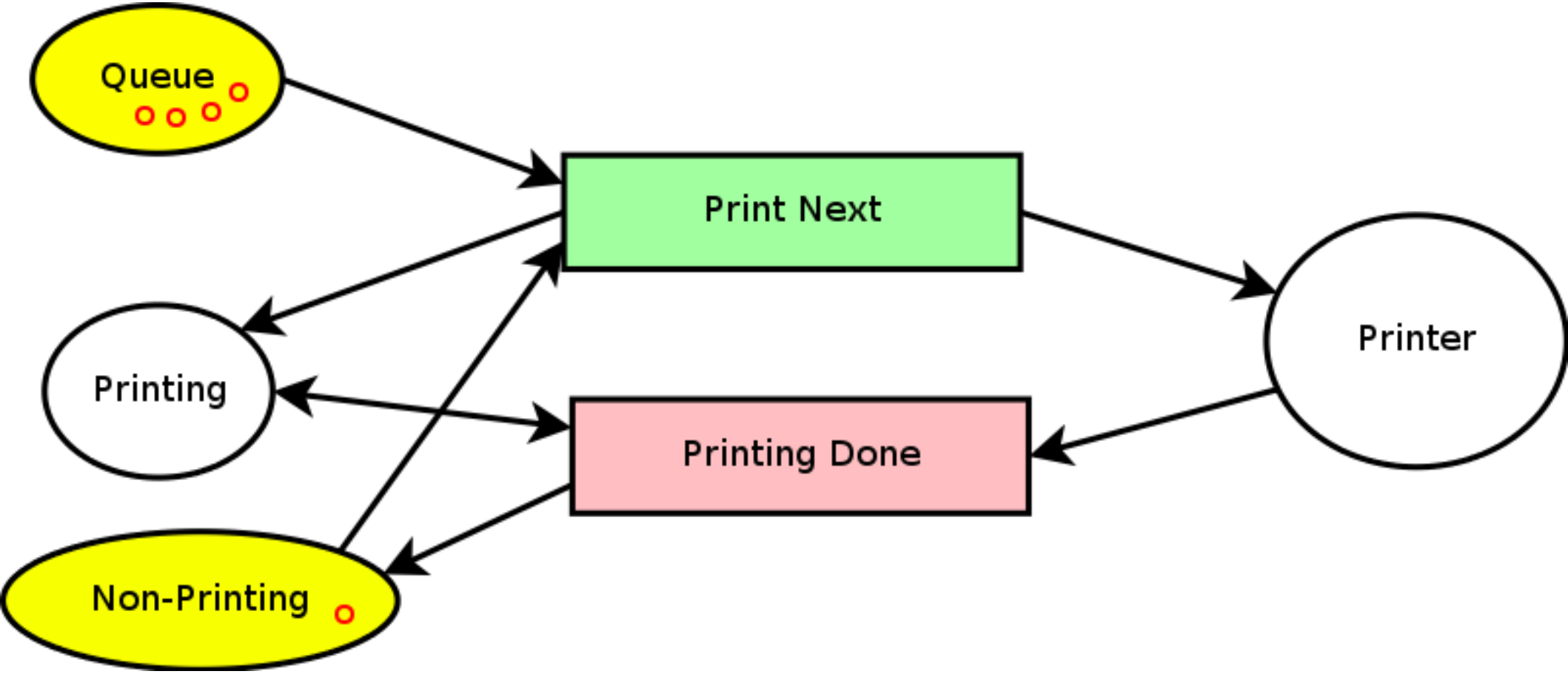
$G : T \rightarrow Expr$ is a guard function

$E : A \rightarrow Expr$ is an arc expression

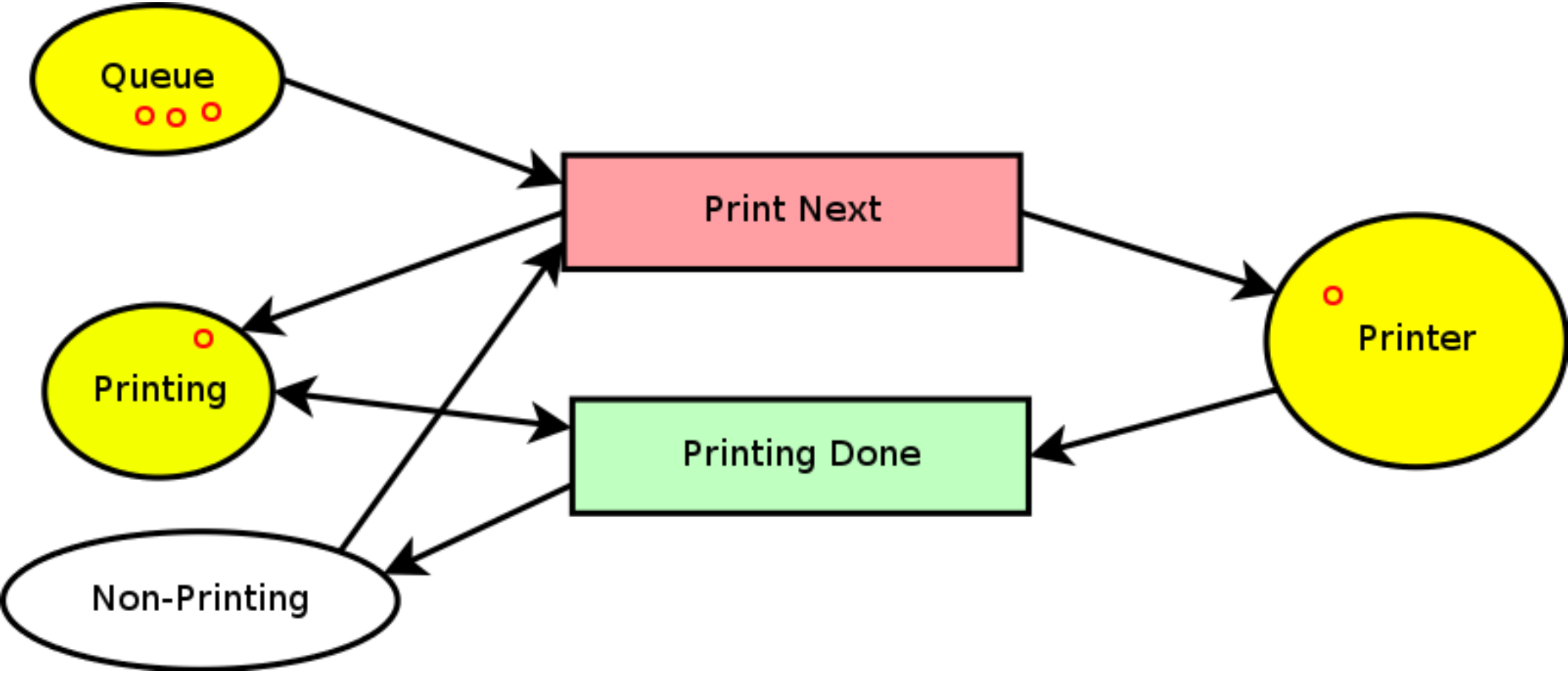


Example

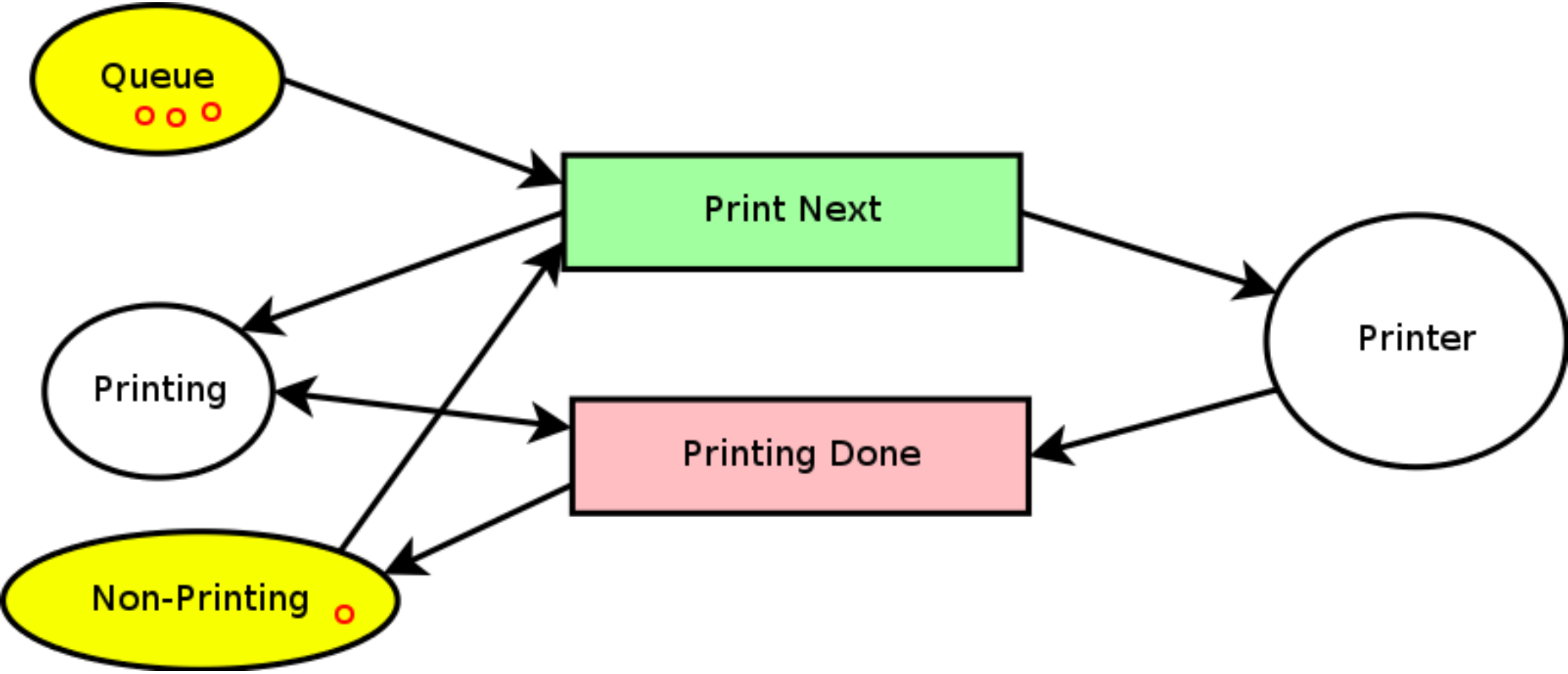
$$N = (\Sigma, P, T, A, C, G, E, I)$$



Petri-Nets



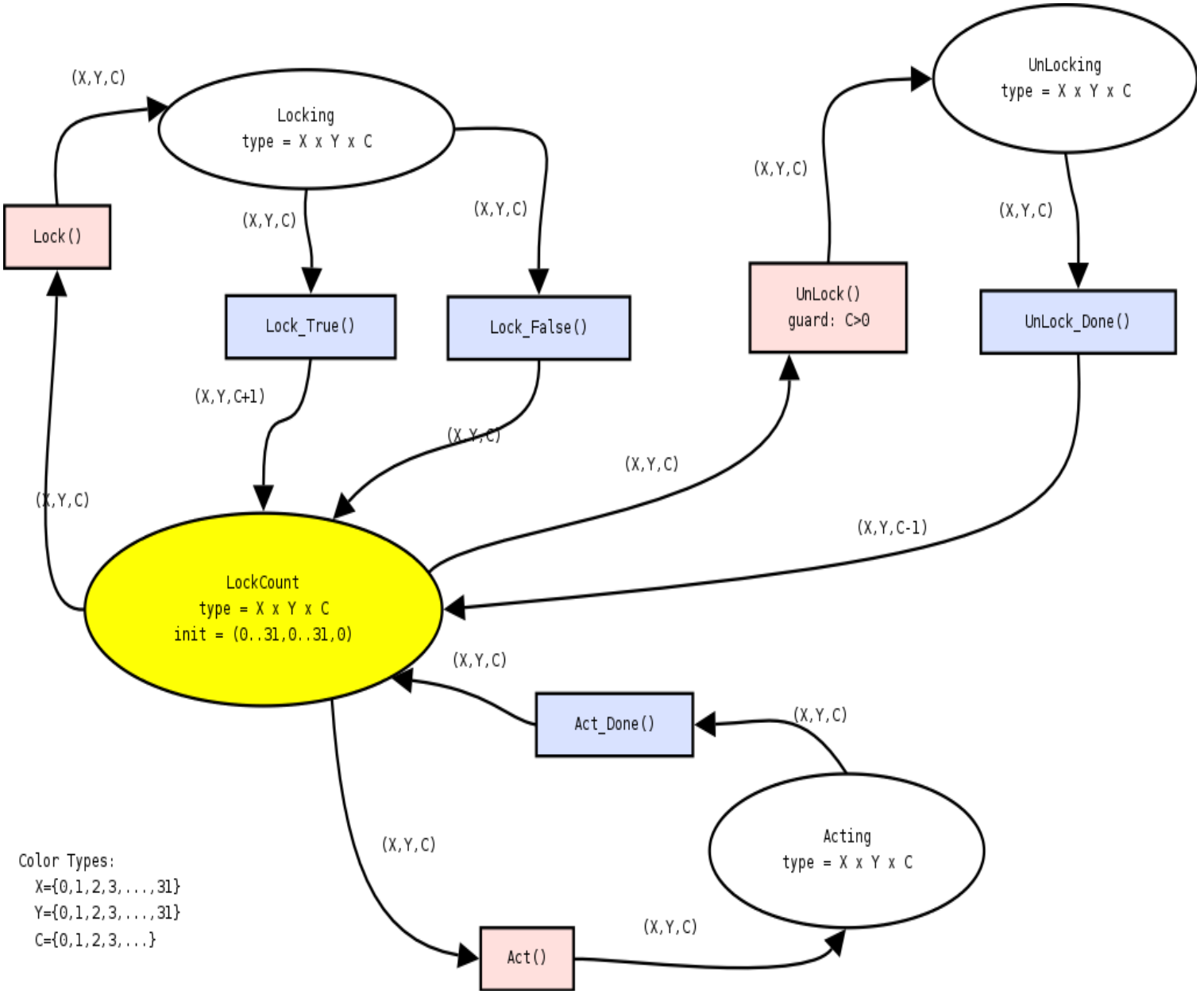
Petri-Nets



Marking

Petri-net description of API

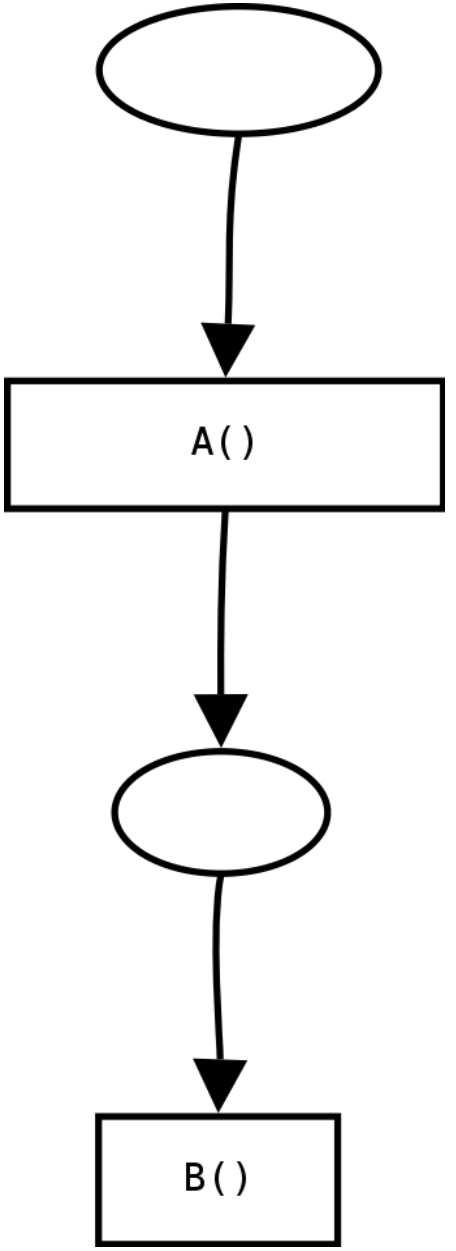
```
out Lock()
in LockTrue()
in LockFalse()
out Unlock()
in UnlockDone()
```



Script > Petri-net mapping

- Sequence

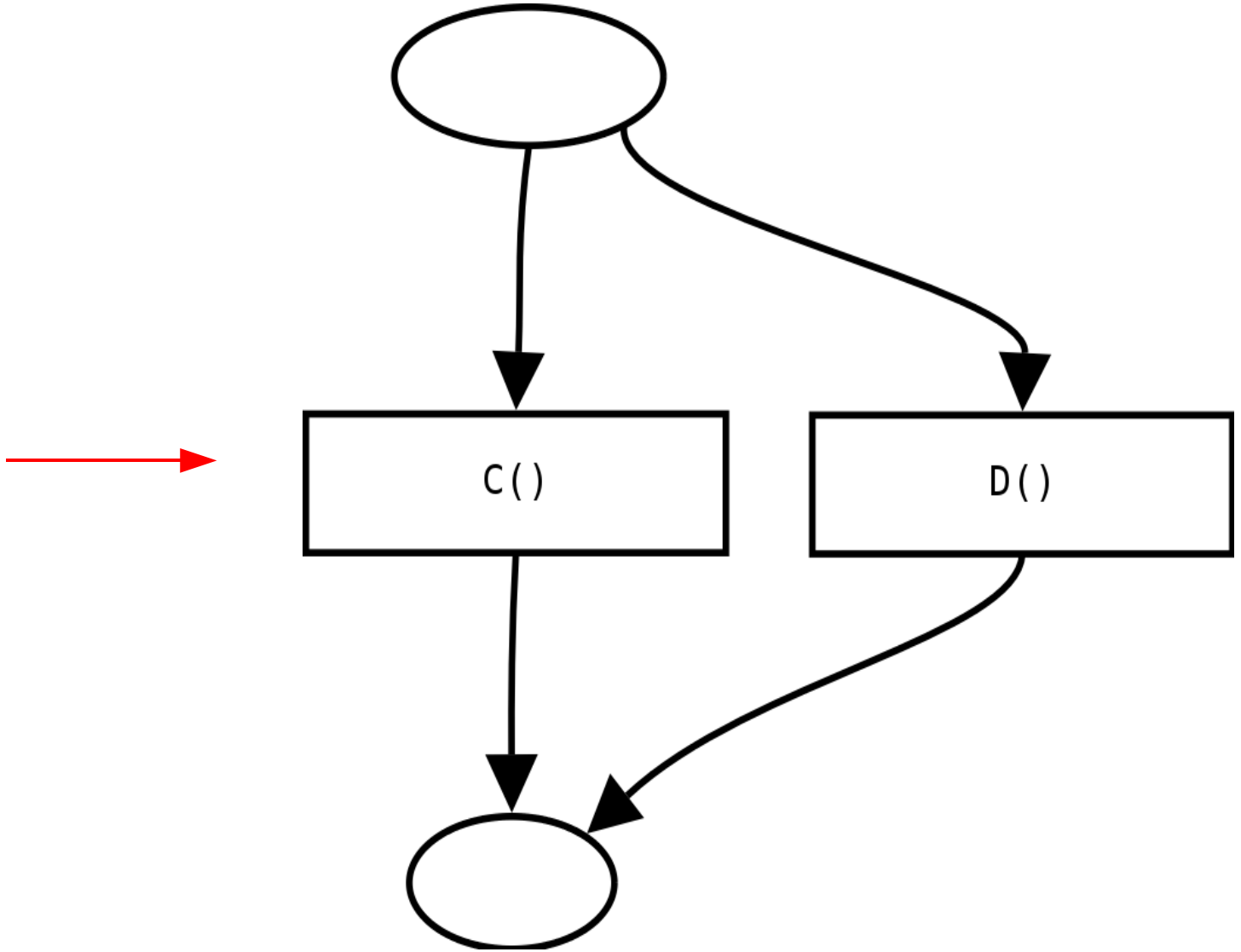
```
{
a();
b();
}
```



Script > Petri-net mapping

- Branching

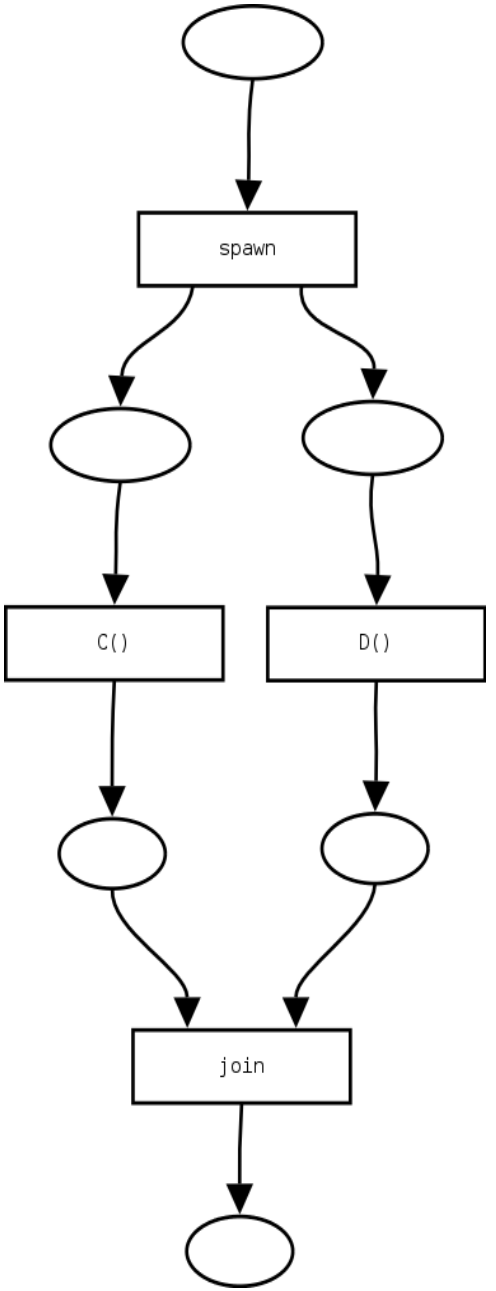
```
if .
  c();
else
  d();
```



Script > Petri-net mapping

- Parallel execution

```
par{
 c();
 d();
}
```

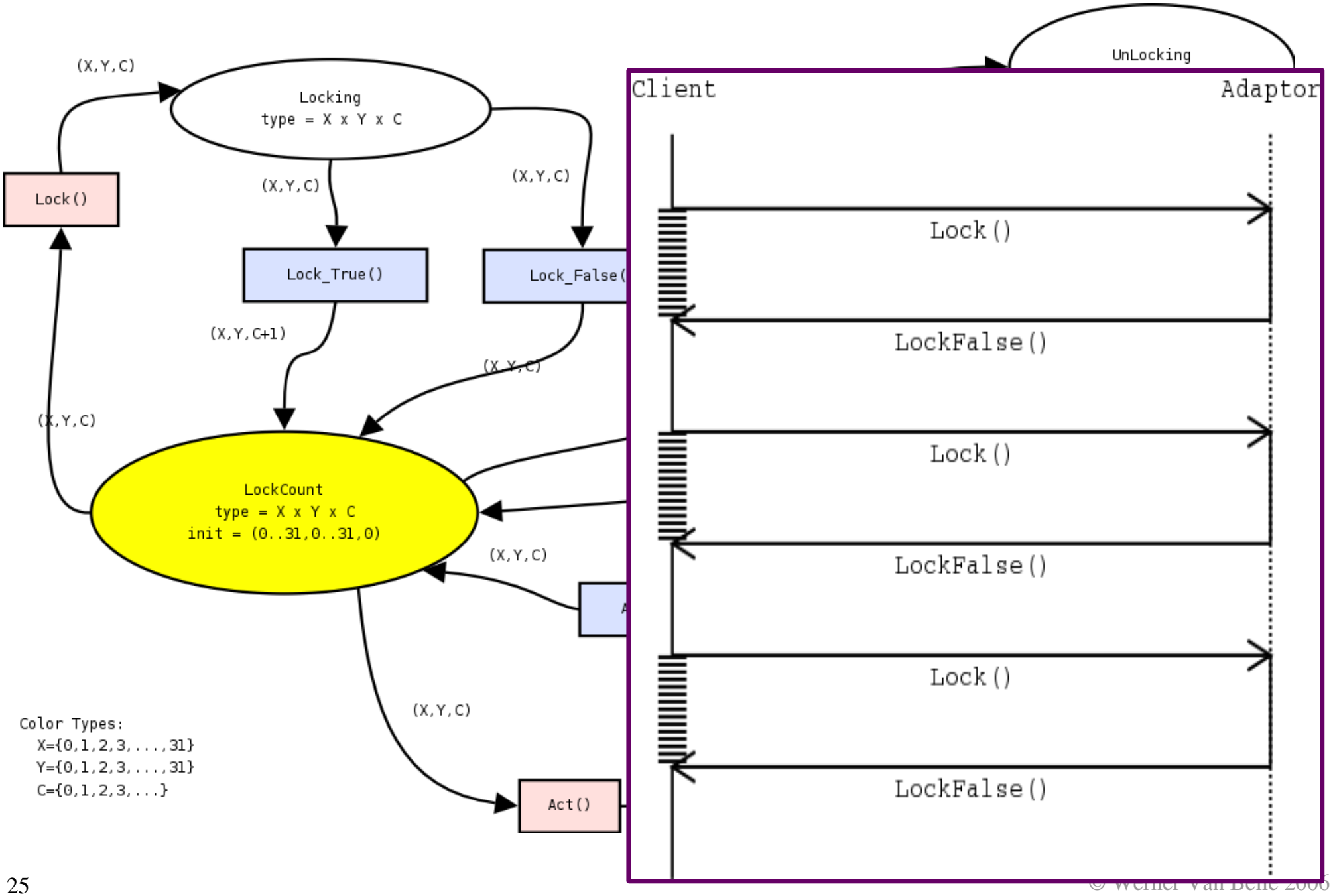


Petri-nets

- Modular & Intuitive graphical notation
- Allows for deduction of interesting properties
  - Deadlock-freedom
  - Homestate
  - Termination
  - Reachability
- Executable
  - Automatic program testing
  - Simulation
- Scheduling independence

Step 2 - **Learning Petri-Nets**

Behavior Expected by Client ?



Reinforcement Learning

- Depends on a function that associates every state and possible action in that state with a preference
  - **Policy**: which actions to take in a specific context ?
  - **Rewards**: when does the learner receive an immediate reward
  - **Value function**: which action in which state will receive the best reward in the long run ?
  - **Model of the environment**: how does one action lead to the following ?

Reinforcement Learners

- Temporal Difference learning / Q learning
  - introduce a state-value function that maps every state and choosen action to the maximum expected future reward.

$$Q(s,a) := Q(s,a) + \alpha(r + \gamma max_{a'} Q(s',a') - Q(s,a))$$

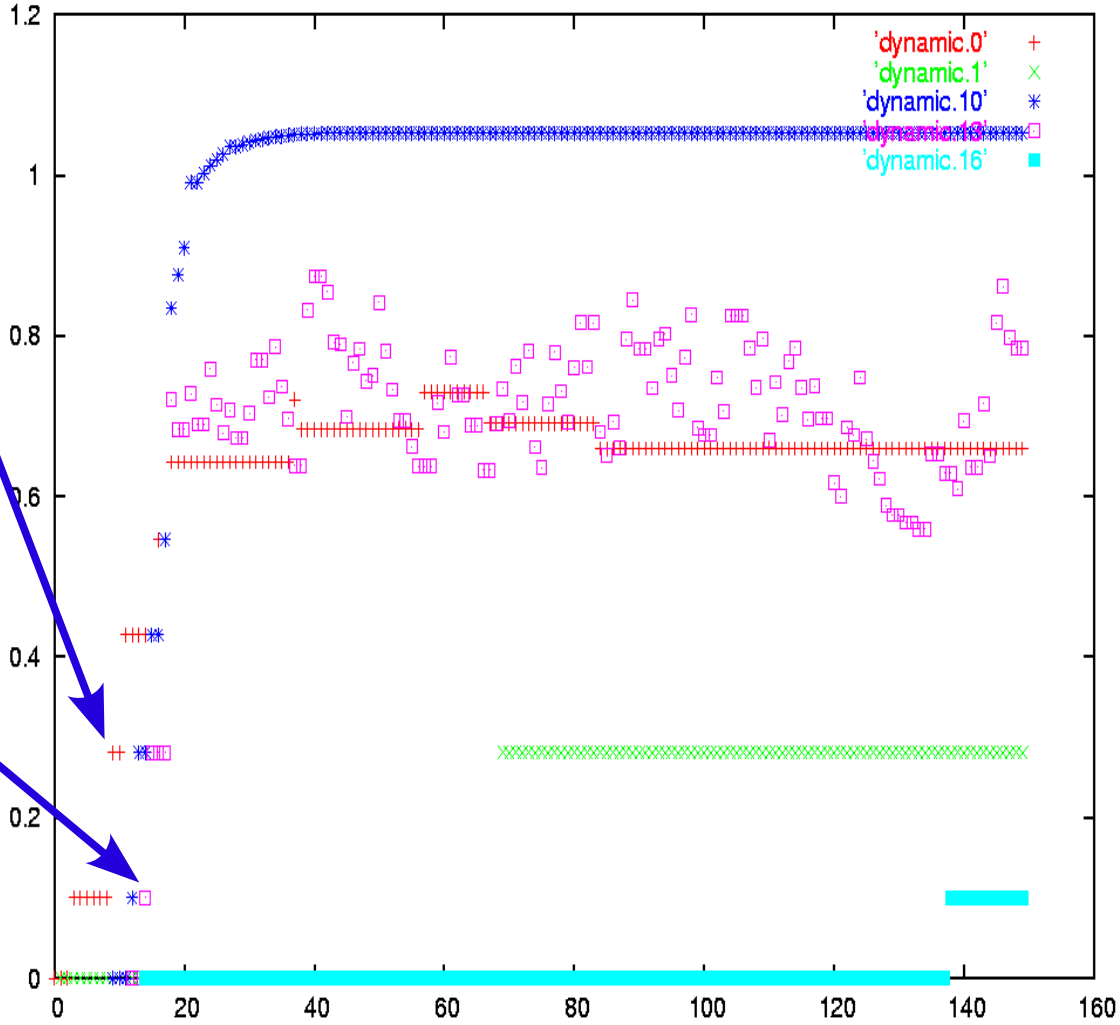
- For Petri-nets:
  - associate every possible transition with a value, update the value according to

$$Q(s,a) = Q(s,a) + \alpha(r + \gamma\, max_{a'} Q(s',a') - Q(s,a))$$

$$Q(a) = Q(a) + \alpha(r + \gamma\, max_{a' \in s'\bullet} Q(a') - Q(a))$$

Petri-net Q-learning

```
(dynamic-0 (12)
  (in lock-out?? [9 TMP1])
  (in lock_true?!)
  (out lock_true!!))
(dynamic-1 (3)
  (in unlock-out?? [TMP1 14])
  (in return_unlock_true?!)
  (out return_unlock_true!!))
(dynamic-10 (189)
  (in unlock-out?? [TMP1 TMP2])
  (in return_unlock_true?!)
  (out return_unlock_true!!))
(dynamic-13 (182)
  (in lock-out?? [TMP1 TMP2])
  (in lock_true?!)
  (out lock_true!!))
(dynamic-16 (2)
  (in unlock-out?? [TMP2 TMP1])
  (in return_unlock_true?!)
  (out return_unlock_true!!))
```



Step 3 - **Use of Constraint Logic for Action Determination**

Proper Server Locking

- Adapter will keep clients alive but still needs to realize the behavior on the server side
  - Force concurrency state on server side, 'just in time'
  - Which transitions and in which order should they be fired ?

=> Constraint Logic System using prolog

Petri-net > Prolog Rules

```
step(lock(X,Y),M,N):-
  ( \+ var(M), \+ var(N),
    del_marking(M,[ready,[]],Markinga1),
    del_marking(N,[ready,[]],Markingb1),
    del_marking(Markingb1,[locking,[X,Y]],_));
  ( \+ var(M), var(N),
    del_marking(M,[ready,[]],Markinga1),
    add_marking(Markinga1,[ready,[]],Markinga2),
    add_marking(Markinga2,[locking,[X,Y]],N));
  (  var(M), \+ var(N),
    del_marking(N,[ready,[]],Markinga1),
    del_marking(Markinga1,[locking,[X,Y]],Markinga2),
    add_marking(Markinga2,[ready,[]],M));
  ( var(M), var(N),
    empty_marking(Markinga0), empty_marking(Markingb0),
    add_marking(Markinga0,[ready,[]],M),
    add_marking(Markingb0,[ready,[]],Markingb1),
    add_marking(Markingb1,[locking,[X,Y]],N)).
```

Going forward

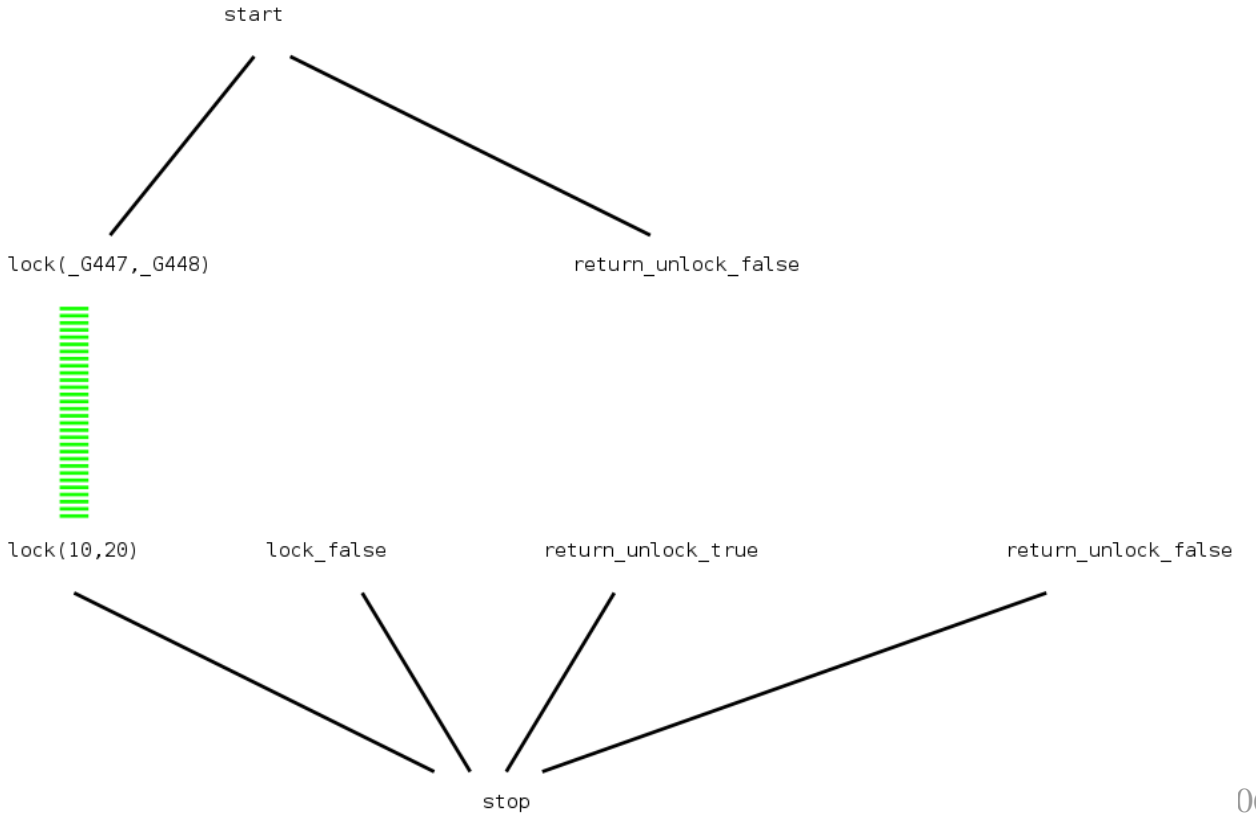
- <initial state>
  - lock(_G410,_G411) [in]
    - lock(_G594,_G595) [in]
    - lock_false [out]
    - lock_true [out]
  - unlock_false [out]
    - lock(_G707,_G708) [in]
    - unlock_false [out]

Going backward

- <final state>
  - lock(10,20) [in]
  - unlock_true [out]
  - unlock_false [out]

Constraint solving

```
solve_trace(ForwardTrace, BackwardTrace):-
    trace_matches(ForwardTrace,BackwardTrace).
solve_trace(ForwardTrace, BackwardTrace):-
    fwd_steps(ForwardTrace,NewForward),
    bwd_steps(BackwardTrace,NewBackward),
    solve_trace(NewForward,NewBackward).
```



Future

- Sensor deployment
  - Petri-nets can be used as formalism behind scenarios
  - Optimal deployment = minimalisation problem with discontinuities and time factor
  - Combination of numerical and symbolic approaches
    - Constraint Logic Based System
    - Dynamic Programming (Simplex)

Future

- Explanation based argumentation
  - based on shared 'facts'
  - when facts do not match -> explanation required
  - explanation could be the extension of the knowledge-base
  - communication should mainly focus on the inconsistencies between the different 'views' on the same information.