Verification of Distributed Systems: an Overview

Serge Haddad
LAMSADE-CNRS UMR 7024
Université Paris-Dauphine

Organisation

- Introduction
- Properties: from specification to verification
- Finite system verification
- Infinite system verification

Introduction

A very brief history

Program verification

- The formalism is the *programming language* or some abstraction.
- The properties to be checked are the *partial correction* and the *termination*.
- The properties are expressed via a first-order logic.

Reactive system verification

- New features: concurrency, non determinism, time, randomness, etc.
- New properties: safety properties, liveness properties, etc.
- Numerous formalisms and languages

Formal models for verification

Expressiveness versus decidability and complexity

- Realistic systems require highly expressive models.
- However, this quickly leads to undecidablity or untractability.
- Thus, useful formal models requires abstraction performed by the modeller *(interesting also from a modelling point of view).*

Desirable features of a model

- Modularity and hierarchy
- Parametrization
- Refinements
- Time ...

From conception to verification: formal semantics

- Ensures that the model transformation is valid
- Often obtained via operational rules related to constructors

Property Specification

Generic properties *(never exhaustive)*

- Interpretation valid whatever the model *(e.g. deadlocks)*
- Specialized efficient algorithms

Specific properties *(possible untractability of verification)*

- Express properties relative to the model *(e.g. a request will always be followed by an answer in at most 3 s.)*
- Supported by logics-based languages

Model equivalence *(importance of the modelling step)*

- Follows the modelling process.
- For instance, equivalence between the (abstract) model of the service and the (concrete) model of the protocol.

Test of models *(partial guaranty)*

- Does not require any *a priori* model.
- Most efficient algorithms.

Verification Methods

Automatic or semi-automatic methods?

- Automatic methods decrease the cost of verification but their application field is more limited.
- Semi-automatic methods has a wider application field but require experts and can be time consuming.

Structural methods

- Results give more information.
- Efficient algorithms handling parametrization.
- Either semi-decision algorithms or algorithms for subclasses of the considered formalisms.

Behavioural methods

- Easy to understand and applicable to all finite models and some infinite ones.
- Must include elaborate algorithms in order to cope with the complexity.

Properties:
from specification to verification

Transition systems

Low-level formalism which generally is the semantics of some *high-level* formalism $\Rightarrow$ *implicit description*

A (possibly infinite) graph whose vertices are *states* and whose edges are *transitions, when finite much bigger than the high-level specification*

In a state, some *atomic properties* hold and some others do not.

A transition is (possibly) labelled by the *event* which triggers the transition.

A transition system has a set of initial states.

Interpretation of transition systems

Examples of atomic properties

- The value of variable `x` is `9`. The value of variable `x` is between `13` and `27`.
- The ordinal counter points to the instruction `Ins`.
- Process `p` waits for resource `r`. Process `p` has locked resource `r`.
- Channel `c` is empty. There are two occurrences of messages `m` in channel `c`.

Examples of events

- Process `p` resets variable `x`.
- Process `p` releases resource `r`.
- Process `p` sends (receives) message `m` in (from) channel `c`.

Generic properties of transition systems

The reachability problem: Is some state `s'` reachable from some (initial) one `s`?

The deadlock problem: Is some state without successor reachable from some (initial) one `s`?

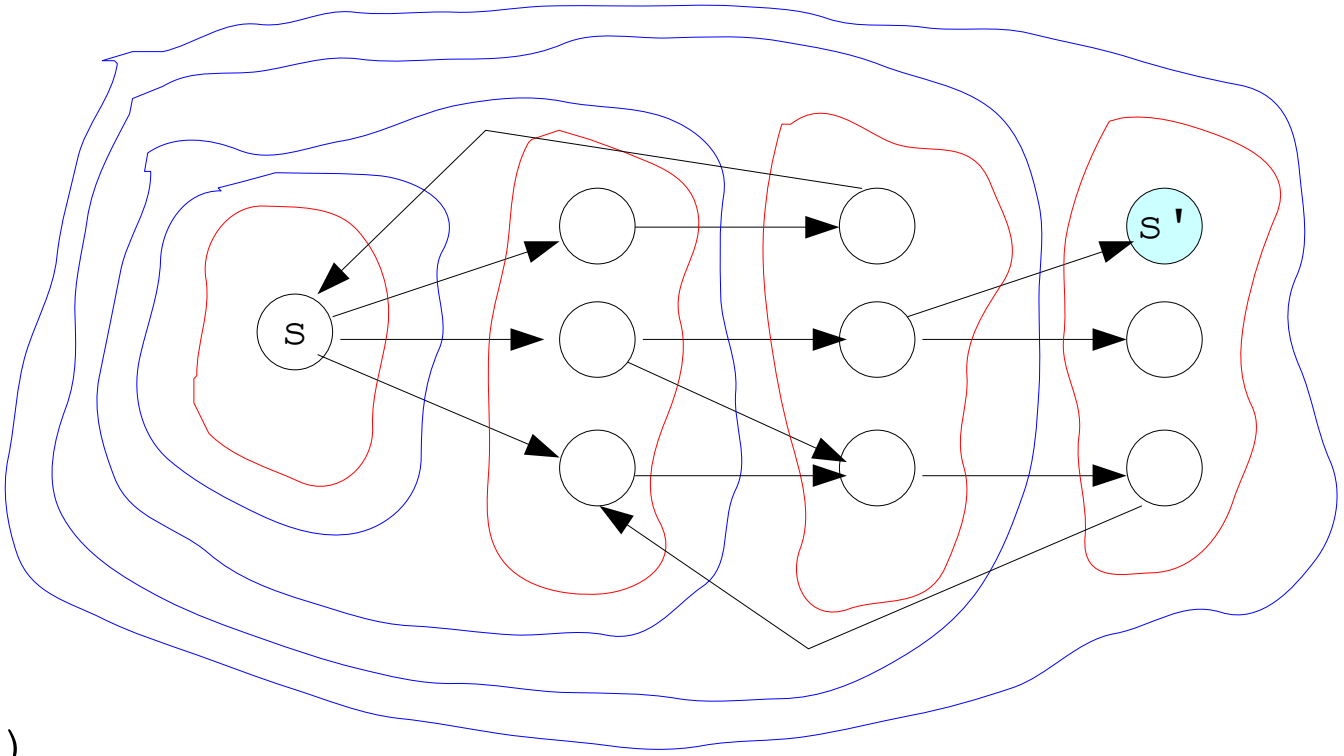
The quasi-liveness problem: Will some event `e` occur from some (initial) state `s`?

The liveness problem: Will some event `e` occur from every reachable state?

The fairness problem (or its negation): Is some subset of events `E` infinitely avoidable from some reachable state?

How to check reachability? (1)

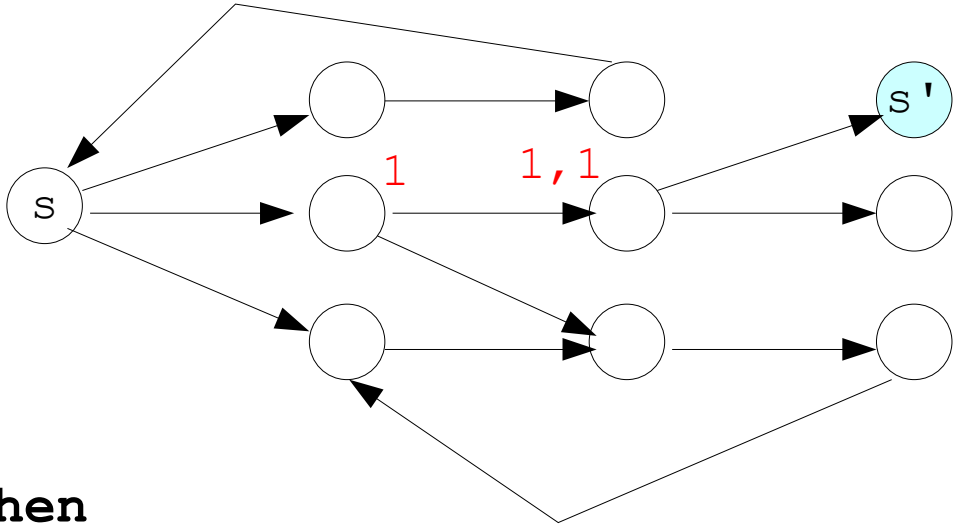
By saturation




```
Search()
   Front = Visited = {s};
   while s' ∉ Visited and Front≠∅ do
      Front = Successor(Front) \ Visited;
      Visited = Visited ∪ Front;
   return(s' ∈ Visited);
```

How to check reachability? (2)

By "limited" memoryless exploration




```
Search()
   return(search(s,s',|S|-1);

search(scur,snext,l)
   if scur=snext or
      snext ∈ Successor(scur) then
      return(true);
   else if l≤1 then
      return(false));
   else
      for sint ∈ S\{scur,snext} do
         if search(scur,sint,⌊l/2⌋) and
            search(sint,snext,⌈l/2⌉) then
            return(true);
      return(false);
```

Compare the algorithms

In summary

Generic properties verification

Checking generic properties can be efficiently performed...
(generally in linear time w.r.t. the size of the transition system)

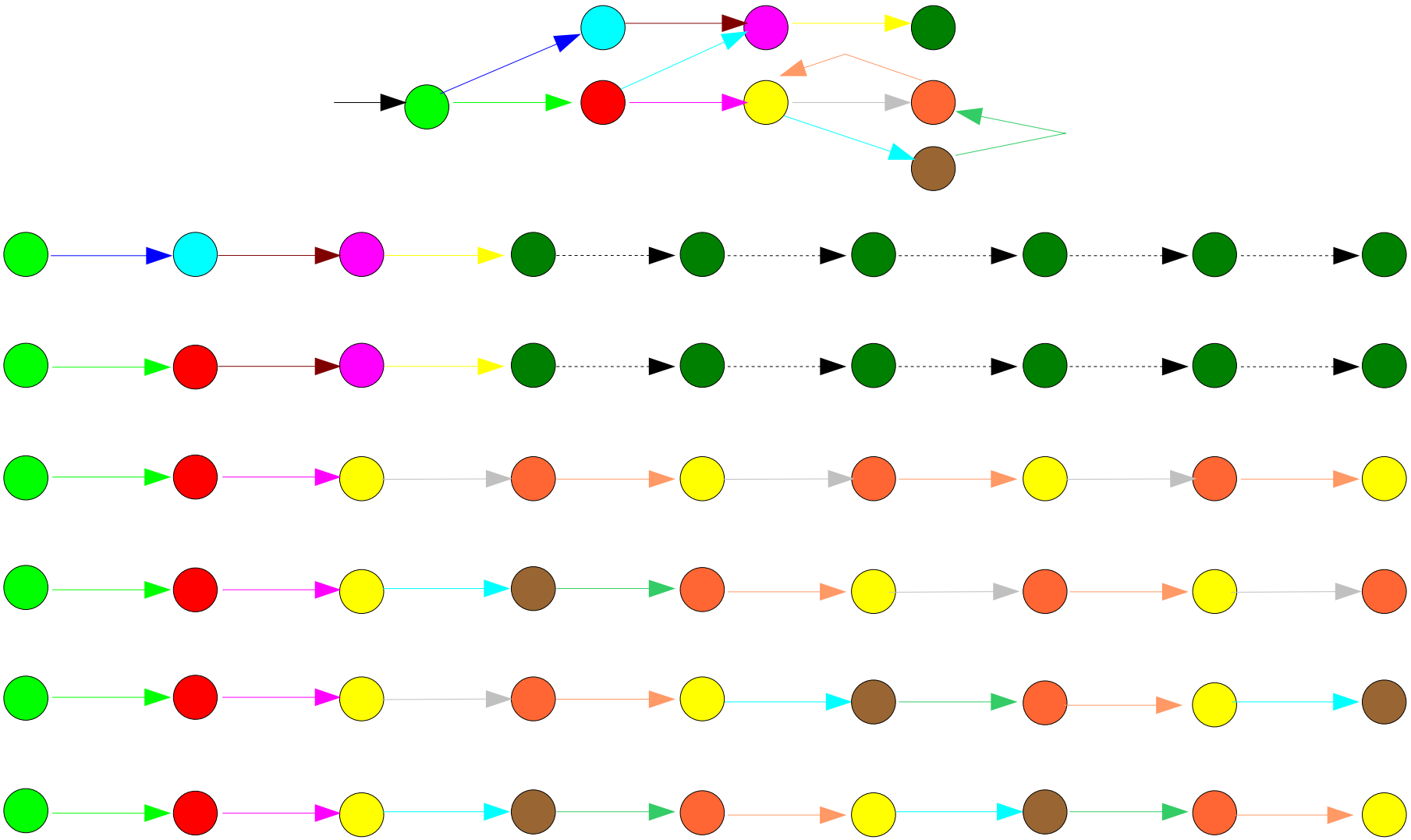
but algorithms must be tailored for each property.

Generic properties expressiveness

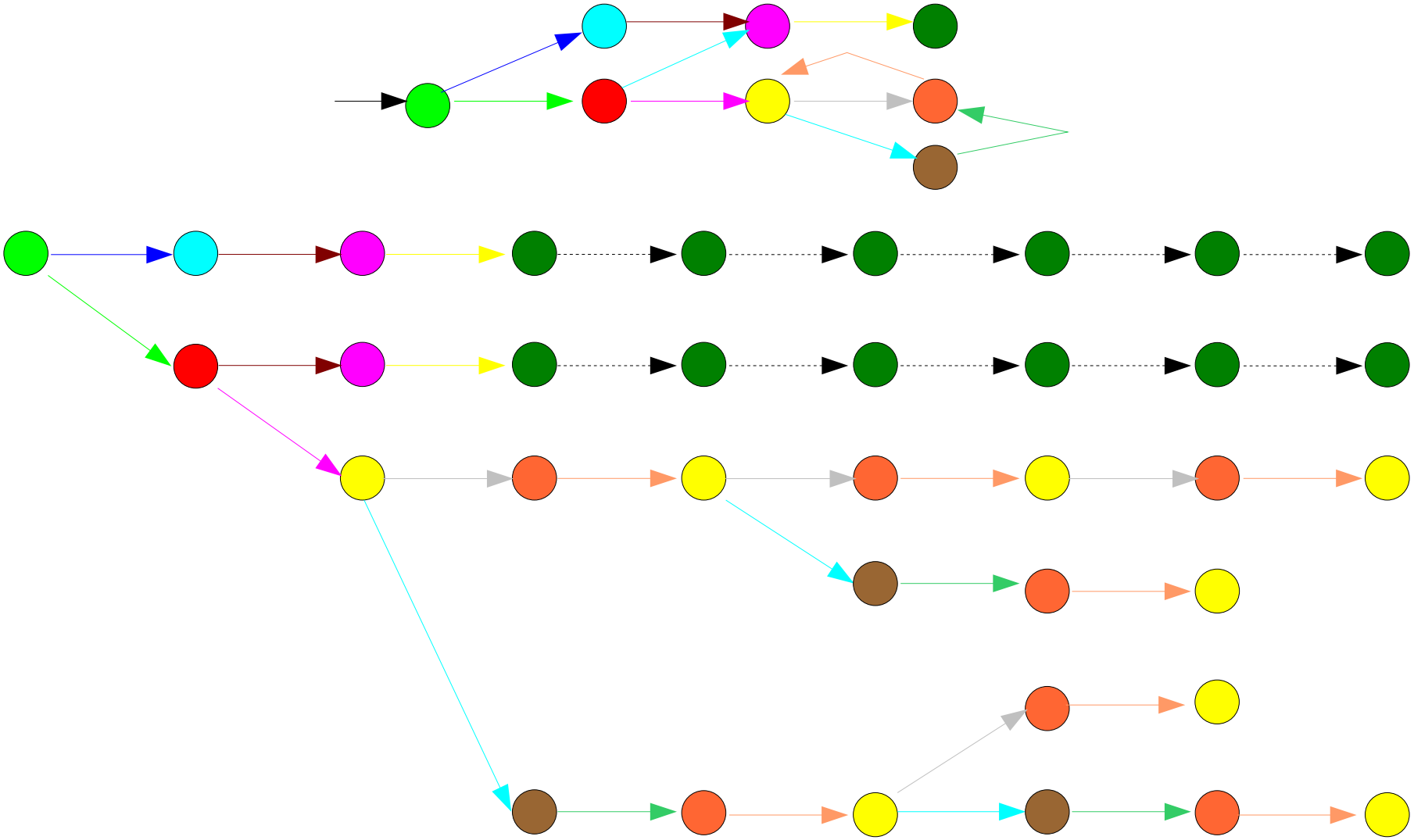
Generic properties satisfaction outlines well-designed systems...

but it is irrelevant for specific properties of a system.
(e.g. every request received by a server will be treated or rejected with an advertisement message)

Linear Temporal Logics
reason about infinite sequences

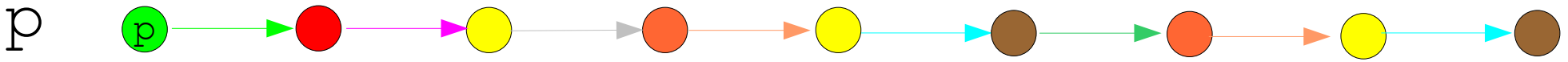


Branching Temporal Logics
reason about the infinite computation tree

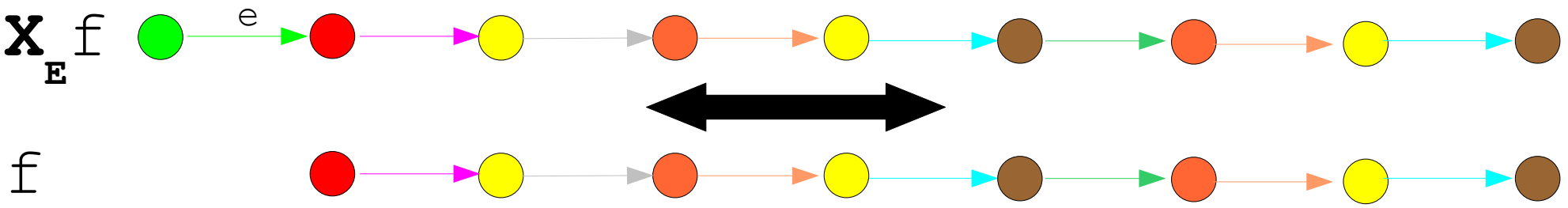


Temporal operators of LTL ($\mathbf{X}_E$)

An atomic property p is true iff it is true in the first state of the sequence

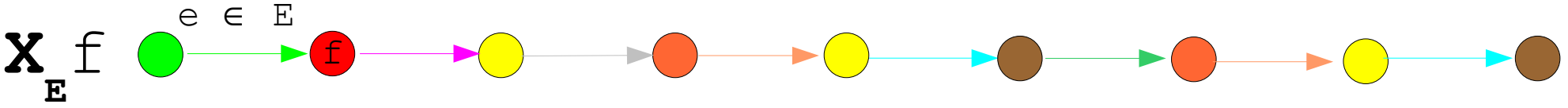


A property f will be true in the next subsequence ($\mathbf{X}_E$)



and e ∈ E

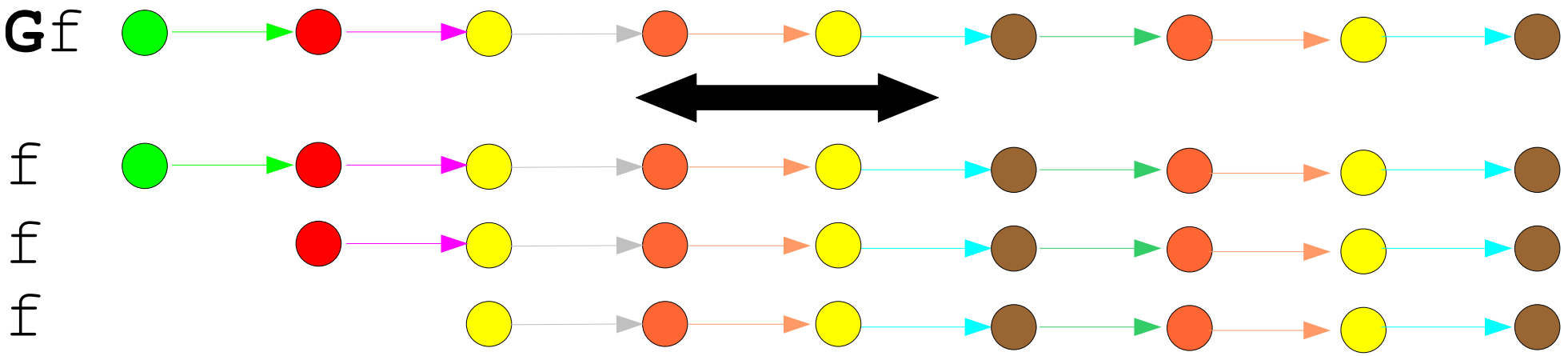
When f is an atomic property



X is an abbreviation for $\mathbf{X}_E$ when **E** is the set of all events

Temporal operators of LTL (**G**)

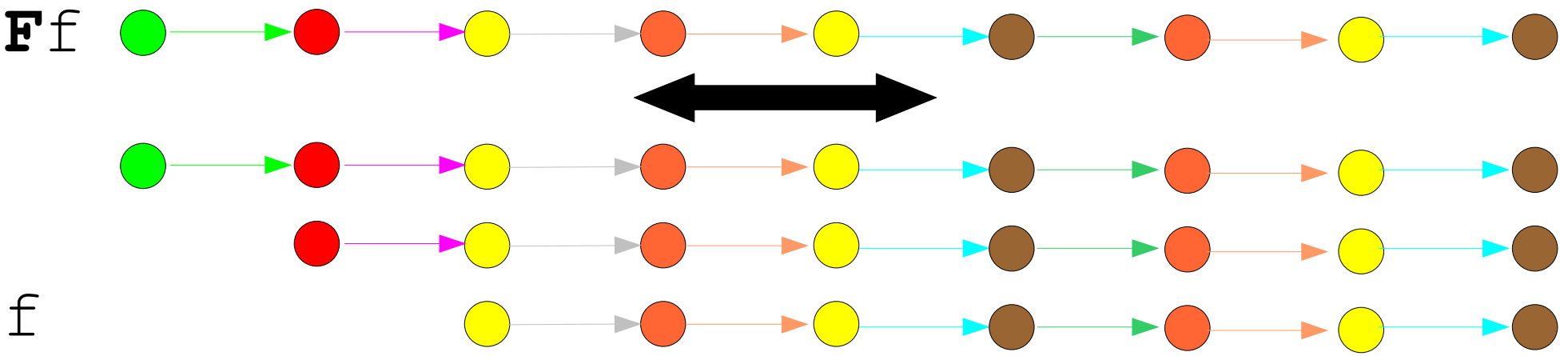
A property `f` is always (**G**) true



When `f` is an atomic property

Temporal operators of LTL (**F**)

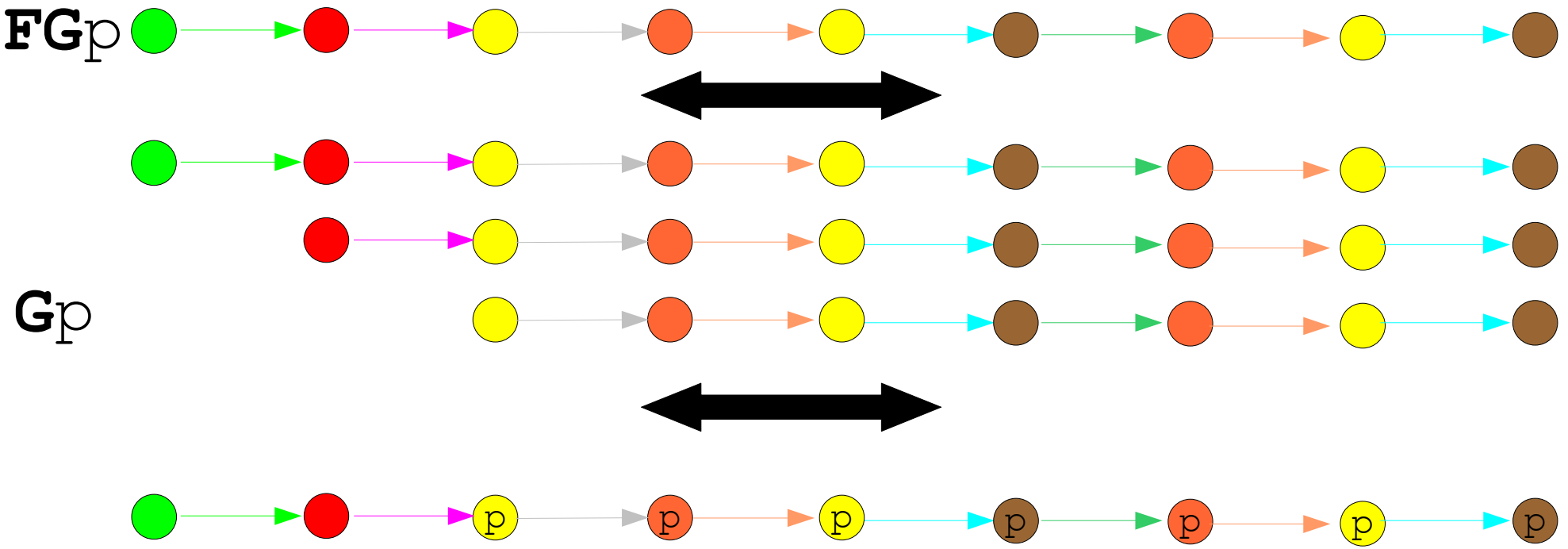
A property `f` will be eventually (**F**) true



When `f` is an atomic property

Combining temporal operators (1)

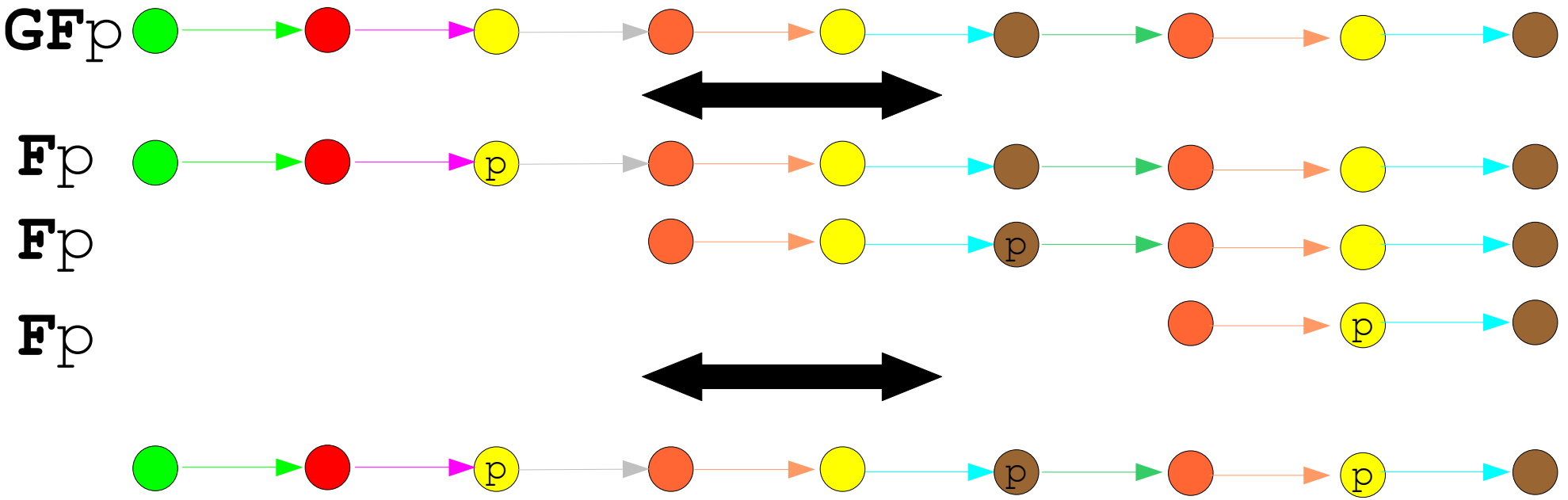
What does **FG**p mean?



FGp means: "at some instant p will hold forever"

Combining temporal operators (2)

What does **GF**p mean?

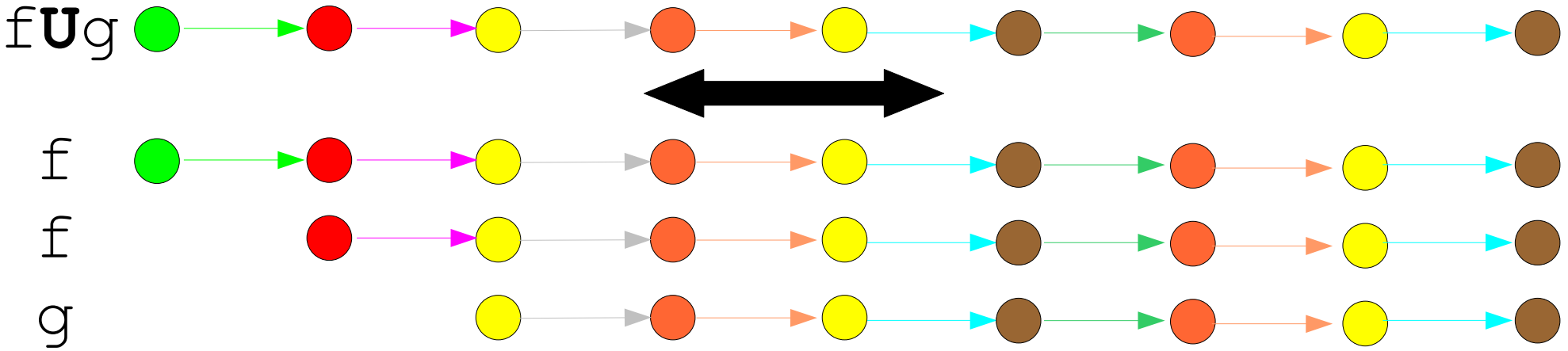


GFp means: "at any instant p will be eventually true" or equivalently "p will be true infinitely often".

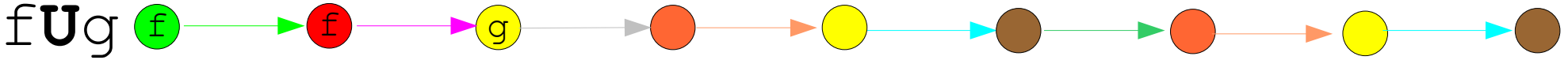
Note that **FG**p⇒**GF**p but the converse is not true.

Temporal operators of LTL (**U**)

A property `f` will hold until a property `g` will be eventually (**U**) true



When `f` and `g` are atomic properties



CTL*

CTL* consists in formula about sequences and states.

A formula relative to a sequence is:

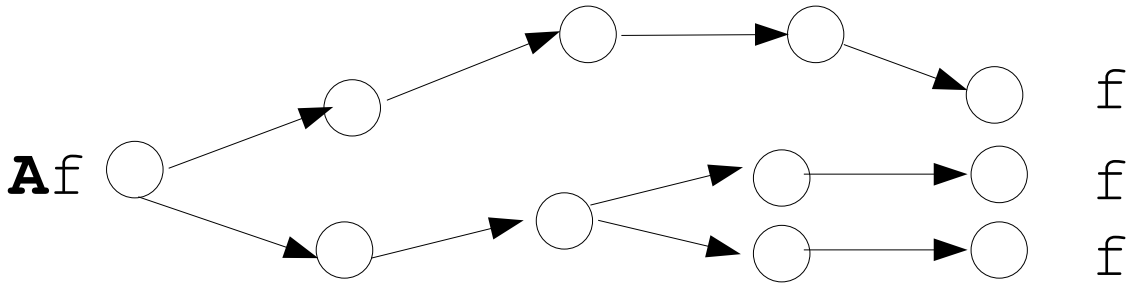
- A formula about states *(and is evaluated on the first state of the sequence as for LTL)*
- $\neg f, f \vee g, f \wedge g, \mathbf{X}_{\mathbf{E}} f, \mathbf{F} f, \mathbf{G} f, f \mathbf{U} g,$ where f and g are formulas about sequences

A formula relative to a state is:

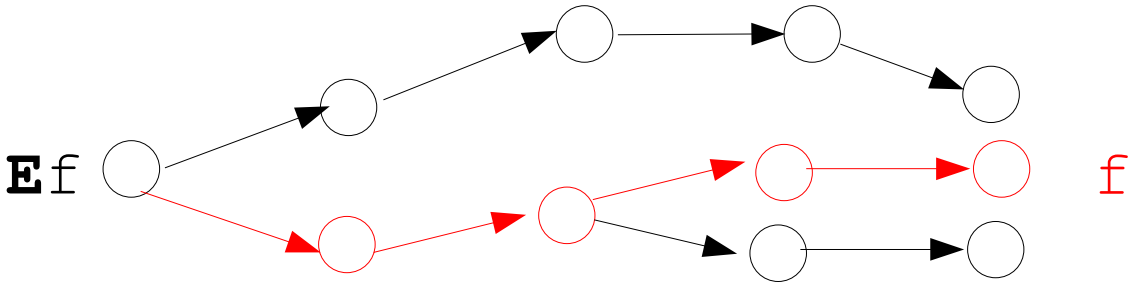
- An atomic property
- $\mathbf{A} f, \mathbf{E} f$ where f and g are formulas about sequences

Branching operators of CTL*

Af means that f is true on every sequence starting from the state.



Ef means that f is true on at least one sequence starting from the state.



CTL

CTL is a restriction of CTL* which limits the way one combines the temporal and the branching operators.

It can be expressed only by formula relative to states:

- An atomic property
- $\neg f, f \vee g, f \wedge g$
- $\mathbf{AX}_{\mathbf{E}} f, \mathbf{AF} f, \mathbf{AG} f, \mathbf{A} f \mathbf{U} g$
- $\mathbf{EX}_{\mathbf{E}} f, \mathbf{EF} f, \mathbf{EG} f, \mathbf{E} f \mathbf{U} g$
- where f and g are formulas

LTL
E does not occur
and **A** occurs only
at the root of the syntactic tree

CTL*

CTL
in the syntactic tree branching operators
and temporal operators occur in edges
$\{\mathbf{E}, \mathbf{A}\}$---$\{\mathbf{X}_{\mathbf{E}}, \mathbf{F}, \mathbf{G}, \mathbf{U}, \mathbf{W}\}$

Generic properties revisited

The reachability problem: **EF**p expressible in CTL and LTL (*via* the negation **AG**$\neg p$)

The deadlock problem: $\mathbf{EFEX}_{\mathbf{stop}}\,\mathtt{true}$ expressible in CTL and LTL (*via* the negation $\mathbf{AG}\neg\mathbf{X}_{\mathbf{stop}}\,\mathtt{true}$)

The quasi-liveness problem: $\mathbf{EFEX}_{\mathbf{e}}\,\mathtt{true}$ expressible in CTL and LTL (*via* the negation $\mathbf{AG}\neg\mathbf{X}_{\mathbf{e}}\,\mathtt{true}$)

The liveness problem: $\mathbf{AGEFEX}_{\mathbf{e}}\,\mathtt{true}$ expressible in CTL but not in LTL (*try*)

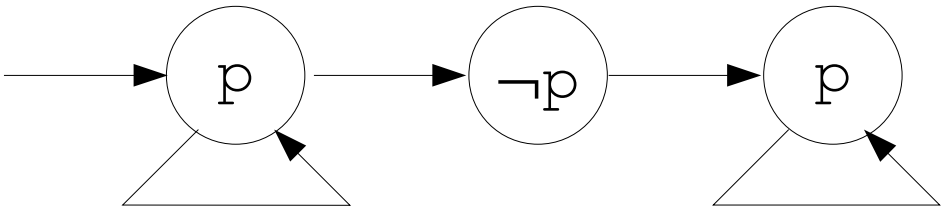
The fairness problem: $\mathbf{AGAFX}_{\mathbf{E}}\,\mathtt{true}$ expressible in CTL and $\mathbf{AGFX}_{\mathbf{E}}\,\mathtt{true}$ in LTL

CTL and LTL are incomparable

The liveness problem is not expressible in LTL. Intuitively it requires to leave the sequence.

Some fairness properties are not expressible in CTL. Intuitively they require to nest temporal operators.

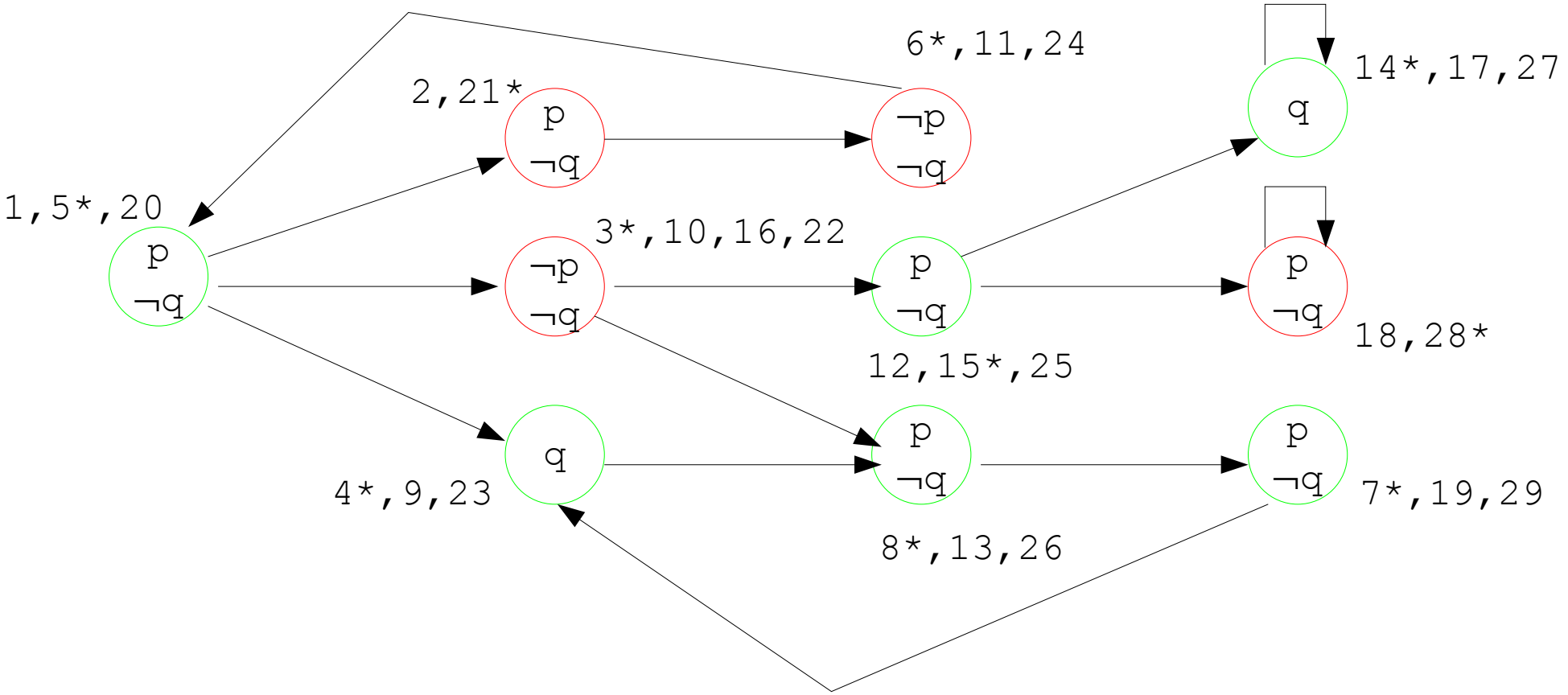
Example: **AFG**p (different from **AFAG**p)



Evaluate the two formulas on this model

Model-checking CTL formulas: **EU**

Model-checking **E**p**U**q formulas is performed by backward exploration *(in linear time w.r.t. the size of the graph)*



How many times is a node visited?

Model-checking LTL formulas

Observation

- A formula of LTL "accepts" (infinite) sequences
- A finite automaton "accepts" (finite or infinite) words
- Let a subset of propositions be a letter, then there is a close correspondence between the two notions of "acceptance"

Idea

- Transform a formula of LTL into a finite automaton which accepts the same set of sequences (i.e. infinite words)

Problem

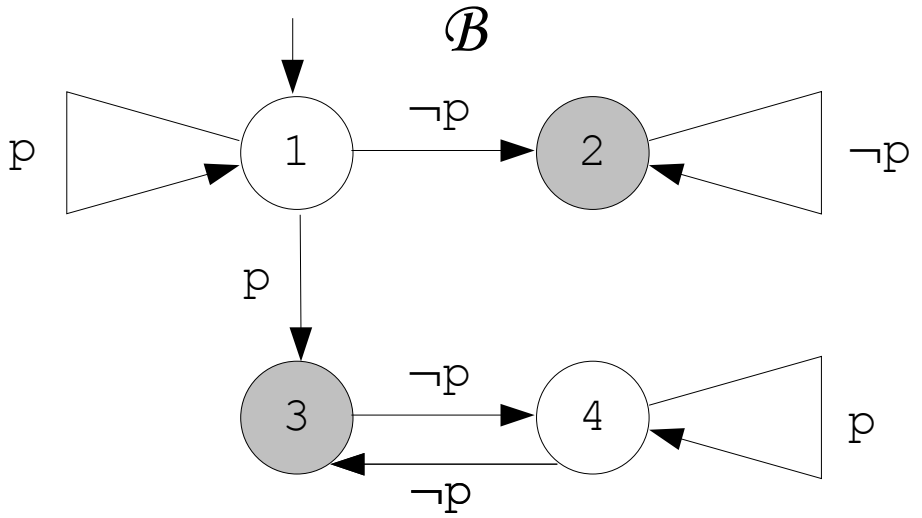
- Which acceptance condition for the automaton?

Büchi automata

Büchi automata are the "simplest" automata which accept infinite words. A Büchi automaton has a set of final states F and a sequence is accepted if there is a corresponding path which meets infinitely often F.

$pppp\neg p\neg p\neg p(\neg p)^{\omega}$ is accepted by this automaton

$ppp\neg ppp\neg pp\ldots$ is rejected by this automaton



Give a description of the sequences accepted by this automaton

Principle of LTL model-checking

Construction

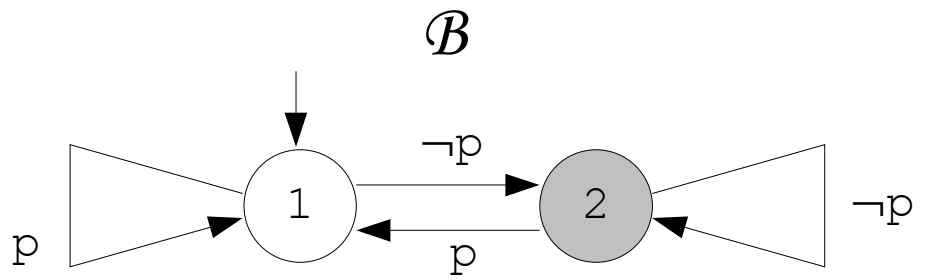
- Given φ a formula, build a Büchi automaton $\mathcal{B}$ for $\neg\varphi$
- Build the synchronized product of $\mathcal{B}$ and model $\mathcal{M}$.
- This synchronized product $\mathcal{B}\otimes\mathcal{M}$ is a Büchi automaton which accepts the infinite sequences of the model satisfying $\neg\varphi$.

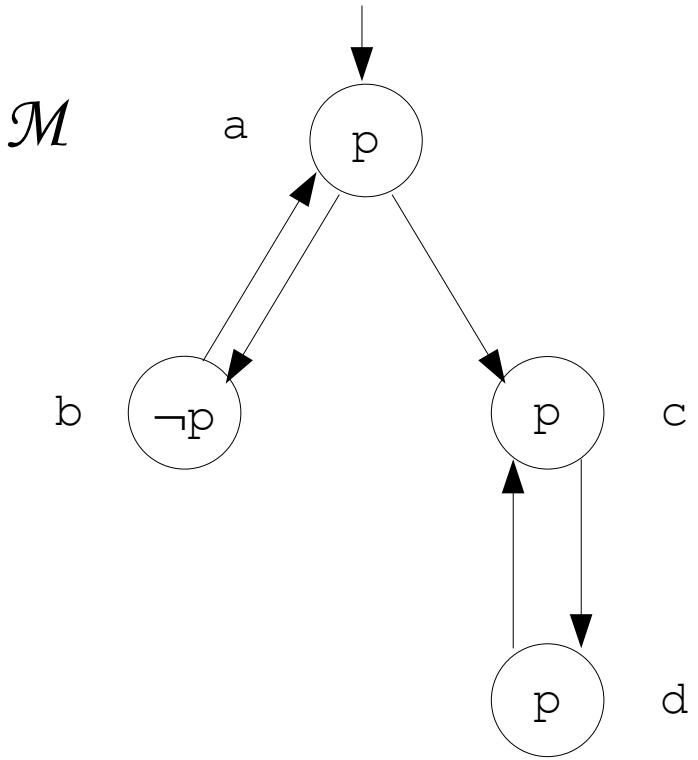
Verification

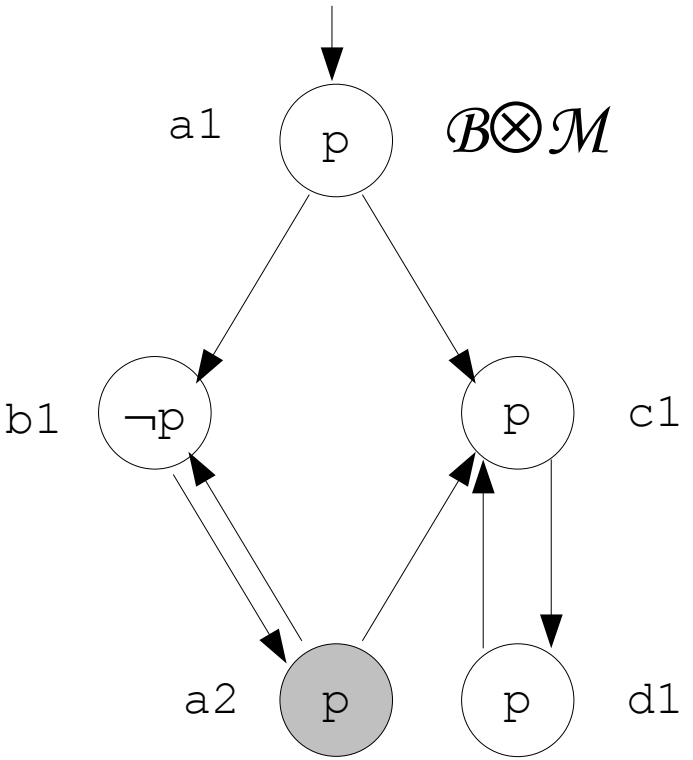
- Check whether $\mathcal{B}\otimes\mathcal{M}$ accepts at least one sequence.

The synchronized product

$\varphi = \mathbf{FG}p$, $\neg\varphi = \mathbf{GF}\neg p$

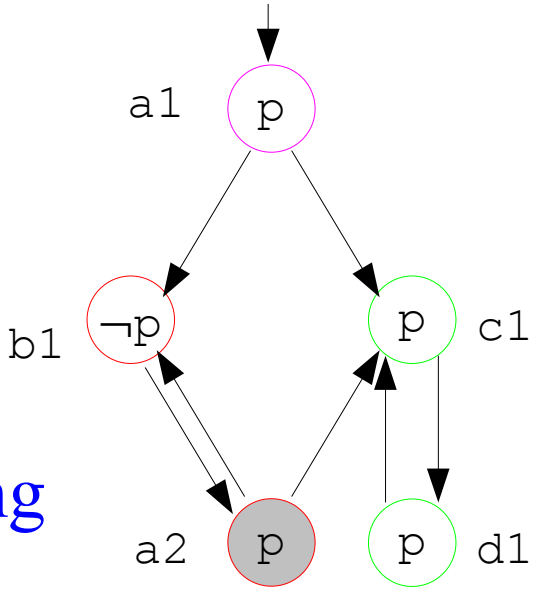






Checking emptyness of the synchronized product

There is no sequence accepted by $\mathcal{B} \otimes \mathcal{M}$
iff the reachable terminal states are contained
in trivial strongly connected components
(i.e. reduced to a state without loop)



Furthermore, it can be checked without computing the strongly connected components by two depth first searchs.

Complexity of LTL model-checking

- $\mathcal{P}$-*space* complete (with on-the-fly techniques)
- performed in $O(|\mathcal{M}| \cdot \exp(|f|))$ time

Model-checking CTL* formulas

Model-checking formula f on model $\mathcal{M}$

- Eliminate the operator **E** by the equivalence:
- **E**g⇔¬**A**¬g
- Build the syntactic tree of f
- Model-checking maximal LTL subformulas of f bottom-up
- Create intermediate "atomic properties" [**A**f'] corresponding to subformulas Af'

f=**A**(**FAG**p∧**GAF**q)

Check for each state **G**p, f becomes **A**(**F**[**AG**p]∧**GAF**q)

Check for each state **F**q, f becomes **A**(**F**[**AG**p]∧**G**[**AF**q])

Check the latter formula

Complexity of CTL* model-checking

- *P-space* complete (with on-the-fly techniques)
- performed in $O(|\mathcal{M}|.\exp(|f|))$ time

Finite system verification

Managing the complexity of finite state systems

Condensed representation of the states

- Using decomposition in components with indexed tables of component states *(decrease the size of a state)*
- Representing set of states instead of states *(BDD)*

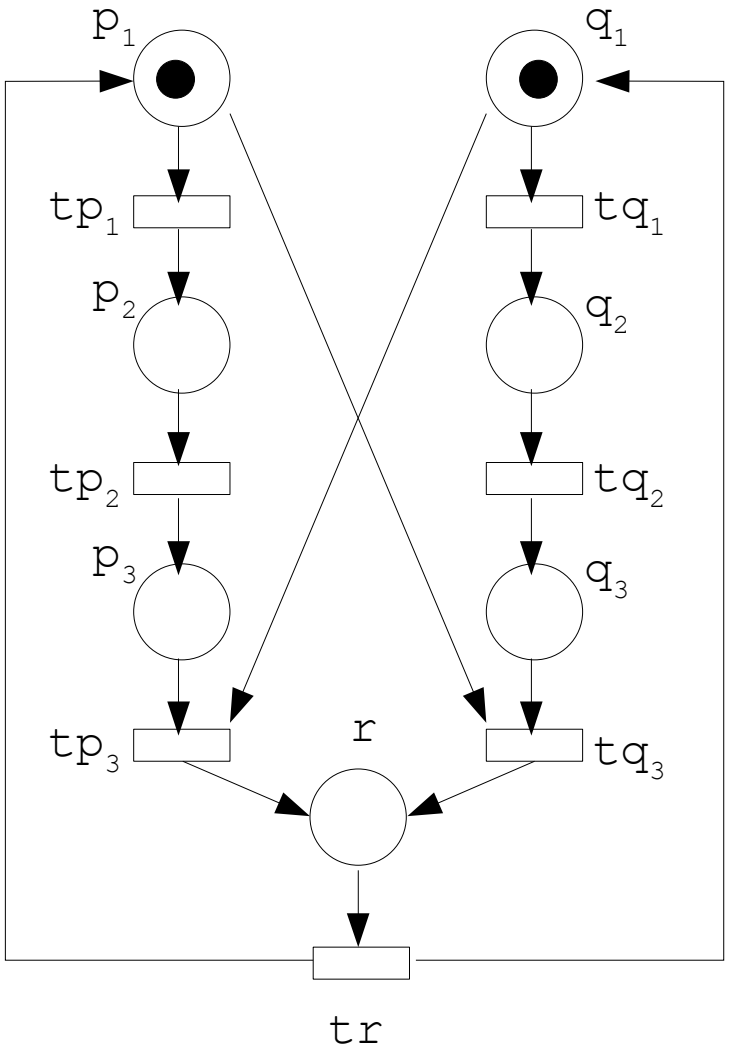
Independence between concurrent events

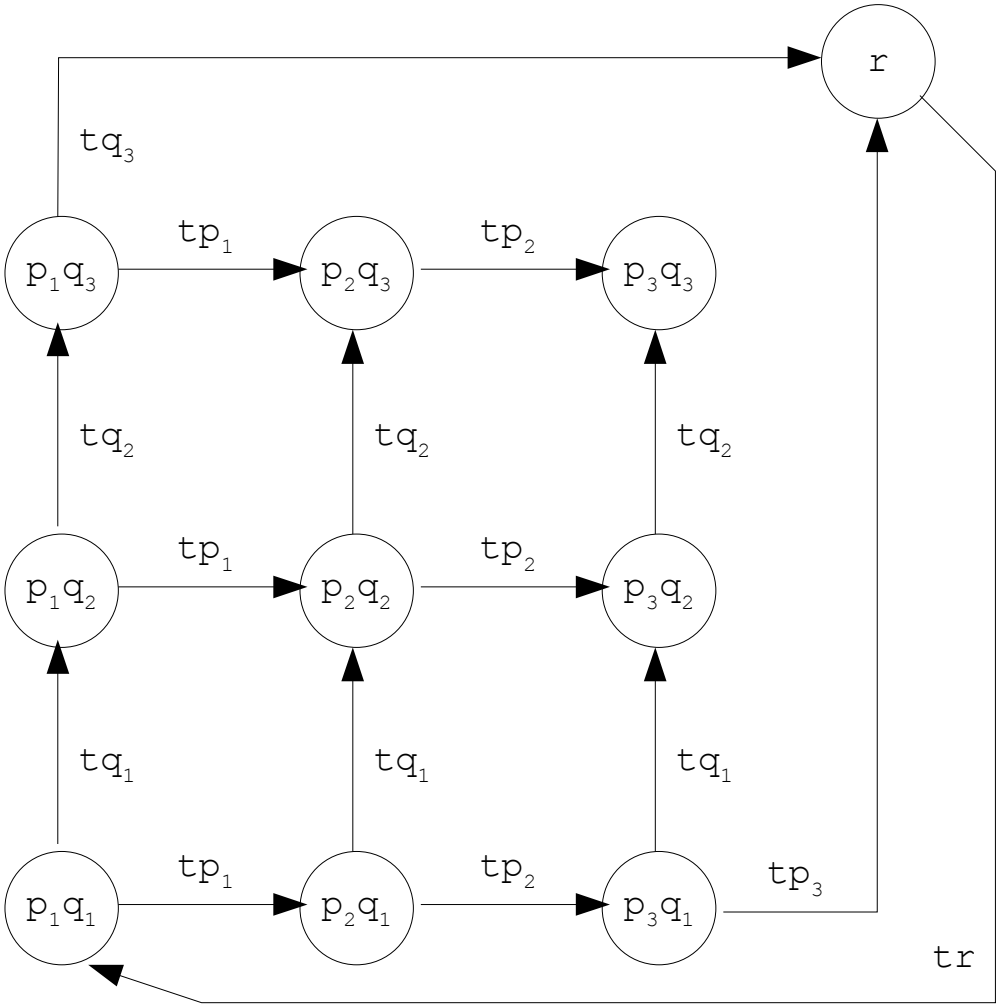
- Partial order methods *(reachability subgraph)*
- Unfoldings *(implicit representation of states by combination of compatible component states)*

Symmetrical systems

- Equivalence relation between states yielding symbolic states as equivalence classes
- Symbolic reachability graph

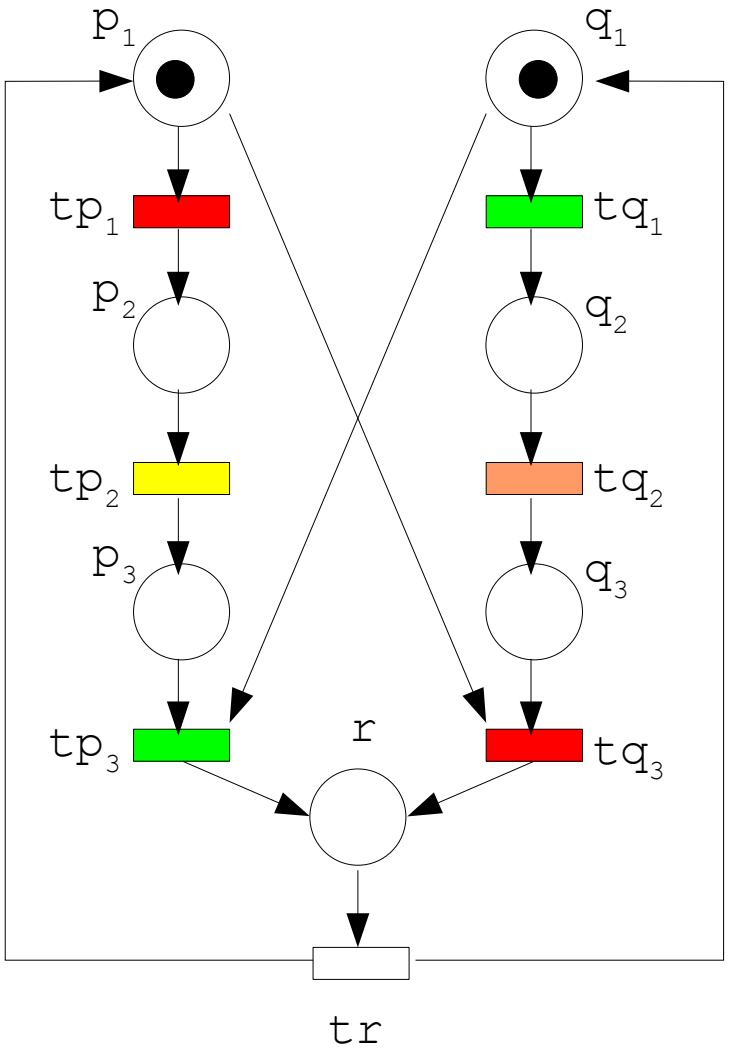
A Petri net and its reachability graph





Conflicts between transitions

$tp_1 \ \# \ tq_3$



Reduction by sleep sets (1)

Order the transition set and check the firing rule in ascending order. Manage a sleep set `S(m)`, by marking `m` (empty for the initial marking).

Let $m \rightarrow_t m'$, then `S(m')` defined from `S(m)` by:

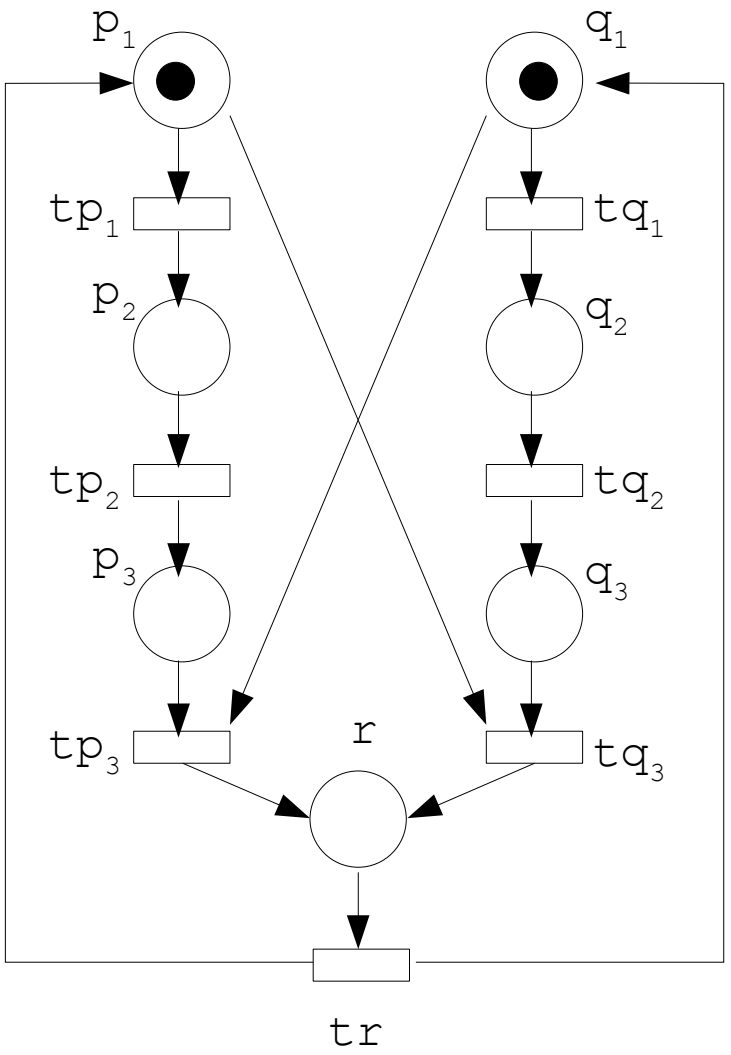
- Delete every transition `t'` such that `t#t'`,
- Add every transition `t'` such that `t#t'`, `t<t'` and $m \rightarrow_t m'$

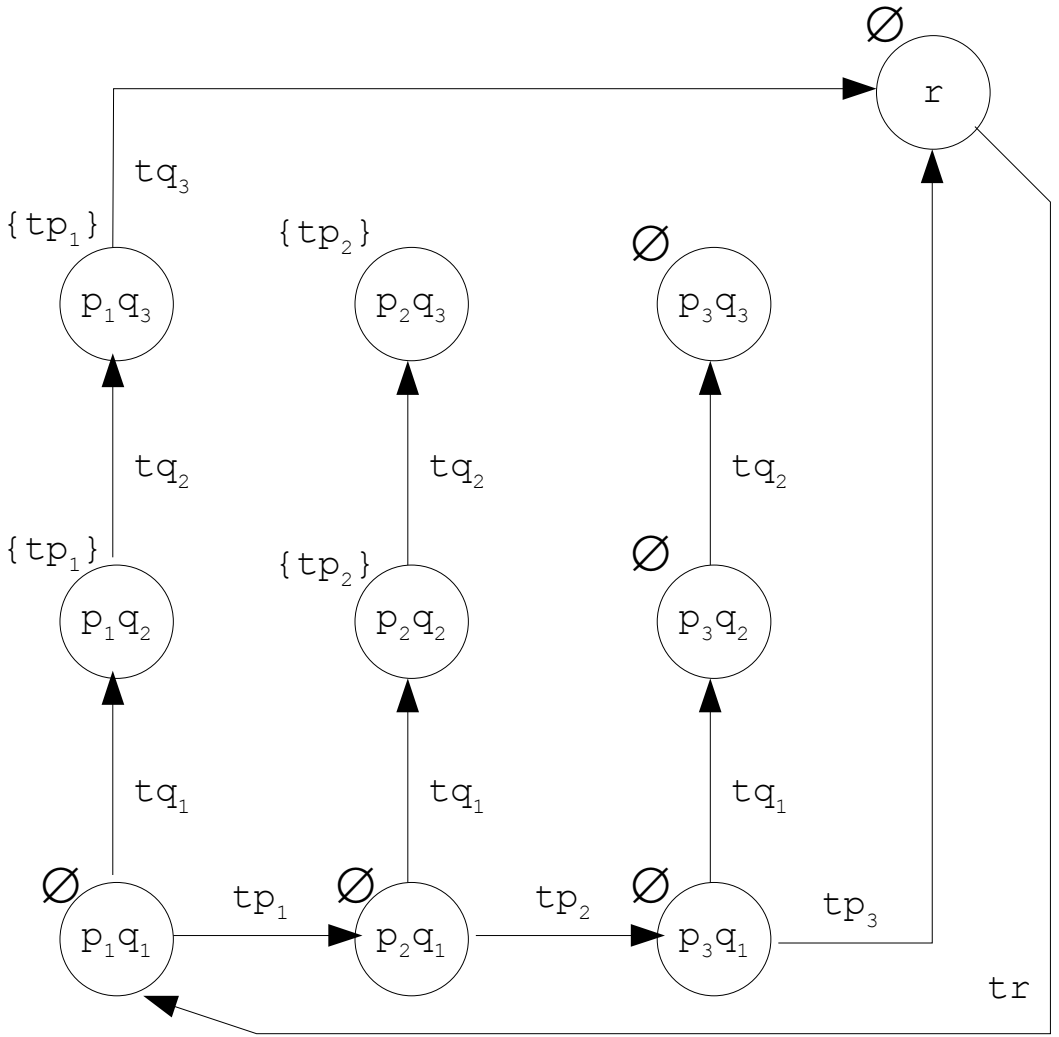
Rule: Ignore transitions from the sleep set of `m`

Property: A marking reached by an ignored transition has already been reached.

Consequence: The modified firing rule preserves reachability and deadlocks (requiring that the sleep set must be empty).

Reduction by sleep sets (2)





Reduction by stubborn sets (1)

Let m be a non dead marking, then T' is a stubborn set if

- There is a fireable transition in T',
- When $t \in T'$ and $t \# t'$ then $t' \in T'$,
- When $t \in T'$ and t is not fireable then there is a place p, input of t, such that every transition input of p belongs to T'

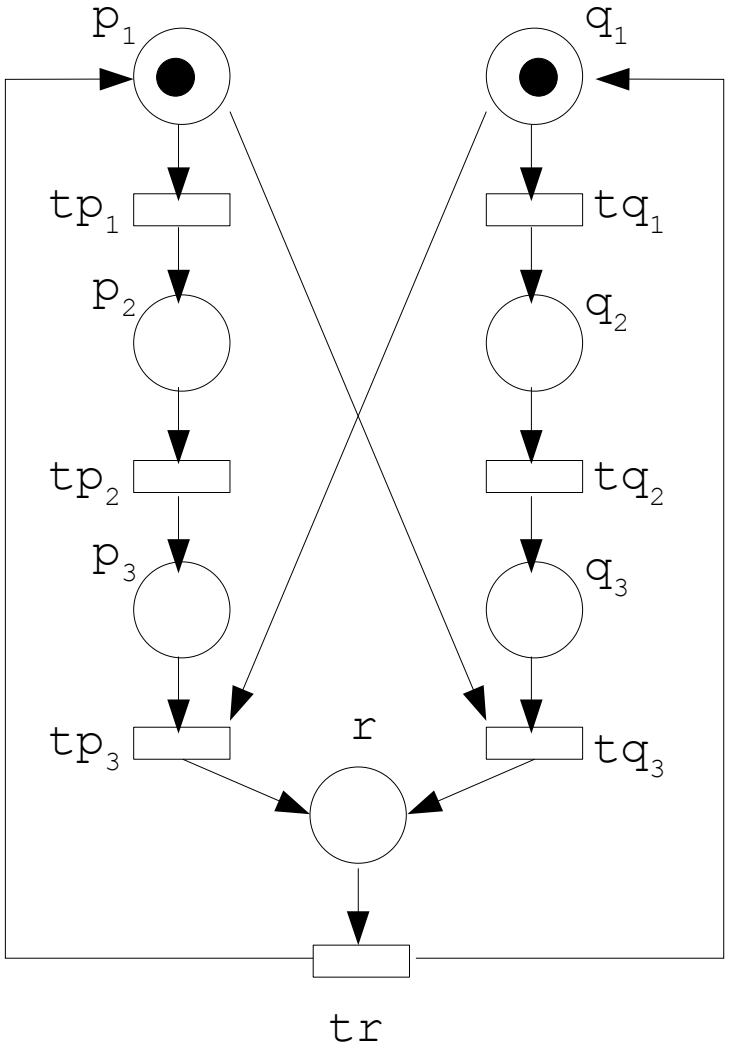
Rule: Fire from m only transitions in T'

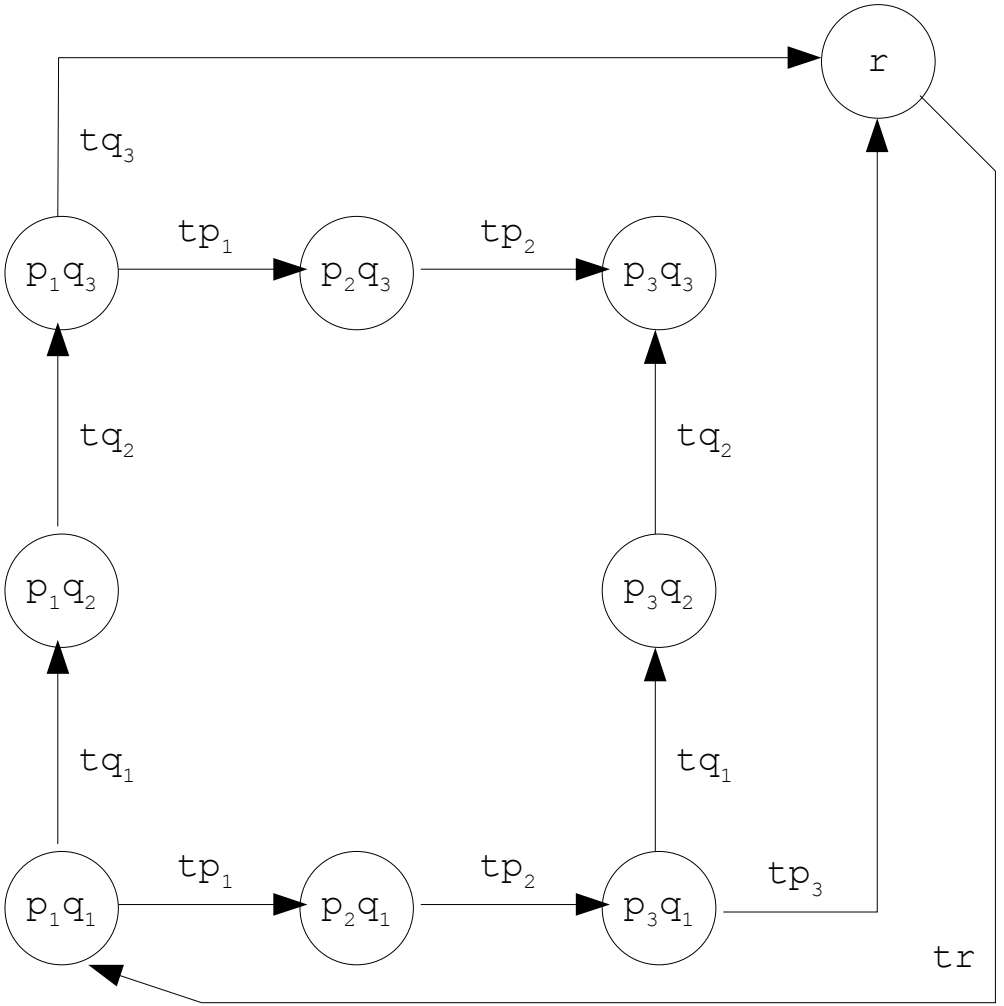
Property: A fireable transition not belonging to T' will remain fireable until its (possible) firing.

Consequence: The modified firing rule preserves deadlocks.

Reduction by stubborn sets (2)

$tp_1 \in T' \Rightarrow tq_3 \in T' \Rightarrow tq_2 \in T' \Rightarrow tq_1 \in T'$





Infinite system verification

When infinite-state systems?

Asynchronous systems

- unbounded channels

Dynamical systems

- creation of processes
- evolution of topology

Parametrized systems

- by the number of processes or resources,
- open systems where the environment is the parameter

Timed systems

Timed systems

Why time in models?

- In telecommunication protocols, time-out are necessary to face faulty environments.
- In critical systems, time is necessary to schedule, suspend or abort tasks.
- In information systems, time is necessay to evaluate the accuracy of a data.

Why time in property languages?

- Bounded time response which expresses temporal relation between events.
- Termination in bounded time which corresponds to absolute time of the execution.

Timed automata: syntax

A timed automaton (TA) is:

- a finite automaton
- enlarged with a set of *clocks* (X) which evolve synchronously and continuously.

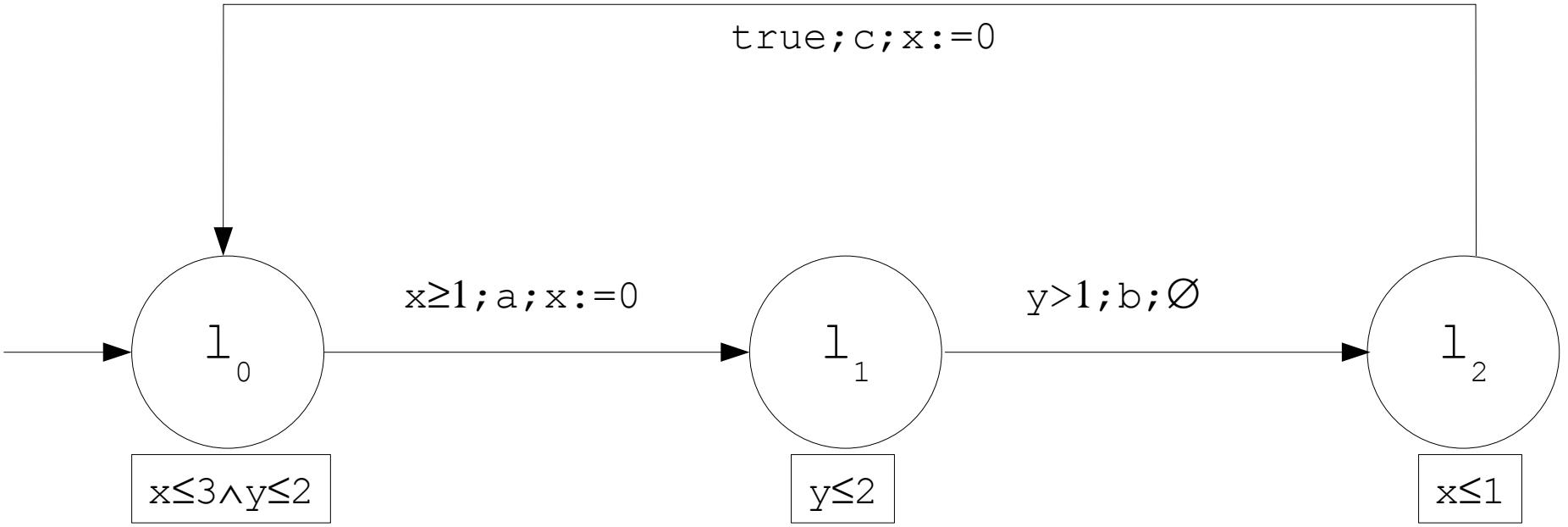
States (*locations*) of an automaton include *invariants* which restrict the way time may elapse in a location and whose syntax is:

- $\bigwedge_{x \in X'} x \sim c$ where $\sim \in \{<, \leq\}$ and c is some constant

Transitions of an automaton include clock *resets* and *guards* which restrict the temporal occurrence of a transition and whose syntax is:

- $\bigwedge_{x \in X'} x \sim c$ where $\sim \in \{>, <, \leq, \geq\}$ and c is some constant

Timed automata: an example



Timed automata: semantics

A configuration of timed automaton is given by:

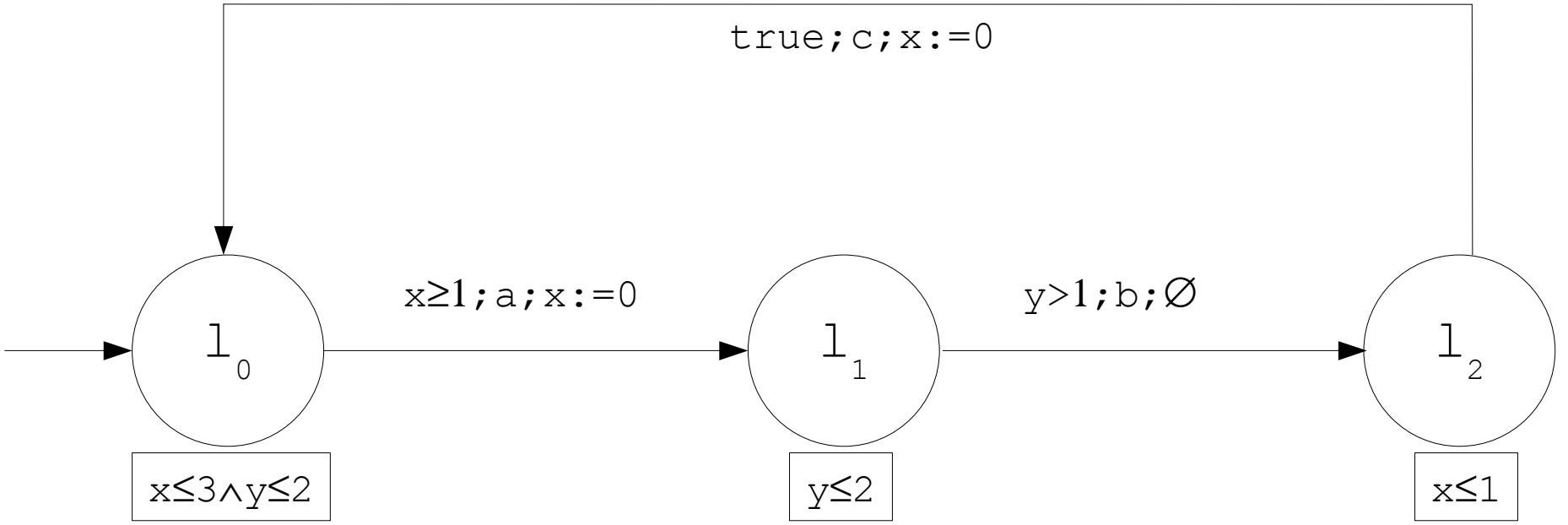
- a location (`l`),
- a value $\mathbf{v}(x)$ per clock satisfying the location invariant ($\mathbf{v} \models Inv(l)$).

A configuration may evolve by a delay `d` such that the invariant is still satisfied: ($\mathbf{v}+d \models Inv(l)$) with $(\mathbf{v}+d)(x) = \mathbf{v}(x)+d$

A configuration may change by a transition `(l,g,a,R,l')` to a configuration $(l', \mathbf{v}')$ iff:

- $\mathbf{v} \models g$
- if $x \in R$ then $\mathbf{v}'(x) = 0$ else $\mathbf{v}'(x) = \mathbf{v}(x)$
- $\mathbf{v}' \models Inv(l')$

Example continued



	l_0	1.4	l_0	a	l_1	b	l_2	0.3	l_2	c	l_0
x	0.0		1.4		0.0		0.0		0.3		0.0
y	0.0		1.4		1.4		1.4		1.7		1.7

What happens next?

Reachability analysis: the key idea

The number of (reachable) configurations is infinite (and even uncountable). So one wants to partition configurations into *regions* such that:

1. Two configurations in a region allow the same transitions and the new configurations belong to the same region.

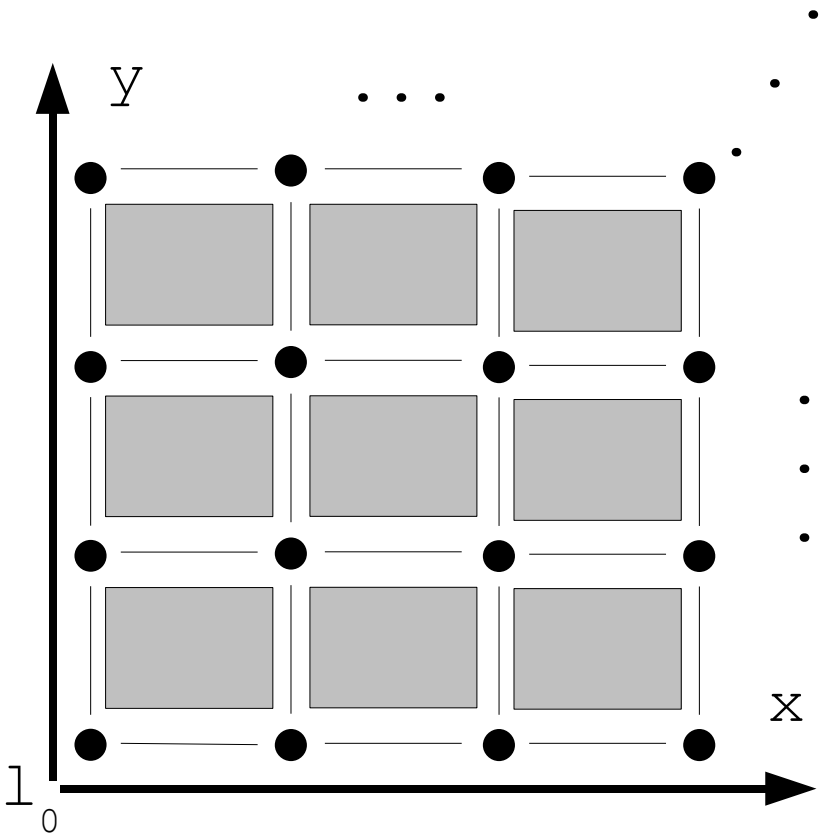
2. If a configuration in a region letting time elapse reaches a new region every other configuration may reach the same region by time elapsing.

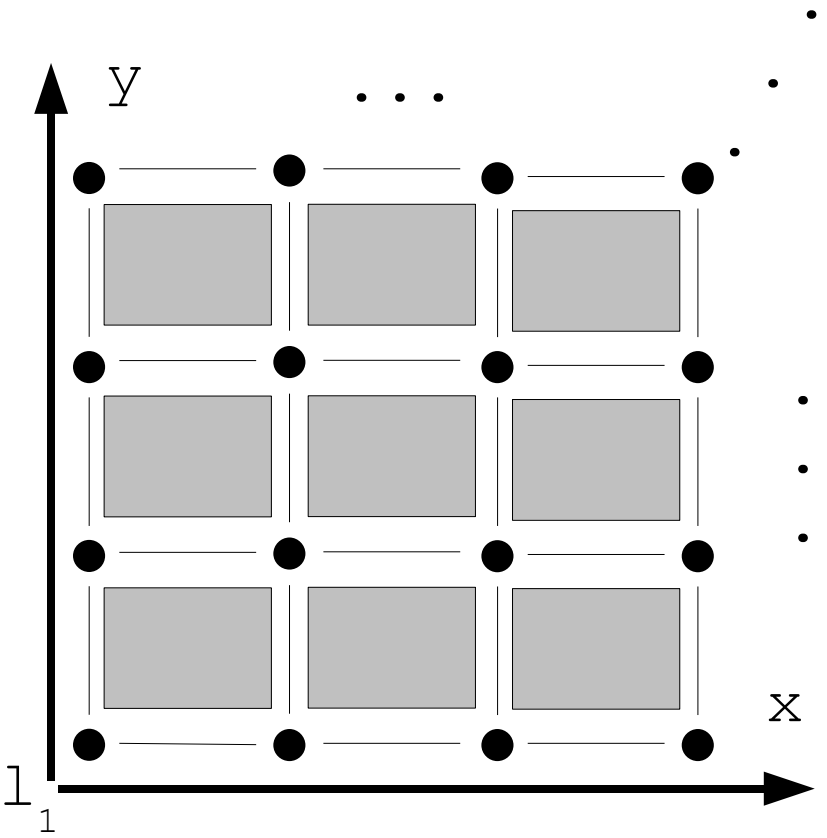
3. There is a finite representation of a region such that the discrete and time successors of the region are computable.

4. The number of regions is finite.

A first partition
(two clocks and two locations)

Guards and invariants check integer values

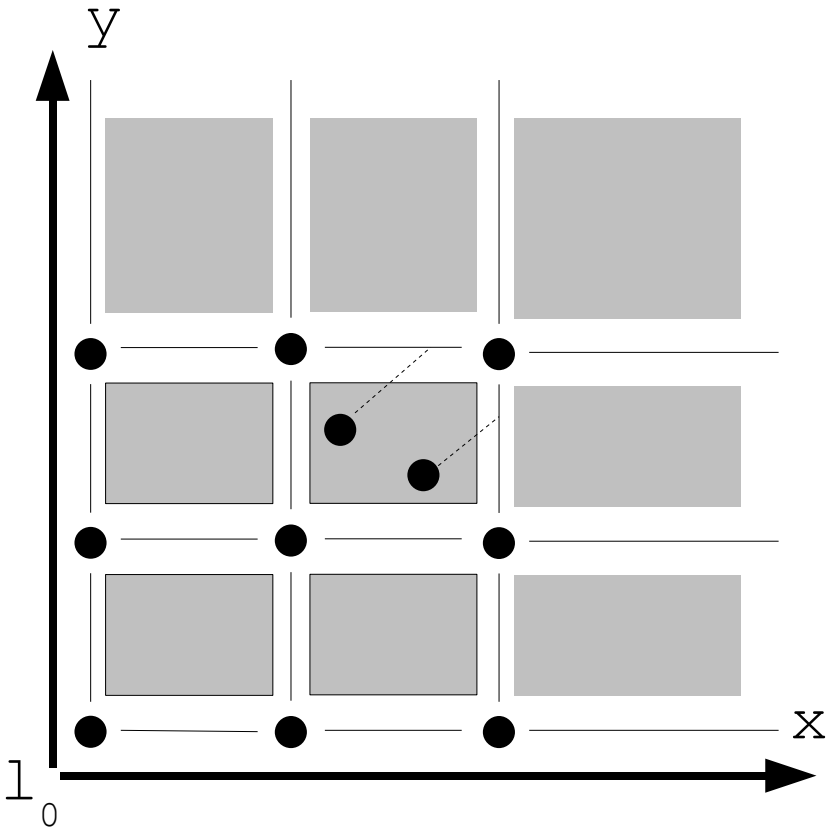


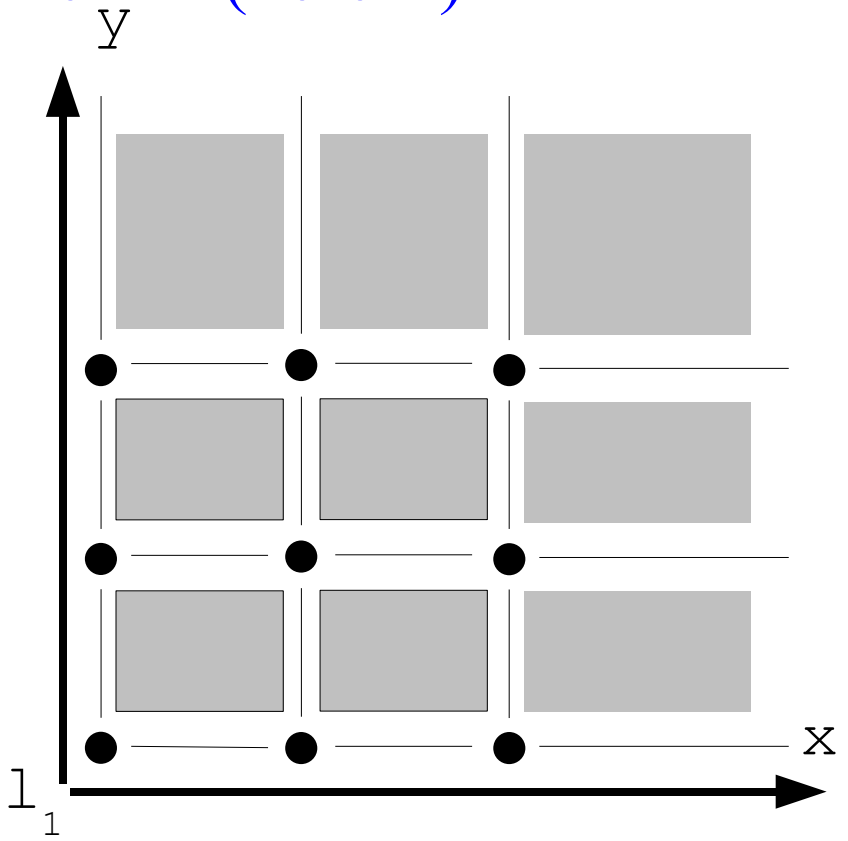


Why this partition is not appropriate?

A second partition
(two clocks and two locations)

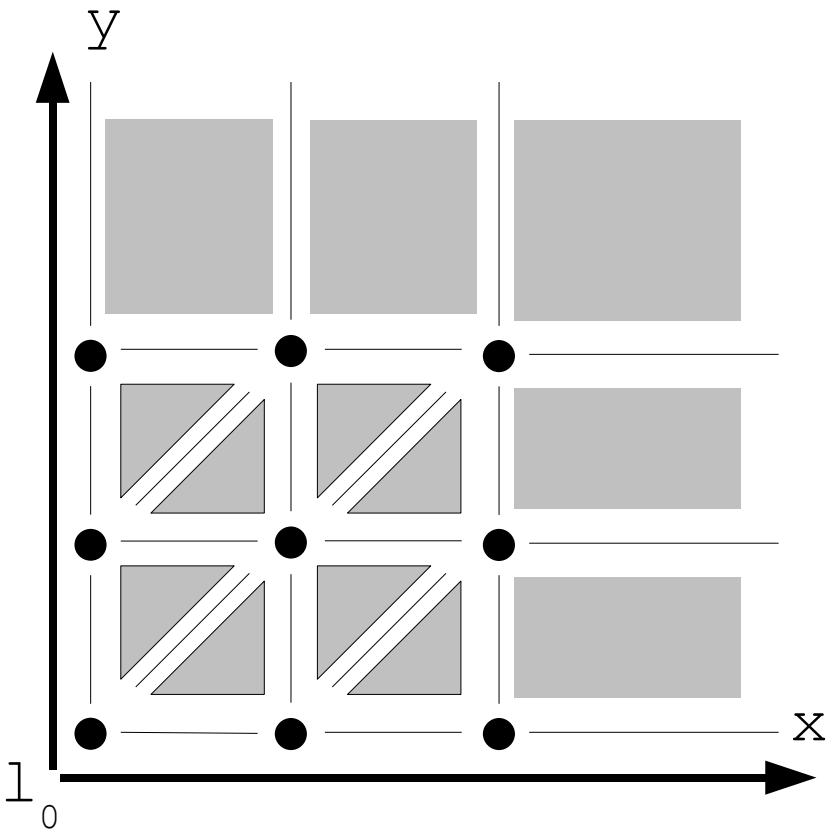
The exact value of a clock is irrelevant when it is beyond the maximal constant of the TA (here 2)

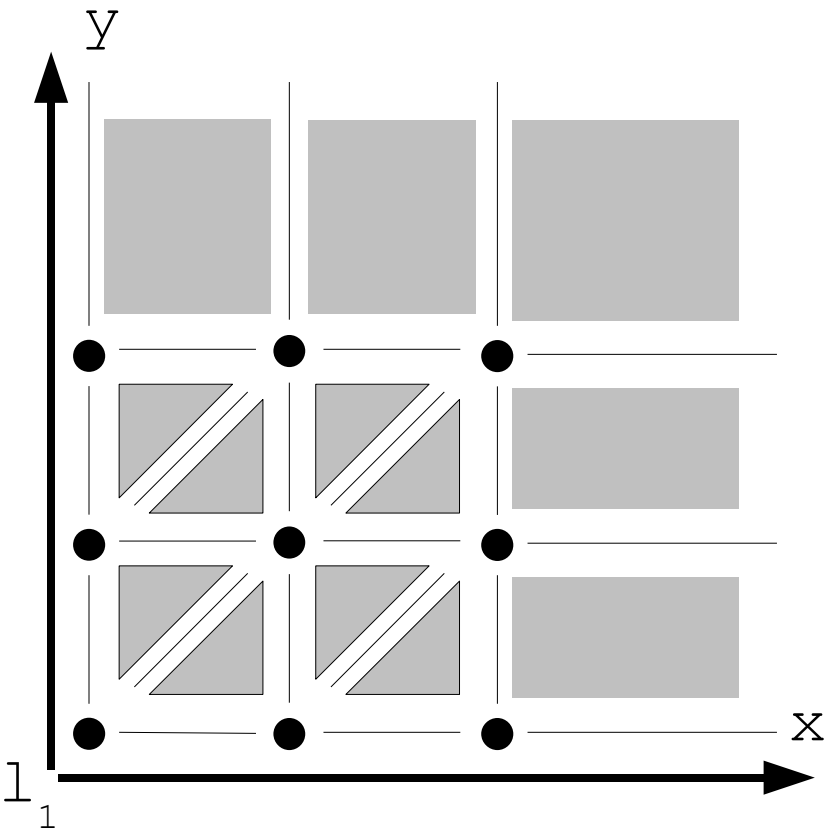




Why this partition is not appropriate?

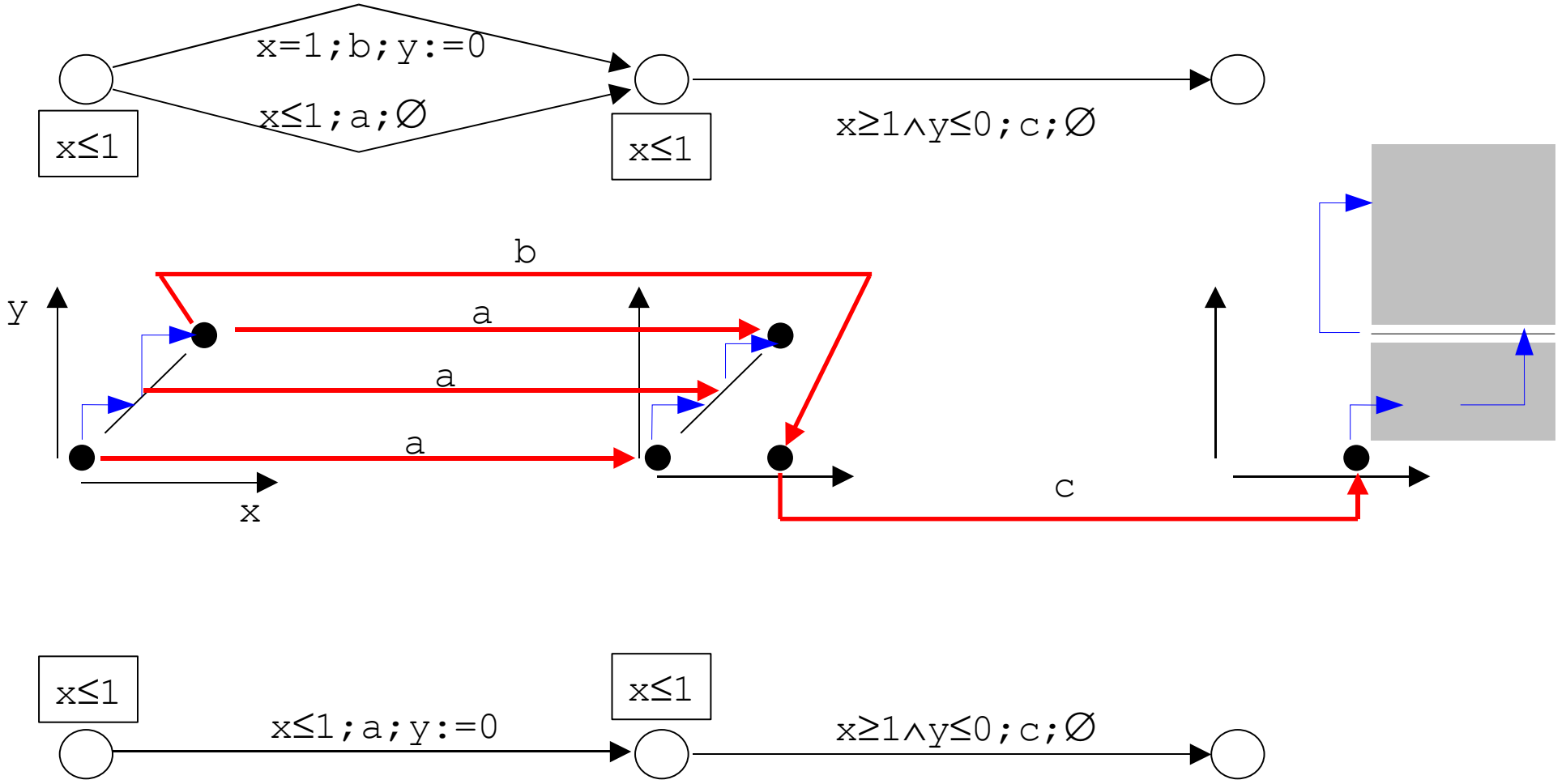
A third partition
(two clocks and two locations)





Check that this partition is appropriate

The region graph: illustration



Build the region graph of the above TA