Concurrency

A concurrent program is non-deterministic and potential parallelism (time sharing when there is a single processor).

Arbitrary interleaving: (interference, an archetypal safety property))

```
process P:
    int i;
    for i := 1 to 10 do
        n := n+1;
```

```
process Q:
    int i;
    for i := 1 to 10 do
        n := n+1;
```

The both processes will do load n first then do n += 1, then do put n back. If both process P and Q take n = 1 simultaneously then the result will be n = 2 after this turn.

Deadlock: a situation where a set of processes are unable to make any further progress, because of mutually incompatible demands they make of shared resources. (Archetypal Liveness property)
Livelock: it is spinning while waiting for a condition that will never become true.
Either can happen if concurrent processes or threads are mutually waiting for each other.

The Coffman conditions

Four conditions (the Coffman conditions) that all must occur for deadlock to happen:

1. Serially reusable resources: the processes involved must share some reusable resources between themselves under mutual exclusion.
2. Incremental acquisition: processes hold on to resources that have been allocated to them while waiting for additional resources.
3. No preemption: once a process has acquired a resource, it can only release it voluntarily—it cannot be forced to release it.
4. Wait-for cycle: a cycle exists in which each process holds a resource which its successor in the cycle is waiting for.

Desired properties

Mutual exclusion: only one process may be active in its critical section at a time.
No deadlock: if one or more processes are trying to enter their critical section, one must eventually succeed.
No starvation: if a process is trying to enter its critical section, it must eventually succeed. (There's no process that continuously using shared data if any process is waiting for that piece of data). **Be careful! If the first process runs indefinitely or dies in non-critical part, the other process may be starving if it's waiting for the first process to change flag.**
Handles lack of contention: if only one process is trying to enter its critical section, it must succeed with minimal overhead.

Dekker's algorithm

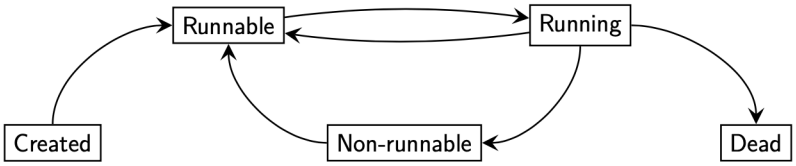
```
static int turn = 1; static int p = 0; static int q = 0;
```

```
while (true) {
p0: non_critical_P();
p1: p = 1;
p2: while (q != 0) {
p3:   if (turn == 2) {
p4:     p = 0;
p5:     while (turn == 2);
p6:     p = 1;
      }
    }
p7: critical_P();
p8: turn = 2;
p9: p = 0;
}
```

```
while (true) {
q0: non_critical_Q();
q1: q = 1;
q2: while (p != 0) {
q3:   if (turn == 1) {
q4:     q = 0;
q5:     while (turn == 1);
q6:     q = 1;
      }
    }
q7: critical_Q();
q8: turn = 1;
q9: q = 0;
}
```

All four property respected.
Cons: Dekker's algorithm has proved hard to generalise to programs with more than two processes.

A thread that is alive is always in one of three states:
running: it is currently executing;
runnable: it is currently not executing but is ready to execute; or
non-runnable: it is not running and is not ready to run—may be waiting on some input or shared data to become unlocked.



Semaphore: a simple but versatile concurrent device for managing access to a shared resource. It consists of a value v ∈ N of currently available access permits, and a wait set W of processes currently waiting for access. It must be initialized S := (k, { }), where k is the maximum number of threads can simultaneously access some resource. e.g. a reception of hotel that gives keys, when a thread want private data it asks for the key (wait), when it leaves it bring key back (signal).
Assume process p executes the operation:

S.wait()

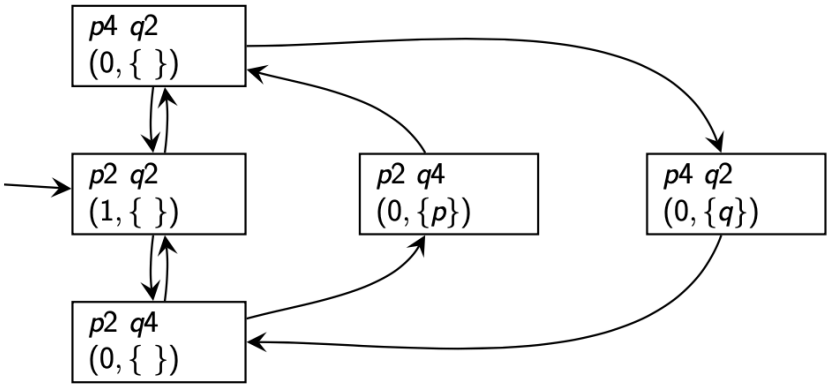
```
if S.v > 0
    S.v--
else
    S.W = S.W U {p}
    p.state = blocked
```

S.signal()

```
if S.W == { }
    S.v++
else
    choose q from S.W
    S.W = S.W \ {q}
    q.state = runnable
```

Hence when p signals S whose wait set is empty, S's value is incremented; otherwise an arbitrary process is unblocked, that is, removed from the wait set, having its state changed to runnable.
Binary semaphore: If S.v ∈ {0, 1}, S is called binary. Sometimes a binary semaphore is called a mutex.

State diagrams: used to describe the behavior of system. They are directed graphs, where nodes represent states and edges represent transitions, that is, state changes.
Mutex problem for 2 processes p, q, p2: S.wait(); q2: S.wait(), p4: S.signal(); q4: S.signal()



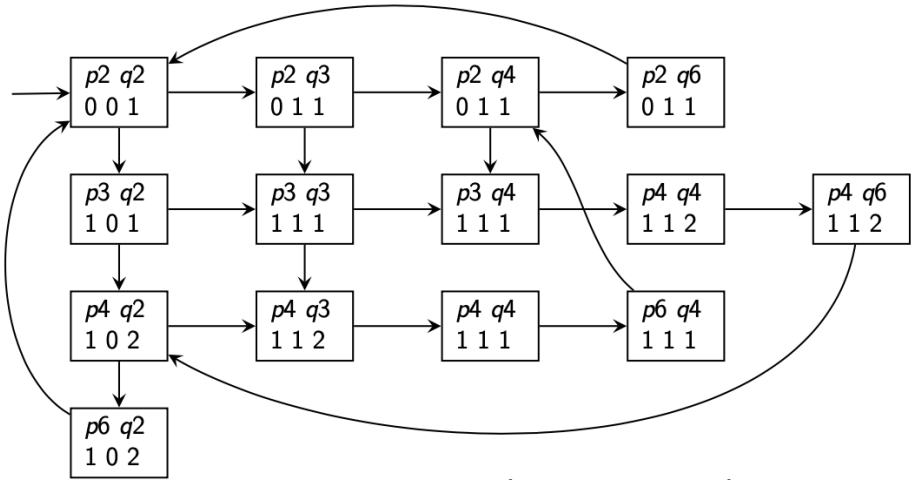
When process number > 2, there is no longer a guarantee of freedom from starvation, because blocked processes are taken arbitrarily from a set. The obvious (fair) way of implementing semaphores is to let processes wait in a queue to avoid starvation. When doing this, it is a strong semaphore.

Peterson's mutex algorithms

```
static int p = 0; static int q = 0; static int turn = 1;
```

```
while (true) {
p1: non_critical_p();
p2: p = 1;
p3: turn = 2;
p4: while (q && turn == 2);
p5: critical_p();
p6: p = 0;
}
```

```
while (true) {
q1: non_critical_q();
q2: q = 1;
q3: turn = 1;
q4: while (p && turn == 1);
q5: critical_q();
q6: q = 0;
}
```



Where each state block stands for (pi, qj, p, q, turn).
Mutex achieved: all states of form (p6, q6, _, _, _) are unreachable. No state of form (p4, q4, _, _, _) is stuck, so there is no deadlock. From each state of form (p4, _, _, _, _), a state (p6, _, _, _, _) can be reached, and similarly for q4, so there is no starvation.

Threads in Java:

To create a thread class and then create an instant:

First way: public class MyThread extends Thread{
    public void run(){
        //insert things to do here
    }
}

Thread myThread = new MyThread()

Second way: used when a class has extend a class
Class MyRunnable implements Runnable{
    public void run(){
        //insert things to do here
    }
}

Thread myRunnable = new Thread(new MyRunnable());

To run a thread: myThread.start()
To suspend a thread (running to non-runnable for a while): myThread.sleep(long milliseconds)
Test a thread is running: myThread.isAlive()
From running to runnable: myThread.yield()
Calling t.join() suspends the caller until thread t has completed.

Synchronized methods/objects: mark that method or object as part of the critical region. Only one process can execute or modify in the critical region at any one time.

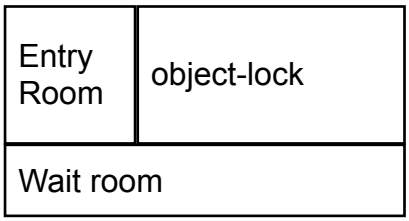
```
class SynchedCounter extends Counter {
  synchronized void increment() {
    int temp = value;
    try { Thread.sleep(1); }
    catch (InterruptedException e) {}
    value = temp + 1;
  }
}
```

synchronized(object_name) to declare the entire object is sychronized, mutually exclusive. However, it's **not suggested**. It requires the user of the shared object to lock the object, rather than placing this inside the shared object and encapsulating it. If a user fails to lock the object correctly, race conditions can occur.

Monitors: a set of synchronized methods and data (an object or module) that queue processes trying to access the data. In Java, a monitor is an object that encapsulates some (private) data, with access to the data only via synchronized methods. It manages the blocking and unblocking of processes that vie for access.

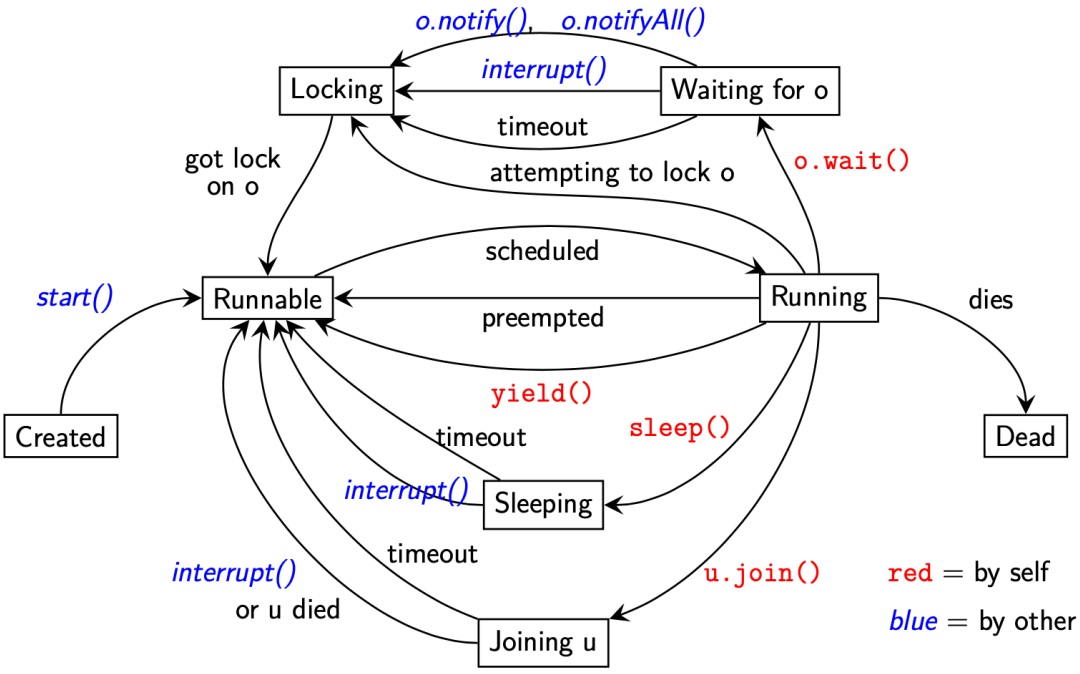
void wait(): Causes the current thread to wait until another thread invokes the notify() method or the notifyAll() method for this object.
void notify(): Wakes up a single thread that is waiting on this object's lock (the choice of thread that is awoken is arbitrary (random), so never code on prediction of executing order).
void notifyAll(): Wakes up all threads that are waiting on this object's lock.

Volatile: Declaring a variable volatile directs the virtual machine to re-load the value of a variable every time it needs to refer to it.



When a process want to use a shared object, it goes to entry room
Entry -> object-lock: random order, but always only 1 in object-lock object at the same time
Object-lock -> exit: finish using shared object
wait -> waiting room: calling wait()
wait -> entry: calling notify() or notifyAll()

Java thread methods that changes state



FSP - LTSA - Syntax

Finite State Process (FSP) LTS: labelled transition system

FSP allows the description of one or more processes that operate independently, but which may synchronise at various points. Each process model in FSP consists of a set of atomic actions (the "alphabet") that can occur in that process, and a definition which specifies the legal sequences of atomic actions in the system.

LTSA app operations

click blue **C** to compile, click 'draw' to see state diagram. On top right there are choice to make default animation.(click **A**)
Click **||** button to compose
Click checkSafety can determine deadlock
Select the checkProgress to check liveness
Check -> LTL property -> <fluent name> to check LTL properties

The **action prefix operator** "->", is fundamental: "If x is an action and P a process then the action prefix x -> P describes a process that initially engages in action x and then behaves exactly as described by P."
TRAFFIC_LIGHT = (green->yellow->red->TRAFFIC_LIGHT).
The "->" operator always has an atomic action as the left operand, and a process as the right operand. In this case, green is the left operand, and yellow -> red -> TRAFFIC_LIGHT is the right operand (the process). TRAFFIC_LIGHT is the name of the process. Repetitive behaviour is captured by recursion. Atomic actions use lower case; process names use upper case.

The **choice operation**, "|" is used to describe a process that can execute more than one possible sequence of actions. "If x and y are actions, then (x -> P | y -> Q) describes a process which initially engages in either of the actions x or y. After the first action has occurred, the subsequent behaviour is described by P if the first action was x and Q if the first action was y."
This describes a traffic light with a button for a pedestrian, which turns the light red. If the button is not pushed, the light remains green:
TRAFFIC_LIGHT = (button -> red -> TRAFFIC_LIGHT | none -> green -> TRAFFIC_LIGHT).
To complicate matters regarding input, output, and choice, FSP allows **non-deterministic choice**.
"Process (x -> P | x -> Q) describes a process that engages in x and then behaves as P or Q."
Note that x is the prefix in both options of the choice.

To model a process that can take multiple values, **indexed processes** can be used. In an indexed process, variables can be used to increase the expressive power of FSP.
BUFF = (in[i:0..2] -> out[i] -> BUFF).
This is equivalent to:

```
BUFF    =   (in[0] -> out[0] -> BUFF
            |in[1] -> out[1] -> BUFF
            |in[2] -> out[2] -> BUFF).
```

Declare constants

```
const N = 3
range T = 0..N
BUFF = (in[i:T] -> STORE[I]),
STORE[i:T] = (out[i] -> BUFF).
```

In the above example, the maximum value of 3 is declared as a constant N, and the range T is used in two places. If the maximum value increases to 5, we need only change N.

A **guarded action** allows a context condition to be added to options in a choice.
“The choice (when B x -> P | y -> Q) means that when the guard B is true, then the actions x and y are both eligible to be chosen, otherwise if B is false, then action x cannot be chosen.” The boolean value B can contain && as and, || as or

```
COUNT (N=3) = COUNT[0],
COUNT[i:0..N]   =   (when(i<N) inc ->COUNT[i+1]
                    |when(i>0) dec ->COUNT[i-1]
                    ).
```

The **STOP** process is a special, pre-defined process that engages in no further actions. It is used for defining processes that terminate.

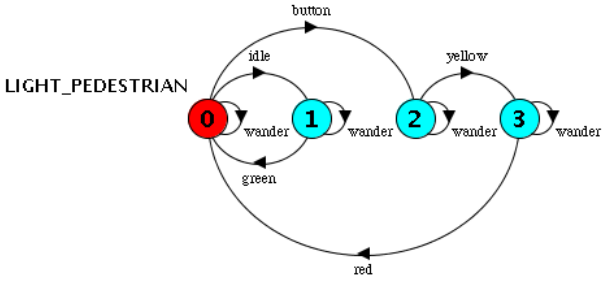
```
ONESHOT = (once -> STOP).
```

The **parallel composition operator** allows us to describe the concurrent execution of two processes. “If P and Q are processes, then (P || Q) represents the concurrent execution of P and Q.”

```
ITCH = (scratch->STOP).
CONVERSE = (think->talk->STOP).
||CONVERSE_ITCH = (ITCH || CONVERSE).
```

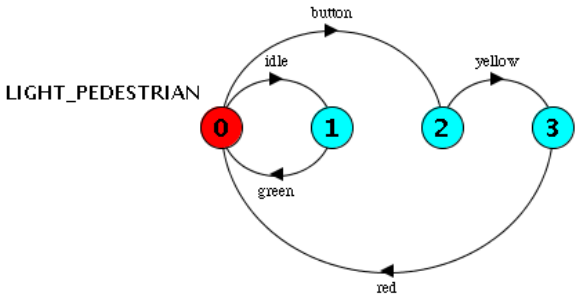
The FSP semantics specify that the two processes will interleave. Possible interleavings are: think -> talk -> scratch; think -> scratch -> talk; scratch -> think -> talk.
FSP insists that when a process P is defined by parallel composition, the name of the composite process is prefixed with vertical bars: ||P.
“If the processes in a composition have actions in common, these actions are said to be **shared**. This is how process interaction is modeled. While unshared actions may be arbitrarily interleaved, a shared action must be executed at the same time by all processes that participate in that shared action.”

```
TRAFFIC_LIGHT = (button -> YELLOW | idle -> GREEN),
GREEN = (green -> TRAFFIC_LIGHT),
YELLOW = (yellow -> RED),
RED = (red -> TRAFFIC_LIGHT).
PEDESTRIAN = ( button -> PEDESTRIAN
                    | wander -> PEDESTRIAN
                    ).
||LIGHT_PEDESTRIAN = (TRAFFIC_LIGHT || PEDESTRIAN).
```



we can use the **relabelling** operator when composing the two. “Given a process P, the process P/{new1/old1, ..., newN/oldN} is the same as P but with action old1 renamed to new1, etc.”

```
TRAFFIC_LIGHT = (button -> YELLOW | idle -> GREEN),
GREEN = (green -> TRAFFIC_LIGHT),
YELLOW = (yellow -> RED),
RED = (red -> TRAFFIC_LIGHT).
PEDESTRIAN = ( button -> PEDESTRIAN
                    | wander -> PEDESTRIAN
                    ).
||LIGHT_PEDESTRIAN =
        (TRAFFIC_LIGHT || PEDESTRIAN/{idle/wander}).
```



Client-server examples
CLIENT = (call -> wait -> continue -> CLIENT).
SERVER = (request -> service -> reply -> SERVER).
||CLIENT_SERVER = (CLIENT || SERVER) /{call/request, reply/wait}.
Writing a number of different definitions for multiple similar processes would be cumbersome and error prone, so instead, we use process labels. "**a:P** prefixes each action label in P with a". To describe two clients executing concurrently, use:
||TWO_CLIENTS = (a:CLIENT || b:CLIENT).
An array of prefix labelled processes can be described using parameterised processes:
||N_CLIENTS(N=3) = (c[i:1..N]:CLIENT).
or the equivalent definition:
||N_CLIENTS(M=3) = (forall[i:1..M] c[i]:CLIENT).

Process labelling
"{a1,..,ax}::P replaces every action label n in the alphabet of P with the labels a1.n,..,ax.n. Further, every transition n->X in the definition of P is replaced with the transitions ({a1.n,..,ax.n} -> X)"
({a1,..,ax} -> X) is just shorthand for the set of transitions (a1 -> X), ..., (ax -> X).
Now we can compose two clients and a server as follows:
||TWO_CLIENT_SERVER
= (a:CLIENT || b:CLIENT || {a,b}::(SERVER/{call/request, wait/reply})).
We can compose N clients and one server following this pattern:
||N_CLIENT_SERVER(N=2)
= (forall[i:1..N] (c[i]:CLIENT || {c[i:1..N]}::(SERVER/{call/request, wait/reply})).

FSP—Variable hiding
'Given a process P, the process P\{a1,..,aN} is the same as P, but with actions names a1,..,aN removed from P, making these silent. Silent actions are named tau and are never shared".
An alternative is to list the variables that are not to be hidden: 'Given a process P, the process P@{a1,..,aN} is the same as P, but with actions names other than a1,..,aN removed fromP".
Hence the following two definitions result in the same LTS:
SERVER_1 = (request -> service -> reply -> SERVER_1) @{request, reply}.
SERVER_2 = (request -> service -> reply -> SERVER_2) \{service}.

Action priority
FSP has operators for action priority. These allow us to express a form of scheduling policy for a model.
The high priority operator P<<{a1,...,aN} specifies that actions a1,...,aN have a higher priority than all other actions in P.
The low priority operator P>>{a1,...,aN} specifies that actions a1,...,aN have a lower priority than all other actions in P.
P = (a -> b -> P | c -> d -> P). ||HIGH = P<<{a}.
Is equivalent to HIGH = (a -> b -> HIGH).

FSP - LTSA - Safety

A number counter example (alphabet extension, safety)

```
const N = 4
range T = 0..N
VAR = VAR[0],
VAR[u:T] = (read[u]->VAR[u] | write[v:T]->VAR[v]).
CTR = ( read[x:T] ->
                ( when (x<N) increment -> write[x+1] -> CTR
                | when (x==N) end -> END
                )
        )+{read[T],write[T]}.
||SHAREDCOUNTER = ({a,b}:CTR || {a,b}::VAR).
```

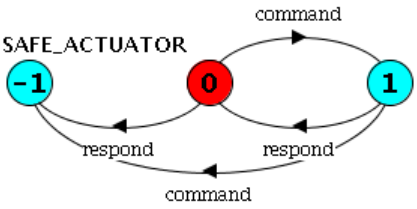
alphabet extension: +{read[T],write[T]}. This is to avoid write[0] can run anytime. Equivalent to set Actions = {read[T],write[T]} and call +Actions at the end of CTR.
However, this does not have deadlock but may be interleave. So we can write a new process INTERFERENCE = (a.write[v:T] -> b.write[v] -> ERROR) then use safety check

To avoid this, we can use a lock

```
LOCK = (acquire -> release -> LOCK).
CTR = (acquire -> read[x:T] ->
                ( when (x<N) increment -> write[x+1]
                                -> release -> CTR
                | when (x==N) release -> END
                )
        )+{read[T],write[T]}.
||LOCKED_SHAREDCOUNTER = ({a,b}:CTR || {a,b}::(LOCK||VAR)).
```

Safety properties: A safety property asserts that nothing “bad” happens during execution. A deadlock is an example of this.
Liveness properties: A liveness property asserts that some “good” eventually happens. For example, that all processes trying to access a critical section eventually do get access.

property SAFE_ACTUATOR = (command -> respond -> SAFE_ACTUATOR).
This **property** says that, whenever a command action is observed, a respond action should occur before another command action occurs. The error states are automatically generated. It must be concurrent into the system to enable.



e.g. ||CHECK_ACTUATOR = (ACTUATOR || SAFE_ACTUATOR).
e.g.

```
property SAFE_RW= ( acquireRead -> READING[1] | acquireWrite -> WRITING).
||SAFE_READERS_WRITERS
        = ( READERS_WRITERS
        || {reader[1..Nread], writer[1..Nwrite]}::SAFE_RW ).
```

FSP - LTSA - Liveness

Progress property

“progress P = {a1, a2, ..., aN} defines a progress property P that asserts that, in an infinite execution of a target system, at least one of the actions a1, a2, ..., aN will be executed infinitely often.”
e.g. COIN = (toss -> heads -> COIN | toss -> tails -> COIN).
progress HEADS= {heads}

TWOCOIN = (pick -> COIN | pick -> TRICK),
TRICK = (toss -> heads -> TRICK).
Note that the following progress property is not violated by the TWOCOIN process:
progress HEADSTAILS = {heads, tails}
This property holds because only one of the actions in the set needs to occur infinitely often. In the TWOCOIN process, the heads action will occur infinitely often because it occurs in both the normal coin and the trickster's coin.
Progress can have indexes
progress WRITE[i:1..Nwrite] = {writer[i].acquireWrite}

LTL - Linear temporal logic

Fluent
A fluent is a property that can change over time. Fluents allow us to describe properties of a system over its lifetime, rather than at just one instant.
FSP fluent syntax
fluent FL = <{s1,...,sN}, {e1,...,eN}>
s1,...,sN and e1,...,eN are actions. The fluent FL is the proposition, and FL becomes true when any of the actions in {s1,...,sN} occur, and then false again when any of the actions in {e1,...,eN} occur.

FL is initially false. If we want a fluent that is initially true, we can describe it using:
fluent FL = <{s1,...,sN}, {e1,...,eN}> initially 1
in which 1 represents true. 0 represents false.
e.g. fluent GREEN = <{green}, {yellow,red}> initially 1
That is, the fluent GREEN becomes true when the green action occurs, and false when yellow or red occur.

As with processes, we can define indexed fluents for two lights:
fluent GREEN[i:1..2] = <{green[i]}, {yellow[i],red[i]}> initially 1
Fluents can be combined using the propositional logic connectives:
&& (and), **||** (or), **!** (not), **->** (implication), **<->**(equivalence)
In addition, we have (bounded) universal and existential quantifiers:
forall [i:1..2] GREEN[i]
exists [i:1..2] GREEN[i]
e.g. Now we can express that we do not want the two lights to be green at the same time:
!forall[i:1..2] GREEN[i]
which is equivalent to
!(GREEN[1] && GREEN[2])
and is also equivalent to:
exists[i:1..2] !GREEN[I]

Temporal logic - Always
The linear temporal logic formula **[]F** (read always F) is true iff the formula F is true at the current instant and at every instant in the future.
e.g. The light is always green, yellow, or red:
assert ALWAYS_A_COLOUR = [](GREEN || YELLOW || RED)
Temporal logic - Eventually
The linear temporal logic formula **<>F** (read eventually F) is true iff the formula F is true at the current instant or at some instant in the future.
This fluent says that the light will eventually become red:
assert EVENTUALLY_RED = <>RED

Temporal logic - Until
The until operator allows us to specify that a certain property is true until another property becomes true. **F U G** is true iff G eventually becomes true, and F is true until that instant.
e.g. the light is initially green, and stays green until the button is pushed
assert INITIALLY_GREEN = (GREEN U button)
Temporal logic - next
The next operator allows us to specify that a certain property is true at the next instant: The linear temporal logic formula **X F** is true iff F is true at the next instant.
e.g. when the button is pushed, the light will go yellow at the next time instant:
assert BUTTON_TO_YELLOW = (button -> X YELLOW)

Some more examples
RESOURCE = (acquire -> release -> RESOURCE).
USER = (acquire -> use -> release -> USER).
||RESOURCE_SHARE = (a:USER || b:USER || {a,b}::RESOURCE).
(a) Both processes a and b eventually acquire the resource.
assert EVENTUALLY_ACQUIRE= (<>a.acquire && <>b.acquire)
(b) It is always the case that if a process acquires the resource, it will eventually release it.
assert USE_BEFORE_RELEASE = [](a.acquire -> <>a.release) && (b.acquire -> <>b.release))
(c) It is always the case that if a process acquires the resource, it will not release it until the other process acquires it.
fluent A_HOLDS = <a.acquire, a.release>
fluent B_HOLDS = <b.acquire, b.release>
assert HOLDS_UNTIL = [](A_HOLDS U B_HOLDS) && (B_HOLDS U A_HOLDS))

Complex systems
Property of complex systems
Emergence
- The system has properties that the individual parts do not
- These properties cannot be easily inferred or predicted
- Different properties can emerge from the same parts, depending upon context or arrangement

Self-organisation
- Order increases without external intervention
- Typically as a result of interactions between parts

Decentralisation
There is not a single controller or ‘leader’
distribution: each part carries a subset of global information
bounded knowledge: no part has a full view of the whole
parallelism: parts can act simultaneously
Feedback
Positive feedback amplifies fluctuations in system state. e.g. in exponential model of red foxes, more parent red foxes results in more baby foxes, result in more parents
Negative feedback dampens fluctuations in system state. e.g. in exponential model of red foxes, less parent red foxes results in less baby foxes, result in less parents

Model
A simplified description, especially a mathematical one, of a system or process, to assist calculations and predictions.
A model has a **state**, which describes all relevant aspects of the system at a particular point in time.
A model has **update rules**, which describe how the state of a model changes over time

Exponential model

$x_t = x_0 r^t$ where x_0 is initial state, *r* is a parameter that governs how steeply the curve rises

Logistic model

$x_{t+1} = rx_t(1-x_t)$

A system is chaotic if it displays all of the following four properties:

- The dynamical update rule is *deterministic*
- The system behaviour is *aperiodic*
- The system behaviour is *bounded*
- The system displays sensitivity to initial conditions

The Lokta-Volterra model — a chaotic system
Idea: if there is more food, then more birth.

Prey (Rabbits): $dR/dt = \alpha R - \beta RF$
Predators (fox): $dF/dt = \delta RF - \gamma F$
R: rabbit population
F: fox population
α growth rate of the rabbit population
β rate at which foxes predate upon (eat) rabbits
δ growth rate of the fox population
γ decay rate of the fox population due to death and migration

SIR model (Susceptible, Infectious, Recovered): A model of infectious disease transmission
Susceptible: can be infected
Infectious: can infect others
Recovered: cannot be infected nor infect others
State: which people are currently healthy/sick
Update rules: how healthy people become sick

Stochastic models

Stochastic models incorporate the effects of chance.
Stochastic processes are non-deterministic in that a state does not fully determine the next state.
As our population size increases, the influence of chance diminishes.

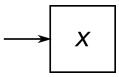
Deterministic models

Generate *unique* outcomes based on a given set of parameters and initial condition.
For each state of a deterministic model, there is only *one* possible future state.

Spatial model

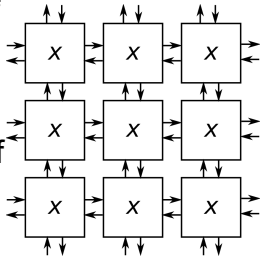
Each person/agent have a location in a square lattice(grid)

Cellular Automata



Automaton: a theoretical machine that updates its internal state based on external inputs and its own previous state

Cellular automaton: an array of automata (cells), where the inputs to one cell are the current states of nearby cells

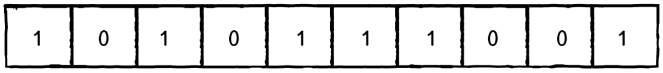


- time is discrete (like the logistic map)
- space is discrete: typically a 1D or 2D grid (3D also possible)
- system state is discrete: each component of a system is in one of a finite set of states (eg, ON or OFF); therefore the entire system has a finite, countable number of states
- update rules are defined in terms of local interactions between components

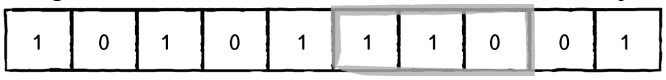
1D Cellular Automata

Space: a one-dimensional grid of cells

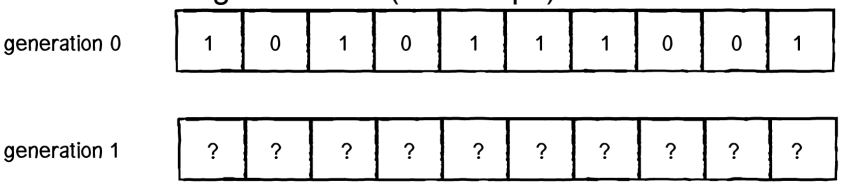
State: each cell takes one of two states



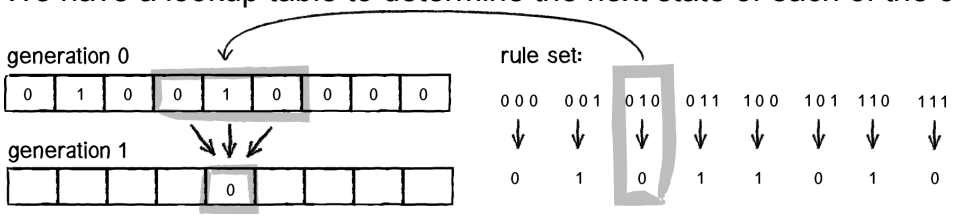
Neighbourhood: the cell itself and its two adjacent. neighbours (K = 3)



Time: discrete generations (time steps)

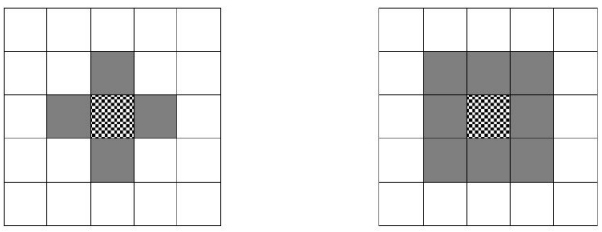


Update: use the state of the neighbourhood in the previous generation
We have a lookup table to determine the next state of each of the cells



2D Cellular automata

Similar to 1D, but with different neighborhood.
the Von Neumann neighbourhood (left): {(0, 0), (±1, 0), (0, ±1)}
the Moore neighbourhood (right): {−1, 0, 1} × {−1, 0, 1}



Conway's Game of Life rules

- each cell can be *alive* (black) or *dead* (white)
- if an alive cell has *fewer than 2* living neighbours, it dies—loneliness
- if an alive cell has *more than 3* living neighbours, it dies—overcrowding
- if an alive cell has *either 2 or 3* living neighbours, it stays alive—survival
- if a dead cell has *exactly 3* living neighbours, it comes alive—reproduction

Implement model (Assumptions/Hypothesis of model):

- Space
  - discrete vs continuous space;
  - single vs multiple occupancy of cells;
  - proximate vs long-range interactions
- Time
  - Will every component of the system be updated simultaneously? (synchronous updating) will they be updated one-at-a-time? (*asynchronous updating*)
  - in what order will the components be updated?
    - discrete: check what happens at each time step
    - continuous: what event happens next and when does it happen?
- Information
  - the scope of variables: global, cell (patch), agent (turtle)
  - the (spatial) range of sensing: what happens as this is increased? (ie, what is the *neighbourhood*)
- State updating
  - will the future state of a component be uniquely specified by its current inputs, environment, etc? (*deterministic updating*)?
  - or will their be some randomness involved? (*stochastic updating*)

A **CA version of the Lotka-Volterra model**

- Rather than modelling the size of each population (the *macro-variable*) directly, we will represent each individual explicitly, and then *observe* (measure) the size of the population.
- Space: Rather than assuming our populations are "well mixed", we will represent the location of each individual on a 2D regular lattice (grid).
- Space: We will make a new assumption that only one animal can occupy a grid space at any point in time.
- Time: Rather than treating time continuously, we will use discrete time steps.
- Information: We will assume that each individual animal is only aware of its immediate neighbours.

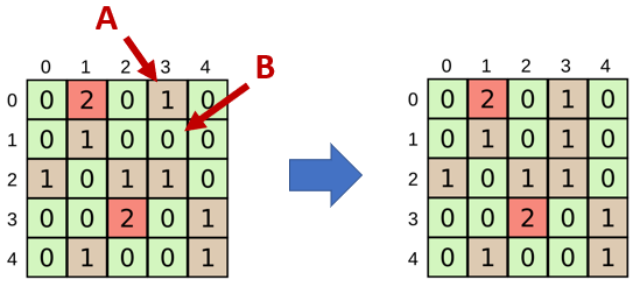
Each cell in our 2D lattice will take one of three values:

$$x_{i,j} = \begin{cases} 0 & \text{if site (i, j) is empty} \\ 1 & \text{if site (i, j) contains a rabbit (prey)} \\ 2 & \text{if site (i, j) contains a fox (predator)} \end{cases}$$

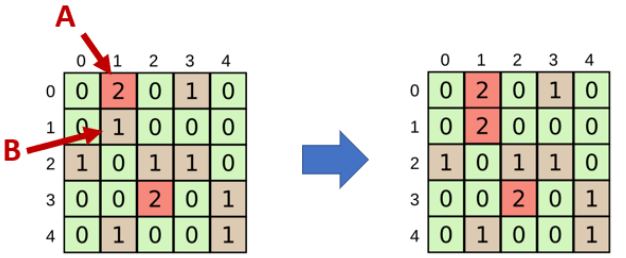
	0	1	2	3	4
0	0	2	0	1	0
1	0	1	0	0	0
2	1	0	1	1	0
3	0	0	2	0	1
4	0	1	0	0	1

Update rules

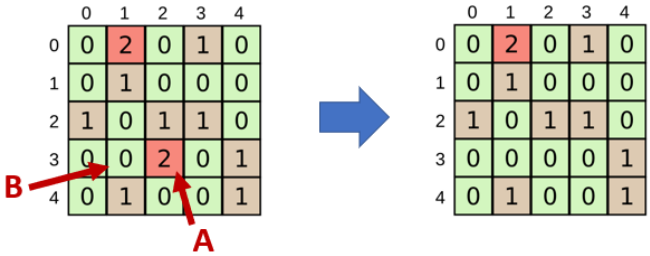
1. Pick a cell (*A*) from our lattice at random
2. Choose one of *A*'s neighbours (*B*) at random
3. Update as follows:
   - If *A* contains a rabbit and *B* is empty, then, with probability α, the rabbit reproduces, *B* now contains a new rabbit. —prey reproduction



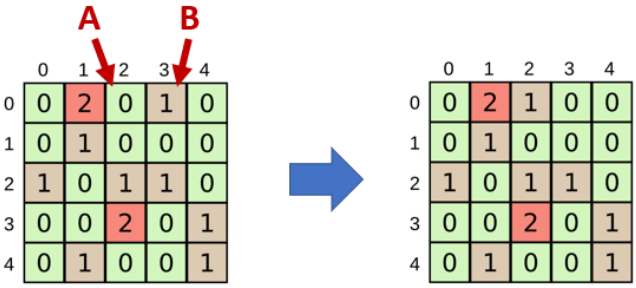
   - if *A* contains a fox and *B* contains a rabbit, or if *A* contains a rabbit and *B* contains a fox, then with probability β the rabbit is eaten; and, if this happens, with probability δ, the cell formerly containing a rabbit now contains a new fox—prey death and predator reproduction



   - if *A* contains a fox and *B* is empty, then, with probability γ, the fox dies—predator death



   - if *A* is empty and *B* contains a fox or a rabbit, then the adjacent animal moves from *B* to *A*—animal movement



Asynchronous CA

The basic CA updates *all* cells synchronously (at the same time) at *every* time step. We could also update cells at different times.

Probabilistic CA

The basic CA update rules are deterministic. We could also use update rules that are probabilistic/ stochastic.

Non-homogeneous CA

The basic CA applies the same update rule to every cell. We could also use context-sensitive rules.

Network-structured CA

The basic CA defines neighbourhood in terms of grid adjacency. We could also use a more complex *network* topology of neighbours

CA Advantages

- they are (relatively) simple and easy to implement
- they can represent interactions and behaviours that are difficult to model using ODEs
- they reflect the intrinsic individuality of system components

CA Disadvantages

- they are relatively constrained (topology, interactions, individual behaviour)
- the global behaviour may be difficult to interpret

Agent based models

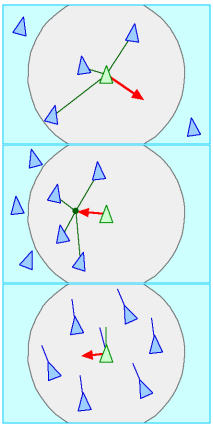
- Agents, Environment, Interactions

'Boid' (bird) model of flocking behavior

Agent behavior: forward motion; turn left/right; (perhaps) acceleration and deceleration

Rules for behavior:

Boids will steer to avoid crowding nearby boids—*collision avoidance*



Boids will steer towards the average location of nearby boids—*safety in numbers*

Boids will steer toward the average bearing of nearby boids

Essential characteristics:

Self-contained

An agent is a modular component of a system, is has a boundary and can be clearly distinguished from and recognised by other agents.

Flocking model: the boids are clearly distinguishable and 'recognised' by other boids.

autonomous

An agent can function independently in its environment, and in its interactions with other agents (within the scope of the defined model).

Flocking model: the behaviour of each boid is entirely defined by the information it obtains about its local environment

dynamic state

An agent has attributes, or variables, that can change over time. An agents current state is what determines its future actions and behaviour.

Flocking model: a boid's state consists of its current heading and speed. This state determines the motion of the boid, and is modified on the basis of information it recieves about its local environment.

social

An agent has dynamic interactions with other agents that influence their behaviour.

Flocking model: boids 'interact' by perceiving and reacting to the location and behaviour of other boids.

adaptive

An agent may have the ability to learn and adapt its behaviour on the basis of its past experiences.

For example, imagine if prey boids in predator-prey model observed that predators more often attacked boids on their right than their left (becuase of the location of their sensors/eyes), they might be able to 'learn' to move to the left of a predator and hence escape detection.

goal-directed

An agent may have goals that it is attempting to achieve via its behaviours.

For example, imagine that, in addition to avoiding predators, prey boids also have the goal of collecting materials to build a nest... This would be another factor influencing their behaviour.

heterogeneous

A system may be comprised of agents of different types: these differences may be by design (eg, predators and prey), or a result of an agents' past history (eg, the 'energy level' of predators, based upon whether it has eaten recently)

Environment
- Agents monitor and react to their environment—the physical or virtual space in which the agent 'functions'.
- The one- and two- dimensional grids of CA were very simple environments.
- Environments may be static (unchanging over time), or they may change as a result of agent behaviour, or they may be independently dynamic.
- Real environments are typically dynamic (and often stochastic). Therefore we may not be able to foresee all possible 'states' of the environment and how agents will respond to them.

Interactions
- A defining characteristic of complex systems is local interactions between agents, as defined by the agent neighbourhood.
- Depending on the structure of the system, the composition of an agent's neighbourhood may be dynamic (ie, change over time as it moves through its environment)
- Neighbourhoods, and hence patterns of interaction, are not always defined in spatial terms—they may also be networks with a particular topology

NetLogo 'Ants' model
Ant rules
1. If an ant is not carrying food:
- move randomly around its environment
- if it encounters food pheromones, move in the direction of the strongest signal
2. If an ant encounters food:
- pick it up
3. If an ant is carrying food:
- follow the nest pheromone gradient back toward the nest
- deposit food pheromones into the environment while moving

Environment
- There is a nest pheromone gradient diffusing out from the nest
- food pheromones evaporate over time

It has strong ability to locate efficient routes

Agent decision making
Agents are often designed to make rational decisions: they will act to maximise the expected value of their behaviour given the information they have received thus far.
- probabilistic: representing decisions using distributions
- rule-based: modelling the decision making process
- adaptive: agents may learn via reinforcement or evolution

Reflexive agents
Based on data from sensors and if … then … rules to decide actions.
Reflexive agents with internal state
Has an internal state changes by the data from sensors, it also remembers the previous actions and and the impact to environment of previous actions.
Goal-driven agents
How much I can approach the goal
Utilitarian agents
Calculate utility of all potential actions and apply the highest utility action
Learning agents
Keep learning from feedback based on data from sensors, keep adding knowledge and changes performance.

Trabelling Salesman Problem
Given a list of cities, and distances between each pair of cities, what is the shortest possible route that visits all cities exactly once and returns to the original city?

Ant Colony Optimisation algorithm

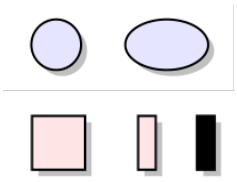
1. Distribute N ants at random among the nodes (destinations)
2. Each ant randomly chooses a new node to travel to, based on:
   pheromone concentration on the edge to that node (weighted by α)
   distance to that node (weighted by β)
   it must not have visited that node previously
3. Repeat step 2 until each ant has visited all nodes
4. Calculate the total route length for each ant
5. The ant that achieved the shortest route deposits pheromone along the edges in their route (with concentration inversely proportional to route length)
6. Pheromones evaporate at a rate ρ

The algorithm is widely applied to problems in vehicle routing and scheduling, also image processing, circuit design and others

Petri net

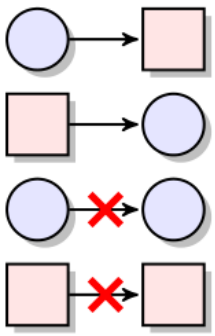
Component of a net structure
A Petri net is a directed graph with two types of nodes: places and transitions.

Places
- A place p is a passive system component
- Graphically represented as a circle or ellipse
- A place has discrete states and can store things



Transitions
- A transition t is an active system component
- Graphically represented as a square or rectangle
- A transition consumes, produces, or transfers things

Arcs
- An arc is not a system component but a logical relation between system components
- Graphically represented as an arrow
- An arc relates a place to a transition or a transition to a place; hence, a Petri net is a directed bipartite graph



Net structure
Sets of places, transitions, and arcs of a Petri net are customary denoted by P, T, and F, respectively. As arcs model relation between places and transitions, is defined as a relation $F \subseteq (P \times T) \cup (T \times P)$ .Then, N = (P, T, F) is a net structure. Places and transitions are the elements of N, and F is the flow relation of N.

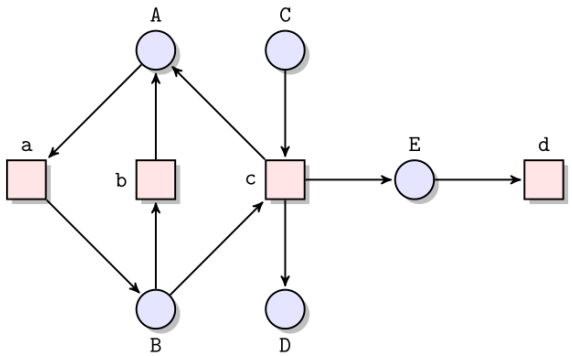


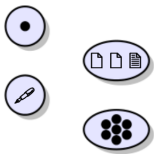
Figure: A net structure (P, T, F) with P = {A,B,C,D,E}, T = {a,b,c,d}, and F = {(A,a), (a,B), (B,b), (b,A), (B,c), (c,A), (C,c), (c,D), (c,E), (E,d)}

Markings

A marking encodes a state of a system and is an arrangement of tokens across places of a net structure.

Tokens

- A token can represent a thing from the real-world that can be consumed, produced, or transferred by a transition
- A token can indicate a condition that is satisfied for an occurrence of a transition
- Tokens are placed inside circles and ellipses that depict places and are graphically represented as black dots
- A symbolic token can be used instead of a black dot to refer to a concrete thing



Elementary net systems only use black dot tokens (i.e., tokens abstract from the nature of things they represent).

Elementary net systems

An (elementary) net system S is a pair (N,M_0), where $N = (P, T, F)$ is a **net structure** with finite disjoint sets of places and transitions (i.e., $P \cap T = \varnothing$) and $M0 : P \rightarrow N^+$ is an **initial marking**.

N^+ is the set of all natural numbers including zero (i.e., non-negative integers)

Each place $p \in P$ holds $M_0(p)$ tokens

Net system S merely describes current state of control flow and availability of resources

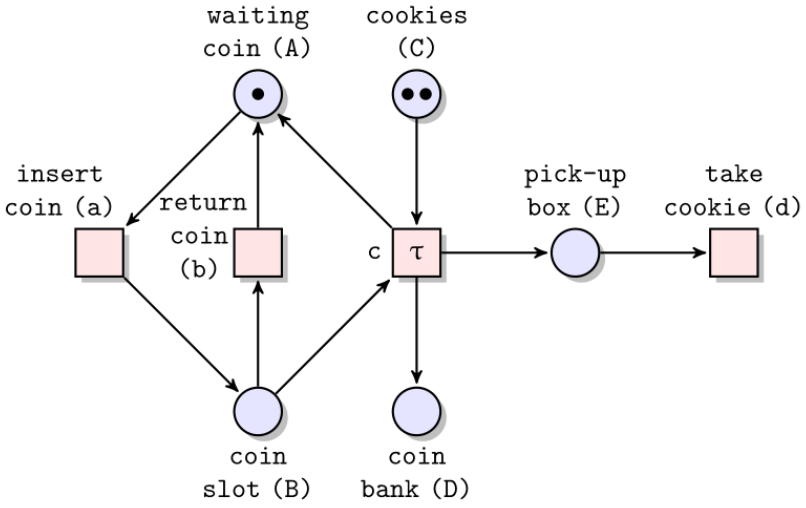


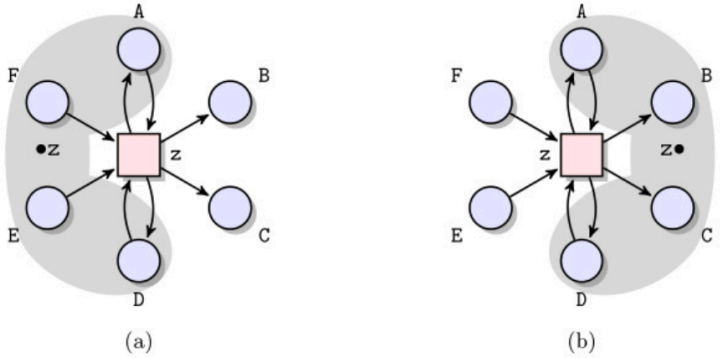
Figure: An abstract model of a cookie vending machine in the initial marking $M_0 = \{(A,1), (B,0), (C,2), (D,0), (E,0)$ or $M_0 = ACC$.

Pre-set and post-set of an element

Given a net structure (P, T, F), the pre-set $\cdot\mathbf{x}$ and the post-set $\mathbf{x}\cdot$ of an element $x \in P \cup T$ are defined as follows:

$\cdot\mathbf{x} =_{def} \{y \in P \cup T \mid (y, x) \in F\}$ I.e. all places that points to the transition.

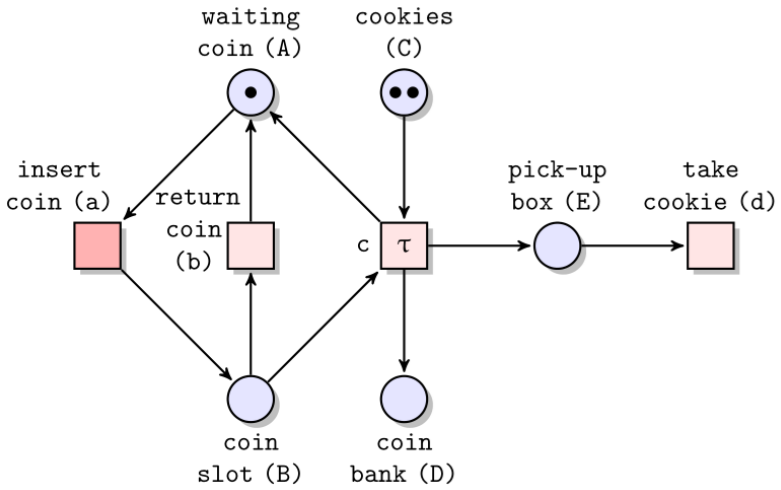
$\mathbf{x}\cdot =_{def} \{y \in P \cup T \mid (x, y) \in F\}$. I.e. all places that transition points to.



(a)Pre-set $\cdot\mathbf{z}$ = A,D,E,F and
(b)post-set $\mathbf{z}\cdot$ = A,B,C,D of transition z

The transition enablement rule

Given a net system (N, M), where N = (P, T, F), marking M enables transition t ∈ T if every place in the pre-set of t contains at least one token (i.e., ∀p ∈ ·t: M(p) ≥ 1).

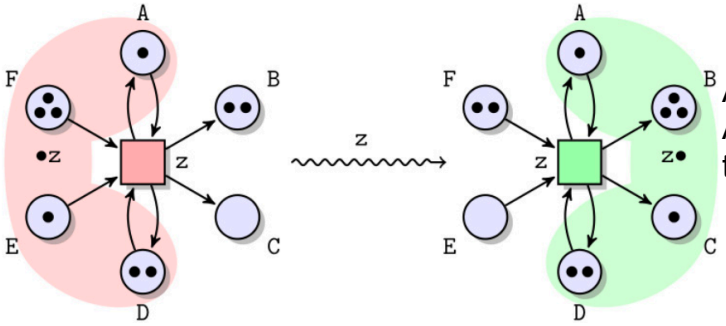


A net system and its only enabled transition a.

Step rule

An enabled in marking transition t can occur. An **occurrence** of t results in a step M -t-> M', where marking M′ is the next state of the system obtained from M by first **destroying one token in every place in the pre-set of t and then creating one fresh token in every place in the post-set of t**. The number of tokens at place p ∈ P in marking M′ is defined as follows:

$$M'(p) = \begin{cases} M(p) - 1 & p \in {\cdot}t \wedge p \notin t{\cdot} \\ M(p) + 1 & p \in t{\cdot} \wedge p \notin {\cdot}t \\ M(p) & \text{otherwise} \end{cases}$$

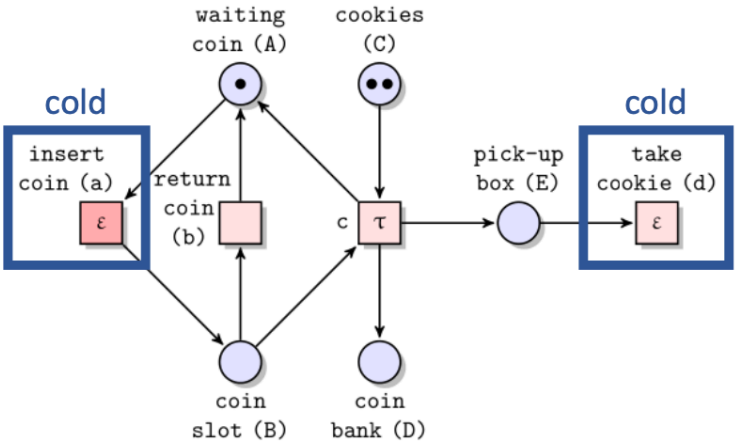


An example step ABBDDEFFF -t-> ABBBCDDFF induced by occurrence of transition z.

Hot and cold transitions

Transitions that will happen eventually are hot transitions.

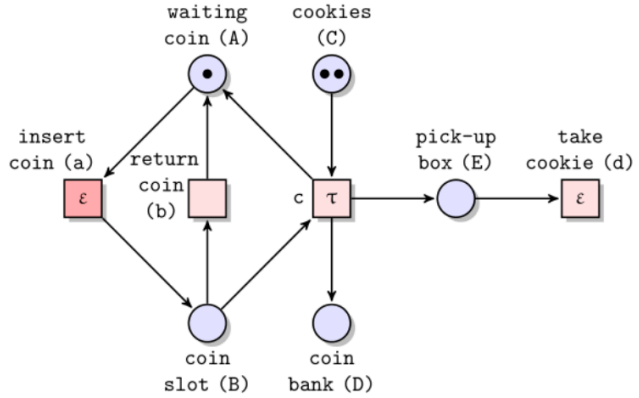
Transitions triggered by an external system and not guaranteed whether they will ever occur are cold transitions.



Reachable markings

For a net system S = (N, M_0), a marking M of N is a reachable marking of S if M = M_0 or there exists a sequence of steps $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} M_2 \xrightarrow{t_3} \cdots \xrightarrow{t_{n-1}} M_{n-1} \xrightarrow{t_n} M_n$,

where $n \in N^+$, $M = M_n$, and $t_i \in T$, $i \in [1..n]$. A **final marking** is a marking in which no hot transitions are enabled (i.e., the system can remain in a final marking forever).



Marking	Reachable	Final
BCC	✓	✗
ACC	✓	✓
ADD	✓	✓
ACDDE	✗	✓
ABCDE	✗	✗

Sequential runs (occurrence sequences)

A sequential run of a net system S = (N, M_0) is a possibly infinite sequence of steps that starts at the initial marking M_0 of S. $M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} M_2 \xrightarrow{t_3} \cdots$

A finite sequential run is complete if it leads to a final marking of the net system
An infinite sequential run is complete if no additional step can be inserted into it

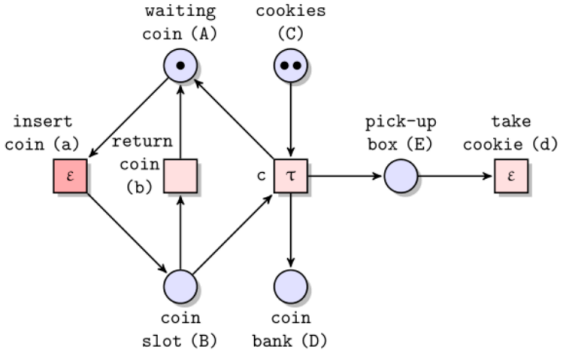


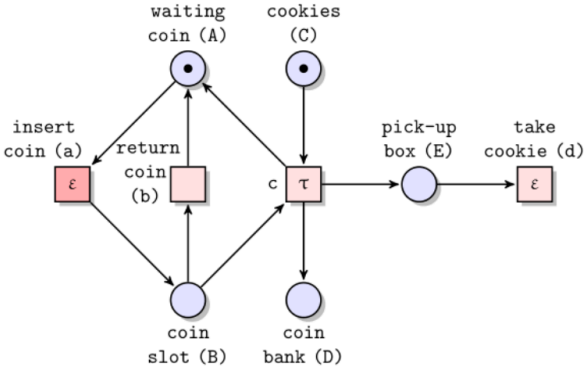
Figure: A net system

Sequential run	Finite	Complete
$ACC \xrightarrow{a} BCC$	✓	✗
$ACC \xrightarrow{a} BCC \xrightarrow{b} ACC$	✓	✓
$ACC \xrightarrow{a} BCC \xrightarrow{c} ACDE$	✓	✓
$ACC \xrightarrow{a} BCC \xrightarrow{c} ACDE \xrightarrow{a} BCDE \xrightarrow{c} ADDEE \xrightarrow{d} ADDE \xrightarrow{d} ADD \xrightarrow{a} BDD \xrightarrow{b} ADD \xrightarrow{a} BDD \xrightarrow{b} \cdots$	✗	✓
$ACC \xrightarrow{a} BCC \xrightarrow{b} ACC \xrightarrow{a} \cdots$	✗	✗

Marking graphs (reachability graphs)

The reachable markings and steps of a net system define the marking graph of S.

- Its nodes describe the reachable markings of S
- Its edges describe the steps between the reachable markings of S
- The initial marking is highlighted with an arrow leading to it
- Each final marking is highlighted with a double line border



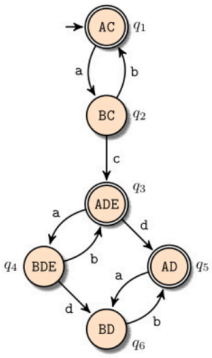


Figure: The marking graph of S.

1. The empty sequence
2. $AC \xrightarrow{a} BC \xrightarrow{b} AC$
3. $AC \xrightarrow{a} BC \xrightarrow{b} AC \xrightarrow{a} BC \xrightarrow{b} AC$
   …
n. $AC \xrightarrow{a} BC \xrightarrow{b} AC \xrightarrow{c} ADE$
n+1. $AC \xrightarrow{a} BC \xrightarrow{b} AC \xrightarrow{a} BC \xrightarrow{b} AC \xrightarrow{c} ADE$
   …
m. $AC \xrightarrow{a} BC \xrightarrow{b} AC \xrightarrow{c} ADE \xrightarrow{d} AD$
m+1. $AC \xrightarrow{a} BC \xrightarrow{b} AC \xrightarrow{a} BC \xrightarrow{b} AC \xrightarrow{c} ADE \xrightarrow{d} AD$
   …
k. $AC \xrightarrow{a} BC \xrightarrow{b} AC \xrightarrow{a} BC \xrightarrow{b} AC \xrightarrow{c} ADE \xrightarrow{d} AD \xrightarrow{a} BD \xrightarrow{b} AD \xrightarrow{a} \cdots$

Interleaving semantics of net systems

The collection of all complete sequential runs of a net system defines the interleaving semantics of the system.

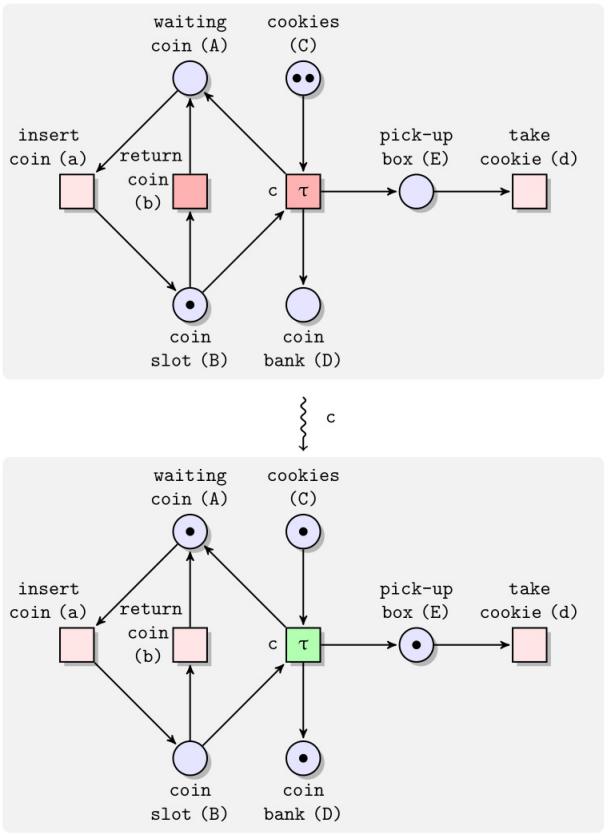
- A net system can describe an infinite collection of complete sequential runs
- Each sequential run of a net system is a directed walk in its marking graph that starts at the node that corresponds to the initial marking of the net system
- The marking graph describes all and only complete sequential runs of the net system

The state explosion problem

In general, a net system can define a very large, or even infinite, marking graph. This phenomenon is known as the state explosion problem.

Actions

A net system can be interpreted as a collection of distributed runs. A distributed run describes the behaviour of the system more accurately but requires more effort to understand. The building block of a distributed run is an action, which describes a step.



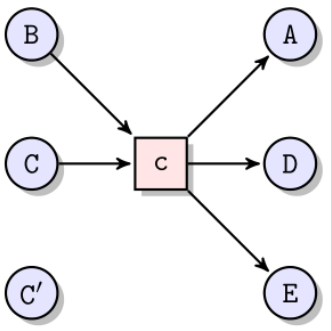
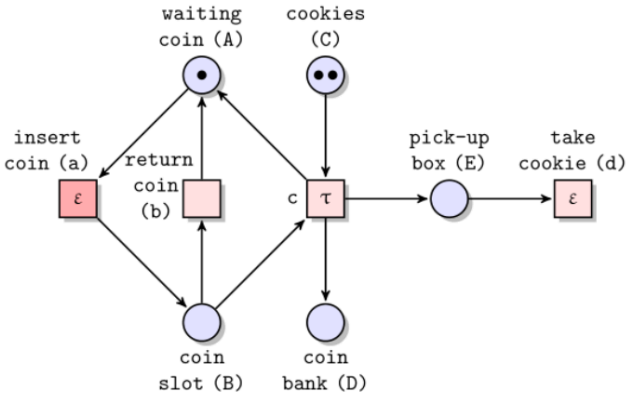


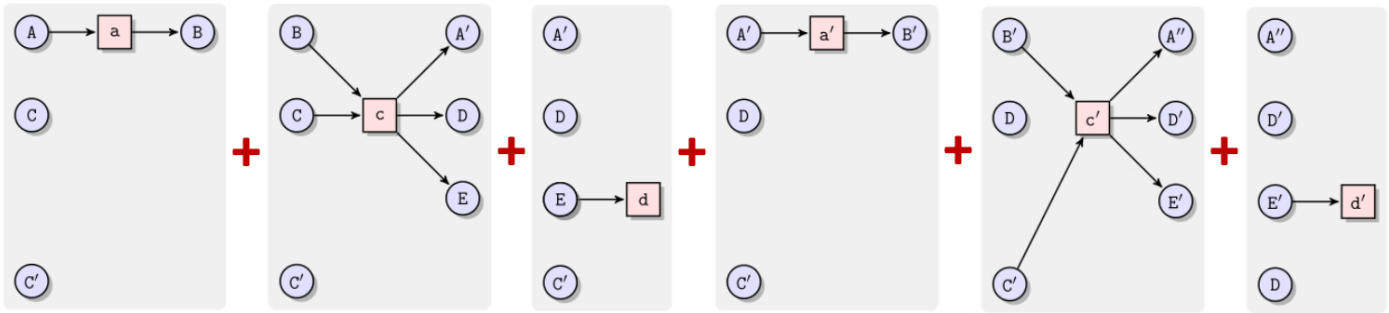
Figure: An action that describes the step on the left.

- The transition refers to the occurred transition
- The places without incoming arcs represent the marking before the transition occurrence
- The places without outgoing arcs represent the marking after the transition occurrence

Distributed runs (processes)

A distributed run of a net system is a possibly infinite acyclic net structure, also called a causal net, that starts at the initial marking and is defined by an uninterrupted sequence of actions executed by the system.





Concurrency semantics of net systems

Each transition of a distributed run represents an action. The collection of all complete distributed runs of a net system defines the concurrency semantics of the system.

- A finite distributed run is complete if its places without outgoing arcs represent a final marking of the net system
- An infinite distributed run is complete if no additional action can be inserted into it

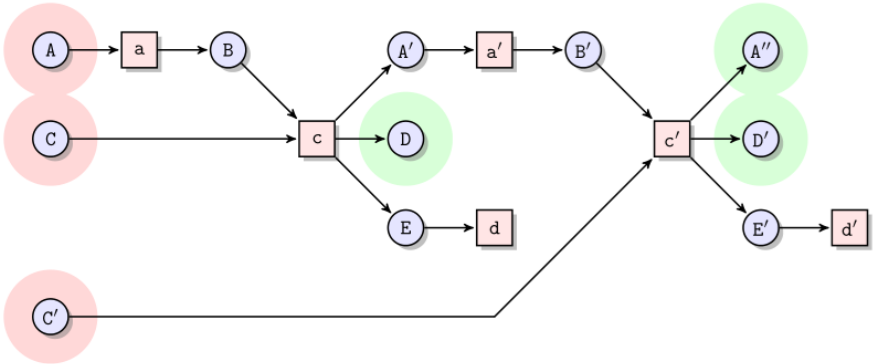
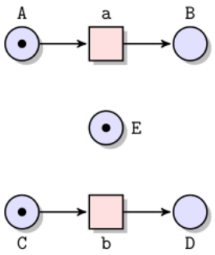


Figure: A complete distributed run from marking ACC to marking ADD.

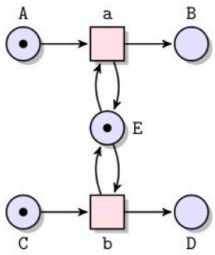
Sequential vs distributed runs

Both systems shown below have the same complete sequential runs[2], namely:

$$\mathrm{ACE} \xrightarrow{a} \mathrm{BCE} \xrightarrow{b} \mathrm{BDE} \quad \text{and} \quad \mathrm{ACE} \xrightarrow{b} \mathrm{ADE} \xrightarrow{a} \mathrm{BDE}.$$



System S_1: Independent a and b.



System S_2: Arbitrary order of a and b.

However, complete distributed runs of these two net systems are distinct.

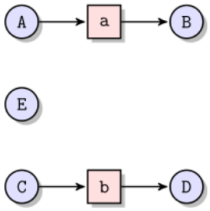


Figure: The only complete distributed run of S_1.

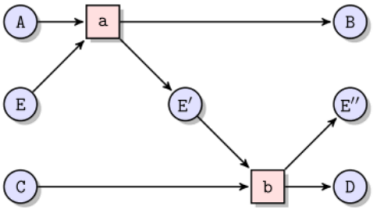


Figure: One of the two complete distributed runs of S_2.

13 /

Causality and concurrency relations

Distributed runs encode causality and concurrency relations between actions. A chain of arcs from element x to element y in a causal net specifies that x causally precedes y, denoted by x ↣ y.
If two elements x and y of a distributed run are not causal (i.e., neither x ↣ y nor x ↢ y holds), then they are concurrent, denoted by x || y.
The causality, inverse causality, and concurrency relations partition the Cartesian product of elements of a distributed run.

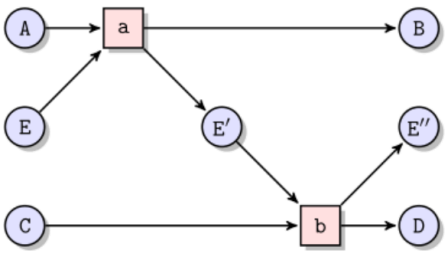


Figure: A distributed run.

	a	b	A	B	C	D	E	E′	E″
a	∥	↣	↢	↣	∥	↣	↢	↣	↣
b	↢	∥	↢	∥	↢	↣	↢	↢	↣
A	↣	↣	∥	↣	∥	↣	∥	↣	↣
B	↢	∥	↢	∥	∥	∥	↢	∥	∥
C	∥	↣	∥	∥	∥	↣	∥	∥	↣
D	↢	↢	↢	∥	↢	∥	↢	↢	∥
E	↣	↣	∥	↣	∥	↣	∥	↣	↣
E′	↢	↣	↢	∥	∥	↣	↢	∥	↣
E″	↢	↢	↢	∥	↢	∥	↢	↢	∥

Decision problems

A decision problem is a yes-or-no question over a possibly infinite set of inputs.
A decision problem is often defined using two sets: the set of possible inputs and the set of inputs for which the answer to the question is yes
A decision problem is decidable, or **effectively solvable**, if there is an algorithm that takes an input, terminates after a finite amount of time, and correctly answers the question posed by the problem.
The problem of deciding whether a given natural number is a prime number is an example of a decidable decision problem
Given an arbitrary computer program and an input, the problem of deciding whether the program will finish running or continue to run forever is known as the halting problem and is an example of an undecidable decision problem

Boundedness
A net system is bounded if the set of all reachable markings of is finite; otherwise, is unbounded.
i.e. if some transitions can keep adding tokens in the system then it is unbounded

k-boundedness and safeness
A net system S = (N, M_0), N = (P,T,F), is k-bounded, k ∈ N, if for every place p ∈ P and for every reachable marking M of S it holds that the number of tokens at p in M is less than or equal to k, that is, M(p) ≤ k.
- A k-bounded system is bounded
- For a bounded system there exists k ∈ N for which the system is k-bounded
- A k-bounded system is also m-bounded, where m ∈ N and m > k
- A 1-bounded system is also called safe

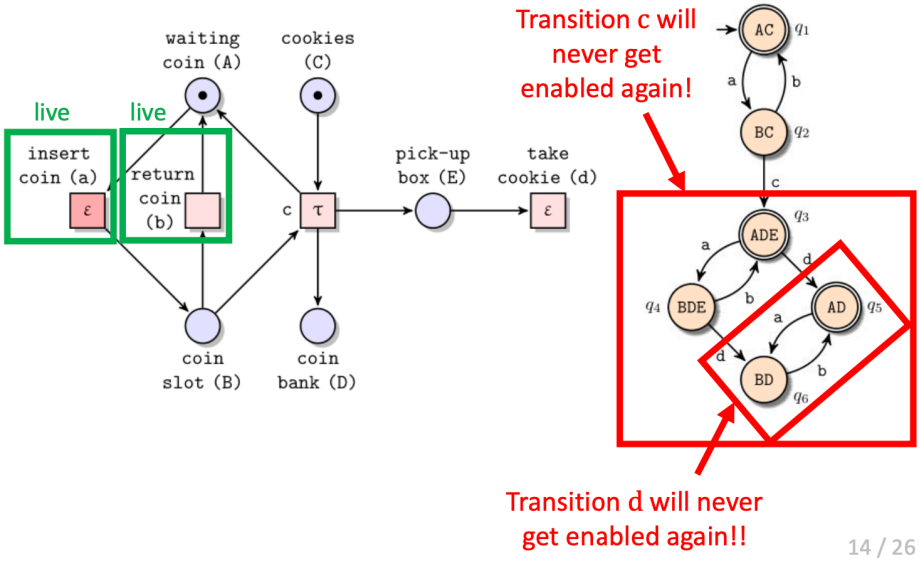
The previous vending machine is a 2-bounded net system

Liveness
For a net system S = (N, M_0), N = (P,T,F), a transition t ∈ T is live if from every reachable marking M of S it is possible to reach a marking that enables t, that is, for every marking M = M_0 or M = M_n such that:

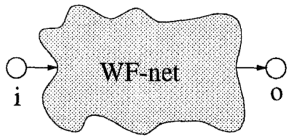
$M_0 \xrightarrow{t_1} M_1 \xrightarrow{t_2} \cdots \xrightarrow{t_{n-1}} M_{n-1} \xrightarrow{t_n} M_n$, where n ∈ N^+, and t_i ∈ T, i ∈ [1..n], there exists a sequence of steps

$M_n \xrightarrow{t_{n+1}} M_{n+1} \xrightarrow{t_{n+2}} \cdots \xrightarrow{t_{k-1}} M_{k-1} \xrightarrow{t_k} M_k$, where k ∈ N^+, k > n, t_k = t, and t_j ∈ T, j ∈ [n..k]

A live transition can become enabled again and again (infinitely many times)
A net system S is live if every its transition is live
If a system is not live, it is possible to reach a marking after which some transitions of the system will never get enabled again

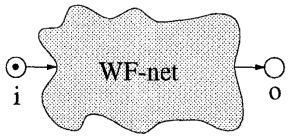


Workflow nets
A workflow net, or a **WF-net**, is a net structure N = (P, T, F) with a special source place i ∈ P that has the empty pre-set (·i = ∅), a special sink place o ∈ P that has the empty post-set (o· = ∅), and every element of N is on the vertex sequence of a directed walk from i to o in the graph (P ∪ T, F).
- The source place is used to denote the beginning of a workflow
- The sink place is used to denote the termination of a workflow
- Every element of a workflow net should have a chance to be executed



Workflow systems

A workflow system is a net system S = (N, M_0), where N = (P,T,F) is a workflow net and M_0 "puts" one token at the source place i ∈ P of N and no tokens elsewhere;
that is, M_0(i) =1and M_0(p) =0, p ∈ P, p ≠ i.



- The token at place i denotes the beginning of a single workflow, or case
- Cases are processed independently
- All tokens of a reachable marking describe a reachable state of the workflow system

Soundness: An intuition

the soundness property of a workflow system, which relates to the dynamics of the system by imposing two constraints:

- Every case eventually terminates; when a case terminates there is a token at the sink place and all the other places are empty
- There are no dead transitions; for every transition t, there is a case that executes t

Option to complete

An option to complete property holds for a workflow system S = (N, M_0)
if and only if from every reachable marking M of S it is possible to reach a marking that puts a token at the sink place of N.

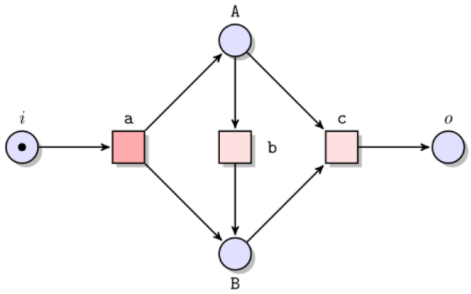


Figure: No option to complete.

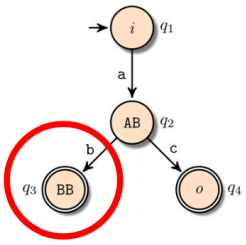


Figure: Marking graph.

Proper completion

A proper completion property holds for a workflow system S = (N, M0), N = (P, T, F), if and only if the marking with one token at the sink place o ∈ P and no tokens elsewhere is the only reachable marking of S with at least one token at place o.

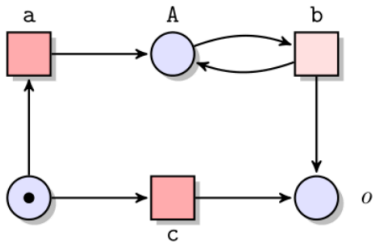


Figure: No proper completion.

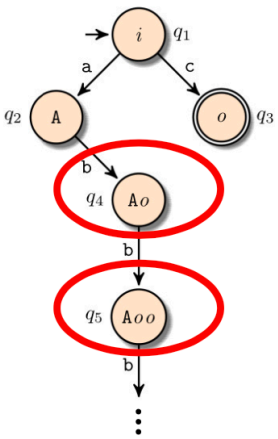


Figure: Marking graph.

No dead transitions

A workflow system S = (N, M_0), N = (P,T,F), has no dead transitions if and only if for every transition t ∈ T there exists a reachable marking of S that enables t.

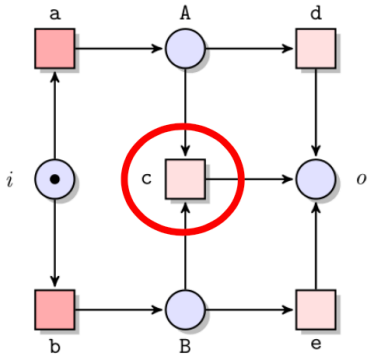


Figure: Dead transition.

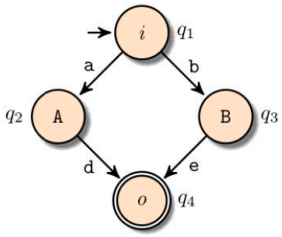


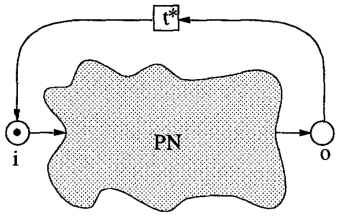
Figure: Marking graph.

Soundness

A workflow system is sound if and only if option to complete, proper completion, and no dead transitions properties hold.

Short-circuiting

The short-circuit version of a workflow system (N, M_0),
$i \in P$ is the source place, and $o \in P$ is the sink place, is the net system $(\bar{N}, M_0)$, where $\bar{N} = (\bar{P}, \bar{T}, \bar{F})$, such that:



- $\bar{P} = P$
- $\bar{T} = T \cup \{t^*\}, t^* \notin T$
- $\bar{F} = F \cup \{(o, t^*), (t^*, i)\}$

A workflow system S is sound if and only if the short-circuit version of is live and bounded.