Discrete event and hybrid systems

Notes from the a.y. 2021/2022 course held by Prof. Tiziano Villa

Author: Lorenzo Busellato

UNIVERSITÀ
di VERONA
Dipartimento
di INFORMATICA

Contents

1 Signals and systems

1.1 Signals

A signal e may be either **present**, condition denoted with e, or **absent**, condition denoted with $\varnothing$.

Definition 1.1 (Pure signal)
A signal:

$$e \coloneqq \mathbb{R} \to \{e, \varnothing\}$$
$$t \mapsto e(t)$$

*is **pure** if and only if at every time $t \in \mathbb{R}$ it is either absent ($\varnothing$) or present (e).*

Intuitively a discrete signal is a signal which is absent most of the time, and the times where it is present can be ordered.

Definition 1.2 (Discrete signal)
A signal:

$$e \coloneqq \mathbb{R} \to \{e, \varnothing\}$$
$$t \mapsto e(t)$$

*is **discrete** if there exists an injective function called **ordering function** that is order preserving:*

$$f \coloneqq T \coloneqq \{t \in \mathbb{R} \mid e(t) \neq \varnothing\} \to \mathbb{N}$$

Example 1.1
Consider the pure signal:

$$x : T_x \coloneqq \mathbb{R} \to \{x, \varnothing\}$$
$$t \mapsto \begin{cases} x & t \in \mathbb{N}_0 \\ \varnothing & t \notin \mathbb{N}_0 \end{cases}$$

The signal is discrete since the set of times in which it is present is the natural numbers set, so we can pick the identity function $f(t) = t$ as the ordering function, which is by definition injective and order preserving.

Example 1.2
Consider the pure signal:

$$y : T \coloneqq \mathbb{R} \to \{y, \varnothing\}$$
$$t \mapsto \begin{cases} y & t = 1 - \frac{1}{n} \\ \varnothing & t \neq 1 - \frac{1}{n} \end{cases}$$

The signal is discrete, in fact we can define an order preserving injective function as:

$$f : T_y \coloneqq \{n \in \mathbb{N}, t = 1 - \frac{1}{n} \in \mathbb{R} \mid y(t) \neq \varnothing\} \mapsto \mathbb{N}$$
$$t \mapsto \frac{1}{-(t-1)}$$

Example 1.3
Consider the signal w obtained from the composition of the signals x and y of the previous examples:

$$w : T_w \coloneqq \mathbb{R} \to \{w, \varnothing\}$$
$$t \mapsto \begin{cases} w & x(t) = x \text{ or } y(t) = y \\ \varnothing & x(t) = \varnothing \text{ and } y(t) = \varnothing \end{cases}$$

Proving that w is discrete requires the definition of an order preserving injective function $f : T_w \to \mathbb{N}$. We can observe that:

$$1 \in T, 1 - \frac{1}{n} \in T \text{ and } 1 > 1 - \frac{1}{n} \quad \forall n \in \mathbb{N}, n > 0$$

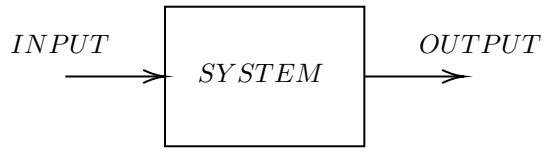
If f is order preserving, it must be true that:

$$f(1) > f\left(1 - \frac{1}{n}\right)$$

but $f\left(1 - \frac{1}{n}\right)$ has no majorant in $\mathbb{N}$ since $1 - \frac{1}{n}$ has no minorant smaller than 1 for $n \in \mathbb{N}, n > 0$, therefore such a function does not exist and the composite signal w is not discrete. The property of discreteness is therefore not generally preserved by signal composition.

1.2 Systems

A **physical system** is a portion of space physically or logically distinct from the surrounding universe with which the system interacts. The main task in systems theory is the construction of a mathematical model (**dynamic system** or **system** for short) which models the phenomena interested by the system in the most precise way while maintaining a reasonably low computational difficulty.



The two main models for a system are the **input/output models** and the **state models**. In both cases the system can be viewed as a "black box" that, given an input signal, produces an output signal.

Definition 1.3 (Transducive system)
***Transducive**, or **combinatorial**, systems are maps between two value domains:*

$$f : \mathbb{D}_T \to \mathbb{D}_V$$
$$t \in \mathbb{D}_T \mapsto f(t)$$

*where $\mathbb{D}_T$ is the time domain, which can be **continuous** with $\mathbb{D}_T \subset \mathbb{R}$ or **discrete** with $\mathbb{D}_T \subset \mathbb{Z}$, and $\mathbb{D}_V$ is the value domain.*

Definition 1.4 (Reactive system)
***Reactive**, or **sequential**, systems are maps between two signal domains:*

$$F : [\mathbb{D}_T \mapsto \mathbb{D}_V] \to [\mathbb{D}_T \mapsto \mathbb{D}_V]$$
$$(t \in \mathbb{D}_T, v \in \mathbb{D}_V) \mapsto F(t, v)$$

*where $\mathbb{D}_T$ is the time domain, which can be **continuous** with $\mathbb{D}_T \subset \mathbb{R}$ or **discrete** with $\mathbb{D}_T \subset \mathbb{Z}$, and $\mathbb{D}_V$ is the value domain.*

Observation 1.1
Using the properties of function chains, it can be verified that any transducive system can be implemented as a reactive system. In fact, given a transducive system f and a reactive system F, we can write:

$$\forall x \in [\mathbb{D}_T \mapsto \mathbb{D}_V], \forall y \in \mathbb{D}_T, (F(x))(y) = f(x(y))$$

in other words transducive systems are a subset of reactive systems.

1. **Memory**: a system can have memory (if it's a reactive sytem) or it can be **memory free** (for both categories). The latter case implies that a transducive system is also reactive. Specifically, a transducive system can be rewritten as a memory free reactive system:

$$\begin{aligned} f : \mathbb{D}_T &\to \mathbb{D}_V \\ t &\mapsto f(t) \\ &\downarrow \\ F : [\mathbb{D}_T \mapsto \mathbb{D}_V] &\to [\mathbb{D}_T \mapsto \mathbb{D}_V] \\ (t, v) &\mapsto F(t, v) = f(v(t)) \end{aligned}$$

   Theorem 1.1 (Memory-free system)
   A reactive system:

$$F : [\mathbb{D}_T \mapsto \mathbb{D}_V] \to [\mathbb{D}_T \mapsto \mathbb{D}_V]$$
$$(x, t) \mapsto F(x, t)$$

   *is **memory free** if and only if there exists a transducive system such that:*

$$f : [\mathbb{D}_T \mapsto \mathbb{D}_V] \to \mathbb{D}_V$$
$$x, t \mapsto f(x(t))$$

   Corollary 1.1.1
   Every composition of memory free systems is a memory free system.

Example 1.4
We can see some examples of memory free reactive systems, such as the normalize system:

$$Normalize \coloneqq [\mathbb{R}^+ \to \mathbb{R}] \to [\mathbb{R}^+ \to \mathbb{R}]$$
$$x \in [\mathbb{R}^+ \to \mathbb{R}], y \in \mathbb{R}^+ \mapsto (Normalize(x))(y) = x(y) - 50$$

The truncate system:

$$Trunc \coloneqq [\mathbb{R}^+ \to \mathbb{R}] \to [\mathbb{R}^+ \to \mathbb{R}]$$
$$x \in [\mathbb{R}^+ \to \mathbb{R}], y \in \mathbb{R}^+ \mapsto (Trunc(x))(y) = \begin{cases} 256 & x(y) > 256 \\ x(y) & -256 < x(y) < 256 \\ -256 & x(y) < -256 \end{cases}$$

The quantize system:

$$Quantize \coloneqq [\mathbb{N}_0 \to \mathbb{R}] \to [\mathbb{N}_0 \to Ints]$$
$$x \in [\mathbb{N}_0 \to \mathbb{R}], y \in \mathbb{N}_0 \mapsto (Quantize(x))(y) = \begin{cases} 256 & \lfloor x(y) \rfloor > 256 \\ \lfloor x(y) \rfloor & -256 < \lfloor x(y) \rfloor < 256 \\ -256 & \lfloor x(y) \rfloor < -256 \end{cases}$$

The negate system:

$$Negate \coloneqq [\mathbb{N}_0 \to Bools] \to [\mathbb{N}_0 \to Bools]$$
$$x \in [\mathbb{N}_0 \to Bools], y \in \mathbb{N}_0 \mapsto (Negate(x))(y) = \neg x(y)$$

The identity system:

$$Id \coloneqq [\mathbb{D}_T \mapsto \mathbb{D}_V] \to [\mathbb{D}_T \mapsto \mathbb{D}_V]$$
$$x \in [\mathbb{D}_T \mapsto \mathbb{D}_V] \mapsto Id(x) = x$$

And the constant system:

$$Const \coloneqq [\mathbb{D}_T \mapsto \mathbb{D}_V] \to [\mathbb{D}_T \mapsto \mathbb{D}_V]$$
$$x \in [\mathbb{D}_T \mapsto \mathbb{D}_V], y \in \mathbb{D}_V \mapsto (Const(x))(y) = c$$

2. **Delay**: the **delay system** is a reactive system defined as:

$$F : [\mathbb{D}_T \mapsto \mathbb{D}_V] \to [\mathbb{D}_T \mapsto \mathbb{D}_V]$$
$$x, y \mapsto \begin{cases} c & y < 1 \\ x(y-1) & y \geq 1 \end{cases}$$

The delay system is a basic sequential system that implements a memory element. To prevent undefined behaviour at the first input the delay system is initialized to some value c. In all other cases the delay element stores the current input and outputs the previous input stored.

3. **Finite memory vs infinite memory**: depending on the time domain of the delay element(s), a system can have **infinite** or **finite memory**.

Observation 1.2
*Finite memory systems are naturally implemented as **finite state automatons** (or **finite transitions automatons**), while countably infinite memory systems are naturally implemented as **infinite state automatons** (or **infinite transitions automatons**). Systems with uncountably infinite memory are not naturally implemented as state automatons.*

4. **Causality**: if we consider the predict function that maps time to a binary number:

$$F : [\mathbb{R} \mapsto bins] \to [\mathbb{R} \mapsto bins]$$
$$x, t \mapsto \begin{cases} 1 & \text{if } \exists z \in \mathbb{D}_T \mid x(z) = 1 \\ 0 & \text{if } \forall z \in \mathbb{D}_T \mid x(z) = 0 \end{cases}$$

The system is mathematically coherent but its physical implementation would make no sense since it's impossible to predict the future.

Definition 1.5 (Causality)
*A reactive system is **causal**, or **implementable**, if and only if* $\forall x, y \in [\mathbb{D}_T \mapsto \mathbb{D}_V], \forall z \in \mathbb{D}_T$*:*

$$\forall t \in \mathbb{D}_T, t \leq z \implies x(t) = y(t) \iff F(x, z) = F(y, z)$$

The definition showcases the fact that a causal system only depends on current and past signals. If F where to be like the predictive function, at some point in time z we could have $F(x, z) \neq F(y, z)$, which contradicts causality.

By composing basic systems we can get new systems that model more complex behaviors, but some problems could arise. If there are cycles in a block diagram of a system for instance, an oscillating unstable behavior can occur. Therefore we need to define the conditions under which a given system unambiguously produces an output signal given an input signal.

Definition 1.6 (Legality)
*A transducive system is **legal** if all of its components are transducive and there are no cycles.*

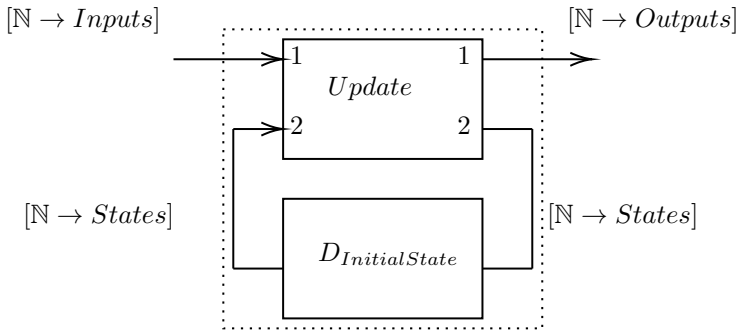
Definition 1.7 (Legality)
*A reactive system is **legal** if all of its components are memory free or delays and every cycle contains at least one delay system.*

2 Finite state machines

As we have previously seen, systems are functions that transform signals. The domains and codomains of these functions are signal spaces, which make the definition of the functions themselves harder. A great simplification of the task can be made by introducing the concept of **state**, and the idea that a system *evolves* through a sequence of **state transitions**. Models of systems characterized in this way are called **state space models**.

2.1 Deterministic finite state machines

A state machine constructs the output signal one value at a time, by observing one value at a time the input signal.



Formally, a deterministic machine is defined by a 5-tuple:

$$\{States, Inputs, Outputs, Update, InitialState\}$$

- $States$ is the space of the states. A machine is finite if its state space is finite.
- $Inputs$ is the set of all the possible input values (or signals).
- $Outputs$ is the set of all the possible output values (or signals).
- $InitialState \in States$ is the initial state of the machine.
- $Update \coloneqq States \times Inputs \to States \times Outputs$ is called **transition function**, and it associates to each couple *present state-input* a *next state-output* couple.

Observation 2.1
Any memory free system can be implemented by a deterministic finite state machine with a single state.

Observation 2.2
Any causal system can be implemented by a state machine. Any system that can be implemented by a state machine is causal.

To represent the behavior of a machine we can use a **transition table**, which is a table with all the possible present state-input/next state-output combinations for the system, or a **transition diagram**, which represents the machine in the form of a graph, in which the nodes represent the states of the machine and the edges represent the transitions between states.

Example 2.1
Consider the parity system:

$$Parity \coloneqq [\mathbb{N}_0 \mapsto Bools] \to [\mathbb{N}_0 \mapsto Bools]$$

$$x \in [\mathbb{N}_0 \mapsto Bools], y \in \mathbb{N}_0 \mapsto (Parity(x))(y) = \begin{cases} true & |trueValues(x,y)| \text{ is even} \\ false & |trueValues(x,y)| \text{ is odd} \end{cases}$$

where:

$$trueValues(x,y) = \{z \in \mathbb{N}_0 \mid z < y, x(z) = true\}$$

The system keeps track of the parity of the previous inputs, holding true if an even number of true inputs was received so far, false otherwise.

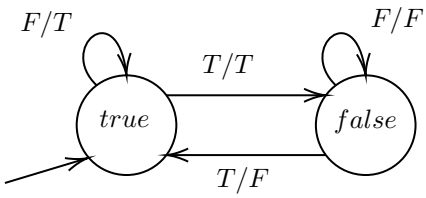
The state description for the parity system is:

$$\begin{aligned} Inputs &= Bools \\ Oututs &= Bools \\ States &= Bools \\ InitialState &= true \\ Update &\coloneqq [Bools \mapsto Bools] \to [Bools \mapsto Bools] \\ q \in States, x \in States &\mapsto \begin{matrix} Update(q,x)_1 = (q \neq x) \\ Update(q,x)_2 = q \end{matrix} \end{aligned}$$

The transition table for any system can be constructed by laying down all possible present state-input combinations and manually plugging them into the Update function, calculating the values for the next state and the output. For the parity system we get:

Present state	Input	Next state	Output
true	true	false	true
true	false	true	true
false	true	true	false
false	false	false	false

The transition diagram can be constructed by laying down as many graph nodes as there are states in the machine's States set, and then adding the edges between them according to the transition table. For the parity system we get:



Observing the transition diagram for a machine we can define some useful properties.

Definition 2.1 (Determinism)
*A state machine is **deterministic** if and only if each state and input has at most one outgoing edge.*

Definition 2.2 (Receptiveness)
*A state machine is **receptive** if and only if each state and input has at least one outgoing edge.*

Definition 2.3 (Unreachability)
*A state q of a machine M is **unreachable** if and only if there is no path on the transition diagram of M that passes through q (except q's self-loops).*

Definition 2.4 (Moore machine)
*A state machine is **Moore** if its outputs are determined only by the current state.*

Definition 2.5 (Mealy machine)
*A state machine is **Mealy** if its outputs are determined by the current state and the current input.*

A discrete time state machine can have multiple implementations that are equivalent to each other, i.e. they produce the same behavior.

2.2 Equivalence of deterministic state machines

Definition 2.6 (Equivalence)
*Two deterministic finite state machines M_1 and M_2 are **equivalent** if and only if:*

1. *Inputs(M_1)=Inputs[M_2]*
2. *Outputs(M_1)=Outputs[M_2]*
3. $\forall x \in [\mathbb{N}_0 \mapsto Inputs],\ M_1(x) = M_2(x)$

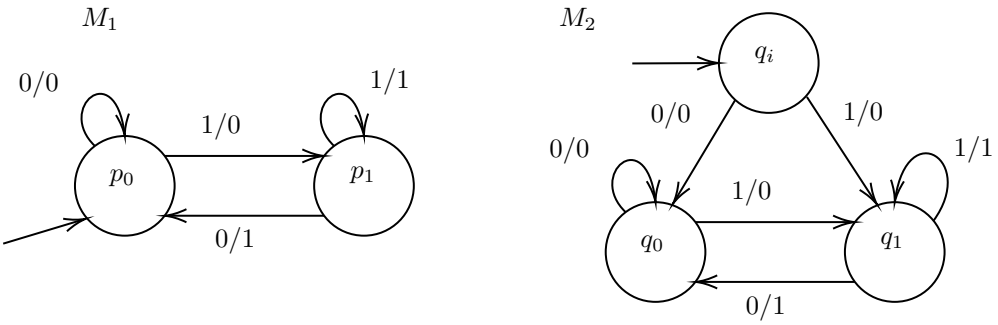
The concept of equivalence between state machines deals with inputs and outputs, so potentially with an infinite amount of values to cross check between the machines. This means that equivalence of state machines defined in this way cannot be checked algorithmically. Since we are working with state machines we can switch from checking input/output relations to checking the state spaces of the two machines with the notion of bisimulation.

Definition 2.7 (Bisimulation)
*Given two deterministic state machines M_1 and M_2, a **bisimulation** is a binary relationship between States(M_1) and States[M_2] such that $B \subseteq States(M_1) \times States[M_2]$ and:*

1. *InitialState(M_1)=InitialState[M_2]*
2. *$\forall p \in States[M_1], q \in States[M_2]$ if $(p, q) \in B$ then $\forall x \in Inputs[M_1]$*
   $Outputs[M_1](p, x) = Outputs[M_2](q, x) \wedge (NextState[M_1](p, x), NextState[M_2](q, x)) \in B$

Theorem 2.1 (Equivalence and bisimulation)
Two deterministic state machines M_1 and M_2 are equivalent if and only if there exists a bisimulation B between them

Example 2.2
Consider the state machines M_1 and M_2:



we add the initial states of the to machines as a pair in the potential bisimulation B:

$$B = \{(p_0, q_i)\}$$

Then, following the algorithm, we check for all possible inputs of M_1 (0 and 1), which state of M_2 is reached with the same output as M_1. For input 0 we go from p_0 to p_0 and from q_i to q_0 with the same output 1, so we add the state pair (p_0, q_0) to the bisimulation:

$$B = \{(p_0, q_i), (p_0, q_0)\}$$

From the state pair (p_0, q_0), under input 0 we get with the same output to the state pair (p_0, q_0) itself, which is already in the bisimulation. Under input 1 we get with the same output to state pair (p_1, q_1), which we add to the bisimulation:

$$B = \{(p_0, q_i), (p_0, q_0), (p_1, q_1)\}$$

From here, under input 1 we self loop into state pair (p_1, q_1) and under input 0 we go back to state pair (p_0, q_0), both of which are already in the simulation. Lastly we need to check what happens in state pair (p_0, q_i) under input 1, and we see that we go to state pair (p_1, q_1), which is already in the simulation.
We have covered all reachable states of both machines, so B is a bisimulation between them and by theorem 2.1 they are equivalent.

Theorem 2.1 ties together a finite notion of bisimulation with an infinite notion of equivalence. Instead of having to check an infinite number of input/output pairs we only need to check a finite number of states and state transitions against each other to determine equivalence.

The notion of bisimilarity is useful for checking equivalence but it is not the most efficient way of solving the task. By noticing that a state machine may have redundant states and state transitions, we can try and search for a minimal state machine equivalent to the one we have. Intuitively two state machines are equivalent if their minimal state machines are isomorphic.

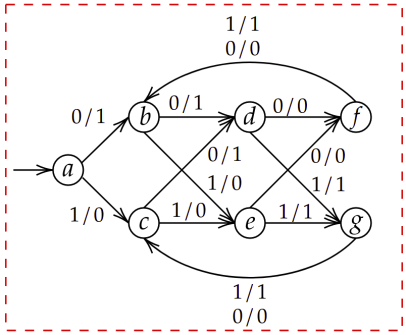
Definition 2.8 (Output split)
If there exist a state set $R \in P$, two states $r_1, r_2 \in R$ and an input $x \in Inputs$ such that $Output(r_1, x) \neq Output(r_2, x)$ then let $R_1 \coloneqq \{r \in R \mid Output(r, x) = Output(r_1, x)\}$, let $R_2 \coloneqq R \setminus R_1$ and let $P = (P \setminus \{R\}) \cup \{R_1, R_2\}$

Definition 2.9 (Next state split)
If there exist two state sets $R, R' \in P$, two states $r_1, r_2 \in R$ and an input $x \in Inputs$ such that $NextState(r_1, x) \in R \wedge NextState(r_2, x) \notin R'$ then let $R_1 \coloneqq \{r \in R \mid NextState(r_1, x) \in R'\}$, let $R_2 \coloneqq R \setminus R_1$ and let $P = (P \setminus \{R\}) \cup \{R_1, R_2\}$

Definition 2.10 (Minimization algorithm)
The minimization algorithm that produces the state machine with the fewest states bisimilar to M is as follows:

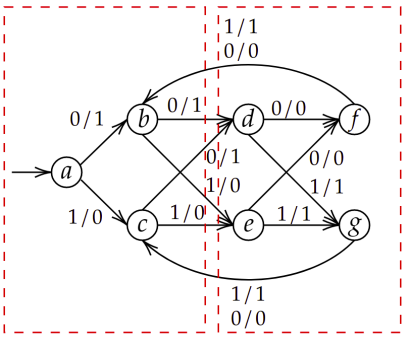
1. *Let Q be the set of all reachable states of M.*
2. *Let $P = \{Q\}$*
   a *While possible, $OutputSplit(P)$.*
   b *While possible, $NextStateSplit(P)$.*
3. *Every state set in P represents a single state of the machine bisimilar to M with the fewest states.*

Example 2.3
Consider the state machine:



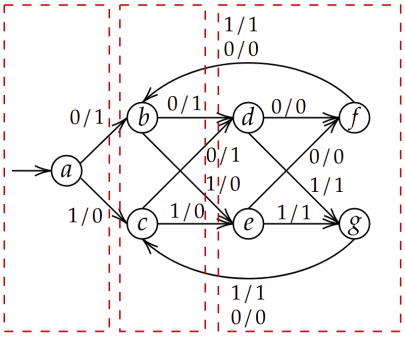
The minimization algorithm applied to the machine yields:

1. $Q = \{a, b, c, d, e, f, g\}$
2. $P = \{a, b, c, d, e, f, g\}$
   a $P = \{\{a, b, c\}\{d, e, f, g\}\}$



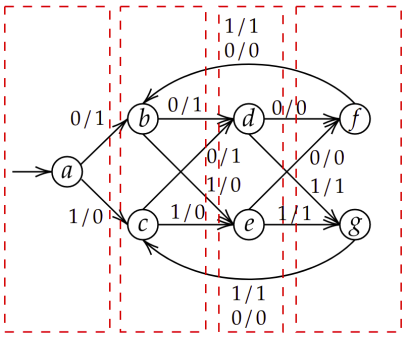
First output split, we separate the states that have transitions 0/1 and 1/0 from the states that have transitions 0/0 and 1/1. We are done here, since there are no more states with different transitions than the others inside the state sets we created.

   b $P = \{\{a\}, \{b, c\}, \{d, e, f, g\}\}$



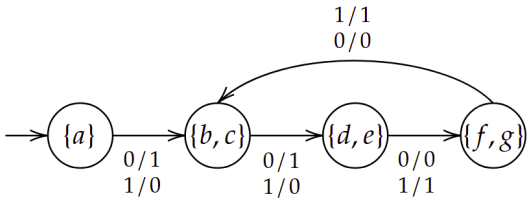
First next state split, we separate the first state set in P in two subsets: a set of states that transition to a state in the set itself and a set of states that don't.

c $P = \{\{a\}, \{b, c\}, \{d, e\}, \{f, g\}\}$



Second next state split, we separate the second state set in P in two subsets: a set of states that transition to a state in the set itself and a set of states that don't. We are done here, since there are no more subsets we can create with the next state split rule.

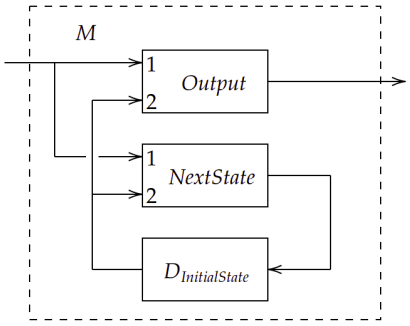
d P is the set of states of the state machine bisimilar to M with the fewest states:



Theorem 2.2 (Minimal state machine bisimulation)
A bisimulation between two deterministic state machines M_1 and M_2 exists if and only if there exists an ***isomorphism*** *between their minimal state machines.*

2.3 State machine composition

We can further split the block diagram of the generic finite state machine M by splitting the "Update" block into two memory-free sub-blocks "Output" and "NextState", which respectively evaluate the output and next state given the current state and the current input.



We now need to study whether or not any composition of finite state machines still results in a legal finite state machine.

Let's consider for example the "LastThree" system, which keeps track of whether or not the last three inputs have been "true":

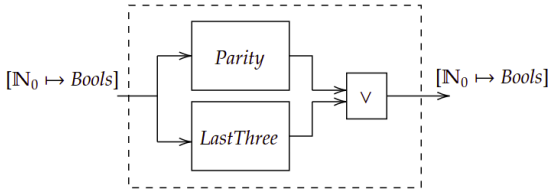
$$LastThree \coloneqq [\mathbb{N}_0 \mapsto Bools] \to [\mathbb{N}_0 \mapsto Bools]$$

$$x \in [\mathbb{N}_0 \mapsto Bools], y \in \mathbb{N}_0 \mapsto LastThree(x, y) = \begin{cases} true & TrueValues(x, y) = 3 \\ false & TrueValues(x, y) \neq 3 \end{cases}$$

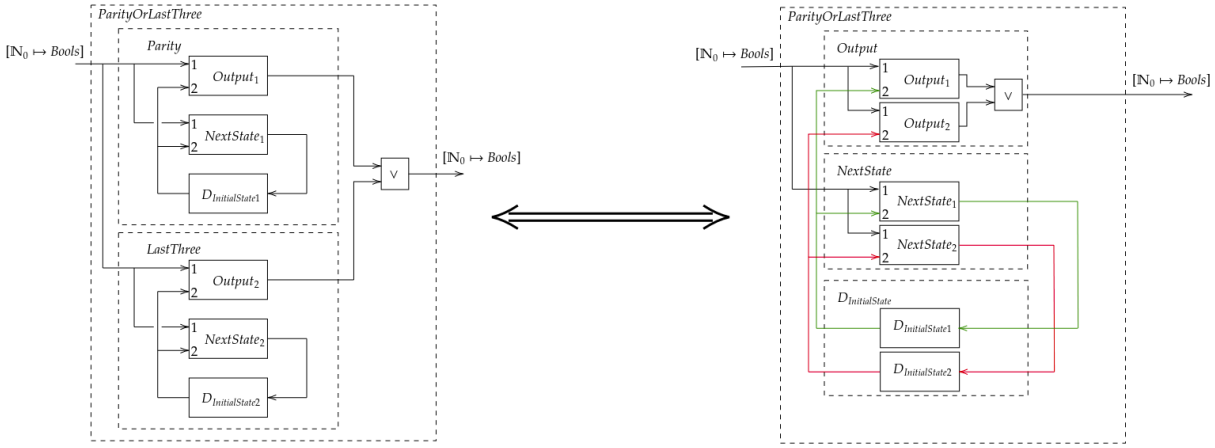
where:

$$TrueValues(x, y) \coloneqq \#\{z \in \mathbb{N}_0 \mid y - 3 \leq z < y, x(z) = true\}$$

We want to know if the parallel composition (i.e. the composition in which both machines share the input and the output is the sum of the machines' outputs) of the LastThree system with the Parity system (see example 2.1) is still a legal finite state machine.



The answer is yes, since we can rearrange the block diagrams of the two systems and get a new machine with all the blocks of the generalized model, which by definition is a legal finite state machine:

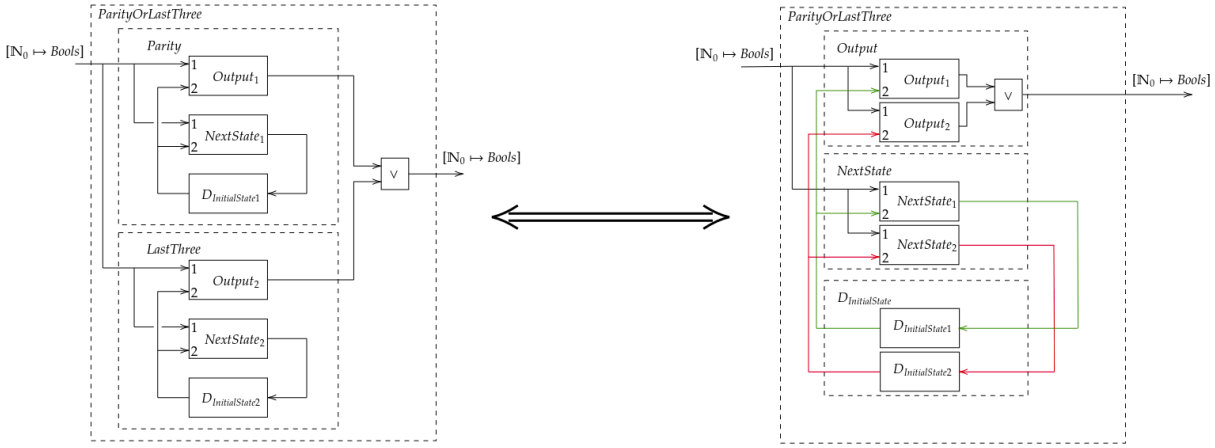


One thing to keep in mind during the rearrangement is that we need to maintain syntactical coherence between the various blocks, i.e. the output space of a given block that feeds into another block must be the same as the input space of the latter block, for all blocks.

Regarding the state space, it is intuitively true that the state space of the composite machine is equal to the cross product between the two smaller state spaces. E.g. for the LastThreeOrParity system the state space will have $2 \times 3 = 6$ states.

Theorem 2.3 (Product machine)
Any block diagram of an n-state machine with states $(s_1, s_2, \ldots, s_n)$ *can be implemented by a single state machine, called* ***product machine****, with state space* $s_1 \times s_2 \times \cdots \times s_n$*. The product machine between machines* M_1 *and* M_2 *is denoted with* $M_1 \bullet M_2$*.*

There are topologies for which the composition yields an illegal finite state machine. Consider the feedback loop with no delay elements:



The system is not legal (see definition 1.7), so its output is undetermined. Assuming the input starts at F and the output starts at F, the output of the OR block would be F, which would be negated by the NOT block in a T output,

which then would turn to T the output of the OR gate which would be negated to F ans so on. The system has no legal stable output, therefore the composition of the OR and NOT blocks in this way does not yield a legal finite state machine.

Another example would have been the same composition with an identity gate in place of the NOT gate, which would have resulted in two stable behaviors (F,F,F,... and T,T,T,...). This is not generally a problem but for our purposes we want to have an unique behavior to have a deterministic state machine.

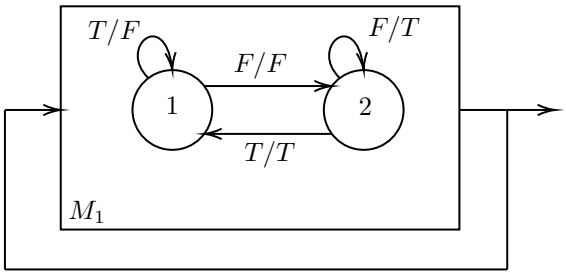
So a composition is well defined if it's syntactically coherent, meaning there are no mismatches between input-state or state-output spaces, it has no transducive cycles and its behavior is stable and unique.

Definition 2.11 (Well formed composition)
*A composition is **well formed** if and only if for every state the solutions to:*

$$Output(s, y) = y \quad Update(s, y) = (s', y)$$

exist and are unique

Example 2.4
Consider the machine composition:



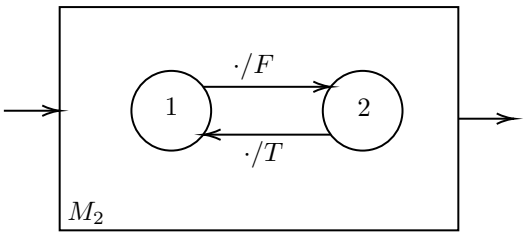
*The machine is abstract in a way, since it has no real input aside from its own output and an abstract **ticking signal** $\{\cdot\}$ that sets the order in which the machine updates its output. We now want to compute:*

$$Output(1, y) = (s, y)$$

In other words we want to know the set of state transitions that output the same value as the input. For state 1 the set has only one member: $(s, y) = (2, F)$. We have now transitioned into state 2, and we want to know once again the set of state transitions that satisfy:

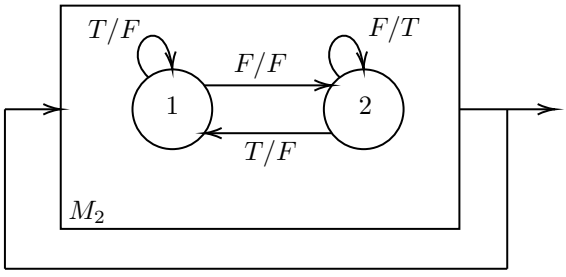
$$Output(2, y) = (s, y)$$

Once again the set has only the member $(s, y) = (1, T)$. For both states we found only one solution each, so by definition 2.11 the composition is well formed and we can replace it with a monolithic machine:



In the machine are not pictured the self loops corresponding to an absent signal, in order to reduce clutter since the absence of a signal does not influence the state.

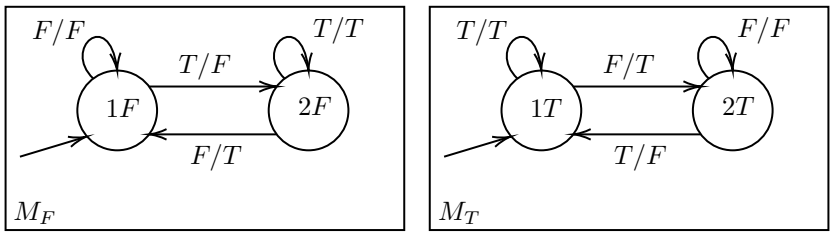
Example 2.5
Consider the machine composition:



Beginning from state 1, we can see that the set of state transitions that satisfy $Output(1, y) = (s, y)$ has only one member, $(s, y) = (2, F)$. From state 2 there is a problem, because the set of solutions of $Output(2, y) = (s, y)$ is the empty set, so no solution exists. By definition 2.11 the composition is not well defined, so we cannot implement a monolithic machine in place of the composition.

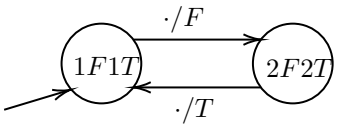
When there are multiple machines combined we must remember that the output of the first is the input of the second, whose output is the input of the third and so on. Another thing to keep in mind is that in the definition of well formed composition the output is the output of only one of the machines, depending on the design.

Example 2.6
Given the state machines:

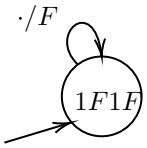


We want to analyze the compositions $M_F \bullet M_T, M_F \bullet M_F, M_T \bullet M_T$ and $M_T \bullet M_F$ (always consider relevant the output of the first machine).

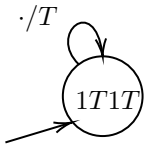
- *$M_F \bullet M_T$: From state $1F$ we either self loop to $1F$ under input F or we transition to $2F$ under input T, but in both cases we output F. Once the second machine receives F as an input it transitions to $2T$. So under any input we transition from $1F1T$ with output F to $2F2T$, therefore the transition is $\bullet/F$. Now from state $2F$ we either self loop to $2F$ under input T or transition to $1F$ under input F, in both cases outputting T. Once the second machine receives T, from state T it can only transition back to $1T$. Therefore the transition from $2F2T$ to $1F1T$ is $\bullet/T$. Having covered all states and state transitions the composition is done. It is well formed since both of the starting machines were Moore and legal.*



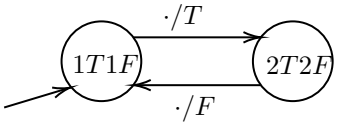
- *$M_F \bullet M_F$: From state $1F$ we can either self loop to $1F$ outputting F under input F or transition to $2F$ outputting F under input T. No matter what the initial input is, the second machine receives F, which makes it self loop to $1F$ outputting F. So the first machine always receives F after the first input, which would result either in a self loop to $1F$ or a transition from $2F$ to $1F$. In the combined machine there is therefore a single combined state $1F1F$ which self loops onto itself under any input outputting F. The composition is well formed since both starting machines were Moore and legal.*



- *$M_T \bullet M_T$: It is the mirror of $M_F \bullet M_F$*



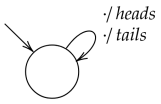
- *$M_T \bullet M_F$: It is the mirror of $M_F \bullet M_T$*



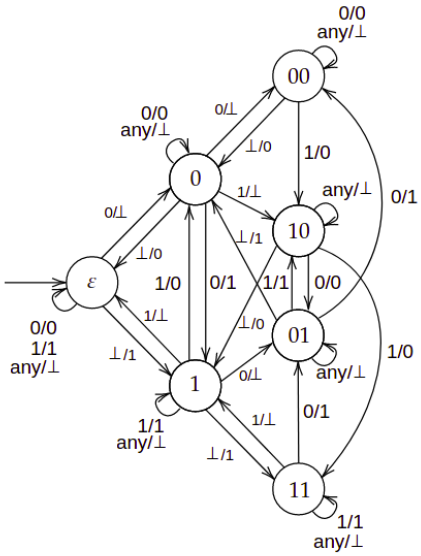
2.4 Non-deterministic finite state machines

The definition of determinism somewhat reduces the range of possible machine behaviors we can model. We introduce the property of non-determinism as a way to model things we cannot model with deterministic state machines, chiefly:

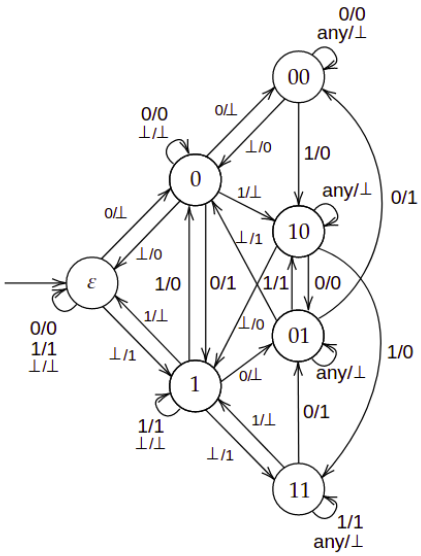
- **Randomness**, e.g. a coin toss, in which heads or tails are equally probable under the same input (the toss itself):



- **Uncertainty**, e.g. a lossy channel, which arbitrarily loses data from time to time:



- **Abstraction** or **property**, i.e. modeling a subset of behaviors of a machine. For example we can model a channel with latency, i.e. a channel that buffers the data it receives before outputting it. A buffer of size 2 (the size specification is only to have a finite memory machine) is modeled as:



In this instance the buffer of size 2 is an abstraction or property of the lossy channel, i.e. a machine that models a wider range of behaviors than some other machine, including the behaviors of the other machine itself.

Definition 2.12 (Non-determinism)
*A system S is **non-deterministic** if any state has more than one outgoing edge labeled with the same input signal.*

While deterministic system are described as functions, non-deterministic systems are viewed as **relations**:

$$NonDetSys \subseteq [Time \mapsto Inputs] \times [Time \mapsto Outputs]$$
$$x \in [Time \mapsto Inputs], y \in Time \mapsto (x, y) \in NonDetSys$$

The (x, y) pairs are called **behaviors** of the system.

2.5 Equivalence of non-deterministic state machines

For non-deterministic systems the notion of equivalence changes, to take into account the description of system behaviors.

Definition 2.13 (Refinement)
*A system S_1 **refines** (i.e. is a more detailed description of) a system S_2 (which is an abstraction or property of S_1) if and only if:*

1. $Time[S_1] = Time[S_2]$
2. $Inputs[S_1] = Inputs[S_2]$
3. $Outputs[S_1] = Outputs[S_2]$
4. $Behaviors[S_1] \subseteq Behaviors[S_2]$

Definition 2.14 (Non-deterministic equivalence)
*A system S_1 is **equivalent** to a system S_2 if and only if:*

1. $Time[S_1] = Time[S_2]$
2. $Inputs[S_1] = Inputs[S_2]$
3. $Outputs[S_1] = Outputs[S_2]$
4. $Behaviors[S_1] = Behaviors[S_2]$

Observation 2.3
Deterministic reactive discrete time systems can be implemented by deterministic state machines. Non-deterministic reactive discrete time system can be implemented by non-deterministic state machines.

For non-deterministic state machines, the 5-tuple that defines the state description of the machine becomes:

$$\{States, Inputs, Outputs, PossibleUpdates, PossibleInitialStates\}$$

- $States$ is the space of the states.
- $Inputs$ is the set of all the possible input values (or signals).
- $Outputs$ is the set of all the possible output values (or signals).
- $PossibleInitialStates \in States$ is the set of possible initial states of the machine.
- $PossibleUpdates \coloneqq States \times Inputs \to \{States \times Outputs\} \setminus \varnothing$ is called **transition function**, and it associates to each couple *present state-input* one or more *next state-output* couple.

Definition 2.15 (Simulation)
*Given two non-deterministic state machines M_1 and M_2, a binary relation $S \subseteq States[M_1] \times States[M_2]$ is a **simulation** of M_1 by M_2 if and only if:*

1. $\forall p \in PossibleInitialStates[M_1]\ \exists q \in PossibleInitialStates[M_2]$ *such that* $(p, q) \in S$, *and*
2. $\forall p \in States[M_1], \forall q \in States[M_2]$ *if* $(p, q) \in S$ *then* $\forall x \in Inputs, \forall y \in Outputs,\ \forall p' \in States[M_1]$ *if* $(p', y) \in PossibleUpdates(p, x)$ *then* $\exists q' \in States[M_2], (q', y) \in PossibleUpdates[M_2](q, x)$ *and* $(p', q') \in S$.

Theorem 2.4 (Non-deterministic refinement)
*A non-deterministic state machine M_1 **refines** the non-deterministic state machine M_2 if there exists a simulation of M_1 by M_2.*

The theorem links a condition on behaviors, with the notion of refinement, with a condition on equivalence of states, with the notion of simulation. Unlike the deterministic case, the concept of refinement is directional, meaning that if M_1 simulates M_2, it is not guaranteed that M_2 simulates M_1. This directionality of the notion of simulation suggests that it is not a powerful enough tool to determine if M_1 refines M_2, since a refinement may be possible even if no simulation can be found.

2.6 Output-deterministic finite state machines

There exists a particular class of non-deterministic machines for which there is a one to one correspondence between system behaviors and executions.

This class of machines is called **output-deterministic**.

Definition 2.16 (Output-determinism)
*A state machine M is **output-deterministic** if and only if:*

1. *There is only one initial state, and*
2. *For every state and input/output pair there is only one successive state.*

Observation 2.4
Output-deterministic state machines present fewer branching than non output-deterministic state machines, therefore if a simulation exists it is generally determinable easier than a non-deterministic state machine simulation.

Definition 2.17 (Deterministic simulation)
Given state machines M_1 and M_2, if M_2 is deterministic then M_1 is simulated by M_2 if and only if M_1 is equivalent to M_2.

Definition 2.18 (Non-deterministic simulation)
Given state machines M_1 and M_2, if M_2 is non-deterministic then M_1 is simulated by M_2 if M_1 refines M_2.

Given any non-deterministic state machine M we can find an output-deterministic state machine $det(M)$ with the process of **subset construction**, which is similar to the process of determinization of non-deterministic finite state automatons. The difference is that we cannot only consider the language of the machine but also its outputs. The drawback of the process is the increase in the number of states, which can be exponential in the extreme cases.

Definition 2.19 (Simulation and determinization)
Given two non-deterministic state machines M_1 and M_2:

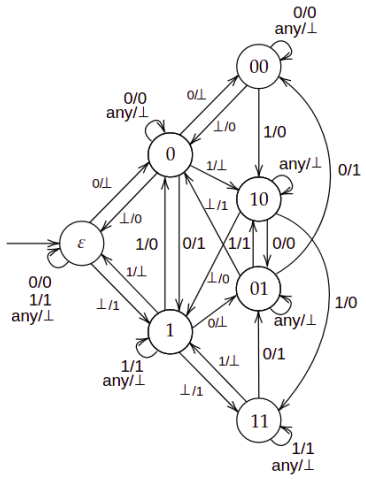
$$M_1 \text{ refines } M_2 \iff M_1 \text{ refines } det(M_2) \iff M_1 \text{ is simulated by } det(M_2)$$

Therefore subset construction reinforces the notion of simulation, and it is implemented as an algorithm that takes as input a non-deterministic state machine M and outputs an output-deterministic state machine $det(M_2)$:

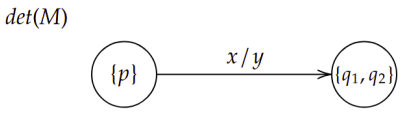
Definition 2.20 (Subset construction)
*The **subset construction algorithm** is as follows:*

1. *Let $InitialState[det(M)] = PossibleInitialStates[M]$ and $States[det(M)] = \{InitialState[det(M)]\}$*
2. *Repeat for each transition not already contained in $det(M)$:*
   - *a Choose $P \in States[det(M)], (x, y) \in Inputs \times Outputs$*
   - *b Let $Q = \{q \in States[M] \mid \exists p \in P, (q, y) \in PossibleUpdates[M](p, x)\}$*
   - *c If $Q \neq 0$ then let $States[det(M)] = States[det(M)] \cup \{Q\}$ and $Update[det(M)](P, x) = (Q, y)$.*

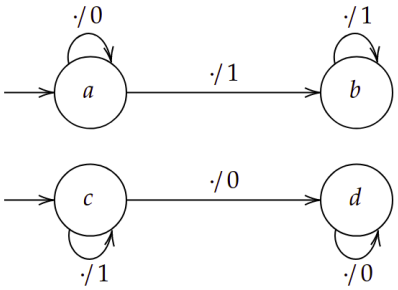
Example 2.7
Consider the machine M:



we obtain an output-deterministic machine equivalent to M by traveling along the nodes of M. In this example it's simple as there is only one: we start by adding as the initial state of machine $det(M)$ the initial state p of machine M. Then we add to a state set each state we can reach from p under the same input output label, getting the state set $Q = \{q_1, q_2\}$. The resulting machine is then:

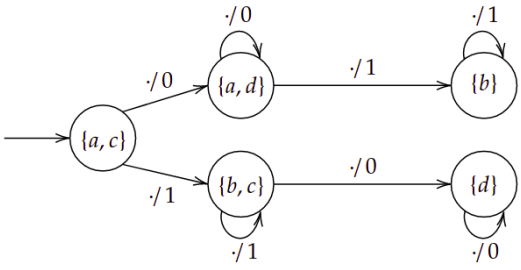


Example 2.8
Consider the machine M:



We start by adding to the initial states set of $det(M)$ all possible initial states of M, a and c. From $\{a, c\}$ we can go through the $\bullet/0$ label and reach state a through a self loop or state d through the outgoing edge, so we add the state set $\{a, d\}$ to the states of $det(M)$. Similarly for the $\bullet/1$ label, we add the state set $\{b, c\}$. Now we have to repeat the process for the state sets we just added. From $\{a, d\}$ we can go through a self loop $\bullet/0$ that brings us back to $\{a, d\}$, which we don't need to add again, or go through $\bullet/1$ to state $\{b\}$, which we add to the states of $det(M)$. Similarly for the state set $\{b, c\}$ we have a self loop for $\bullet/1$ and an outgoing edge $\bullet/0$ leading to state $\{d\}$.

The resulting machine is then:



Observation 2.5
If there exists a non-deterministic state machine that implements a system S with n states, then there exists an output-deterministic state machine that implements S with 2^n states.

Observation 2.6
There may not exist an output-deterministic state machine that implements a system S with fewer than 2^n states.

Observation 2.7
There may not exist an unique non-deterministic state machine that implements a system S with the fewest states.

To define the non-deterministic minimization algorithm we first need to expand the notion of bisimilarity in order to apply it to non-deterministic state machines.

Definition 2.21 (Non-deterministic bisimilarity)
Given two non-deterministic state machines M_1 and M_2, a binary relation $B \subseteq States[M_1] \times States[M_2]$ is a bisimulation between M_1 and M_2 if and only if:

A $\forall p \in PossibleInitialStates[M_1], \forall q \in PossibleInitialStates[M_2], (p,q) \in B$ *, and*

B $\forall p \in States[M_1], \forall q \in States[M_2]$*, if* $(p,q) \in B$ *then* $\forall x \in Inputs, \forall y \in Outputs, \forall p' \in States[M_1]$ *if* $(p',y) \in PossibleUpdates[M_1](p,x)$ *then* $\exists q' \in States[M_2]$ *such that* $(q',y) \in PossibleUpdates[M_2](q,x)$ *and* $(p',q') \in B$*, and*

C $\forall p \in PossibleInitialStates[M_1], \forall q \in PossibleInitialStates[M_2], (p,q) \in B$ *, and*

D $\forall p \in States[M_1], \forall q \in States[M_2]$*, if* $(p,q) \in B$ *then* $\forall x \in Inputs, \forall y \in Outputs, \forall q' \in States[M_2]$ *if* $(q',y) \in PossibleUpdates[M_2](q,x)$ *then* $\exists p' \in States[M_1]$ *such that* $(p',y) \in PossibleUpdates[M_1](p,x)$ *and* $(p',q') \in B$*.*

Conditions A and B ensure that M_2 simulates M_1, while conditions C and D ensure that M_1 simulates M_2. All together the result is that M_1 bisimulates M_2 and vice versa. The existence of two separate simulations is not a sufficient condition for the existence of a bisimulation though. The two simulations must be built in parallel, cross checking the state pairs introduced by one against the other.

Theorem 2.5 (Non-deterministic equivalence)
Two non-deterministic state machines M_1 and M_2 are ***equivalent*** *if and only if there exists a bisimulation B between them*

Definition 2.22 (Non-deterministic minimization)
The minimization algorithm for non-deterministic state machines is as follows:

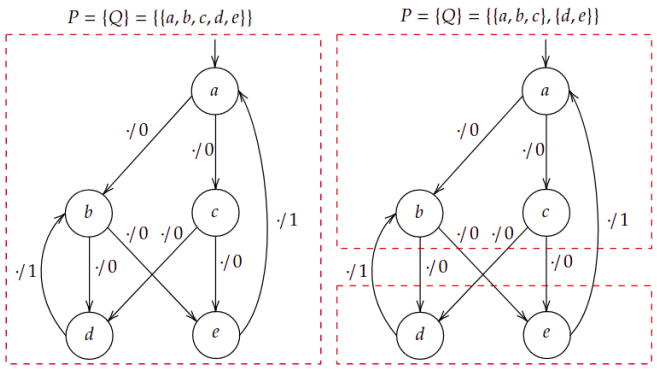
1. *Let Q be the set of all the reachable states of a non-deterministic state machine M.*
2. *Maintain a set P of state sets and:*
   a. *Let $P = \{Q\}$.*
   b. *Until no longer possible split(P).*
3. *When done P is the set of state sets that represent the non-deterministic state machine with the fewest states bisimilar to M.*

Where:

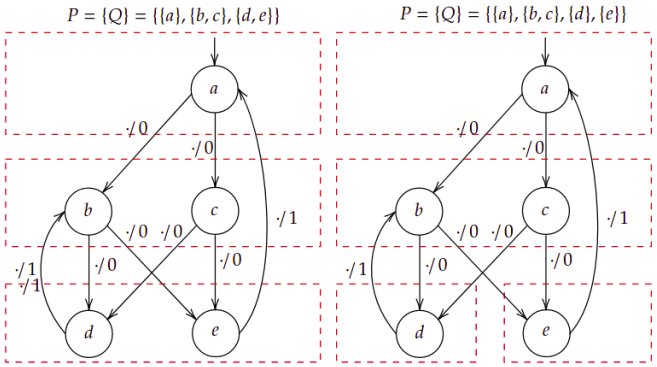
Definition 2.23 (Split)
If $\exists R \in P \wedge R' \in P$ *state sets,* $\exists r_1 \in R \wedge r_2 \in R$ *states,* $\exists x \in Inputs, y \in Outputs$ *such that* $\exists r' \in R', (r',y) \in PossibleUpdates(r_1,x) \wedge \forall r' \in R', (r',y) \notin PossibleUpdates(r_s,x)$ *then:*

1. *Let* $R_1 = \{r \in R \mid \exists r' \in R', (r',y) \in PossibleUpdates(r,x)\}$
2. *Let* $R_2 = R \setminus R_1$
3. *Let* $P = (P \setminus \{R\}) \cup \{R_1, R_2\}$

Example 2.9
Consider the non-deterministic automaton:

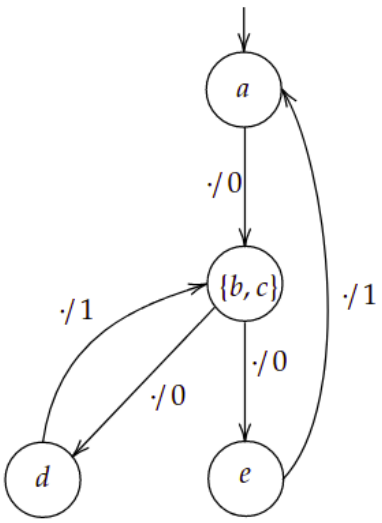


In the first iteration of split we separate the state set $\{d, e\}$ *from the state set* $\{a, b, c\}$ *since for the input/output combination* $\cdot/0$ *the former leads nowhere, while the latter leads to the state set* $\{d, e\}$.



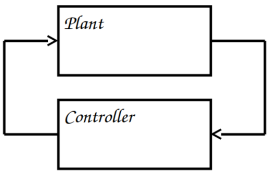
In the next iteration of split we separate the state set $\{a\}$ *from the state set* $\{b, c\}$ *since for the input/output combination* $\cdot/0$ *the former leads to the state set* $\{b, c\}$ *while the latter leads to the state set* $\{d, e\}$. *Finally we separate the state set* $\{d\}$ *from the state set* $\{e\}$ *because for the input/output combination* $\cdot/1$ *the former leads to state set* $\{b, c\}$ *while the latter leads to state set* $\{a\}$.

We then combine the state sets with multiple states into one and obtain the minimized machine:



2.7 Control

The **control problem** concerns a **plant**, i.e. for the scope of this course a finite state machine, automaton or Petri net, and an **objective**. The problem consists in finding a finite state machine called **controller** such that its composition with the plant:



satisfies the objective.

2.7.1 Safety control

The most common control problem is **safety**, in which there are constraints put on all the reachable states of a machine, some of which are marked as **error states** by the machine's designer and therefore should not be reached. The **safety objective** is to keep the machine out of a set of error states.

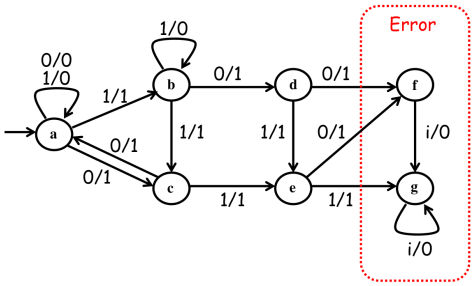
Given a finite state machine plant and a set of error states, in order to design a controller that implements the safety objective we first need to compute the set of **uncontrollable states**.

Definition 2.24 (Uncontrollable state)
*A state is **uncontrollable** if there is no way to prevent the transition from it to an error state.*

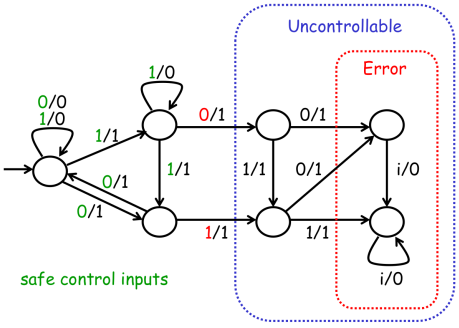
The set of uncontrollable states is constructed as follows:

1. Every state in the error state set is uncontrollable.
2. For all remaining states s:
   - if for all inputs i there exists an uncontrollable state s' and an output o such that $(s', o) \in possibleUpdates(s, i)$, then s is uncontrollable.

For example consider the following plant:



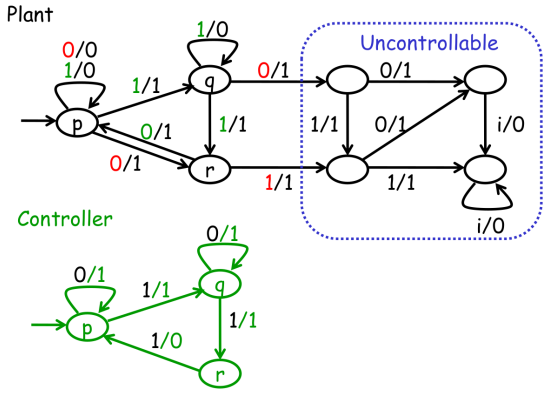
State e is uncontrollable, because once reached all its transitions lead to an error state. State d is uncontrollable as well, because it can either transition directly to an error state or to e, which is an uncontrollable state. Therefore we obtain:



Among the remaining states, which we call **controllable states**, we make another distinction on their inputs. An input that leads back to a safe state is a **safe input**, while an input that leads to an uncontrollable state is an **unsafe input**.

The first idea to design the controller is to choose for each controllable state s of the plant an input i such that $possibleUpdates(s, i)$ contains only controllable states. Then have the controller keep track of the state of the plant and send it inputs based on its outputs in order to control its state transitions and keep it into the controllable states. This means that designing a controller for output deterministic state machines is very easy, since the initial state is known and the output is produced by the plant deterministically, which can then be guided by the controller by choosing suitable control inputs based on the plant output (which is known).

Therefore for output deterministic state machines the controller is simply the controllable part of the plant with safe inputs and outputs reversed. This is intuitively correct, because the controller will constantly output inputs that force the plant to remain in its controllable part.



If the plant is not output-deterministic then the controller will know what the plant does with some degree of uncertainty. This makes control harder, since the information on the plant is limited, and the control actions must cover all possible variances in the plant behavior.

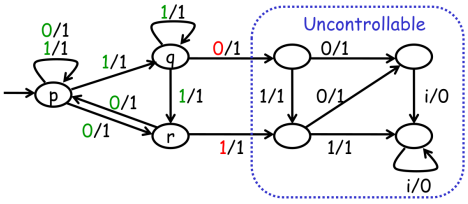
For a non-output-deterministic state machine we can design the controller in an algorithmic way:

1. Let $Controllable$ be the set of controllable states of the plant. A subset $S \subseteq Controllable$ is **consistent** if there is an input i such that for all states $s \in S$, all states in $possibleUpdates(s, i)$ are controllable.

2. Let M be the state machine whose states are the consistent subsets of $Controllable$. Prune from M the states which have no successor, until no more states can be pruned.

3. If the resulting machine contains $possibleInitialStates$ of the initial plant as a state, then it is the desired controller. Otherwise, no controller exists.

Example 2.10
Consider the plant:

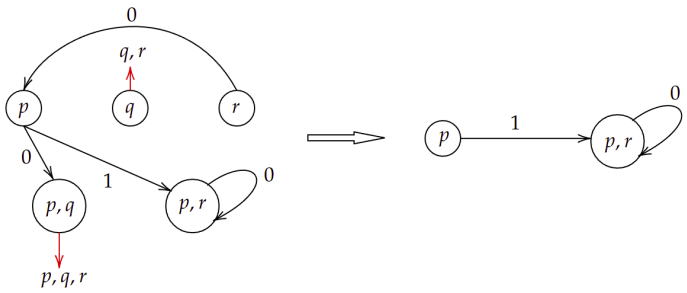


The controller can then be made receptive in any way. For example in state r input 0 is not handled, and it can be fixed by adding a 0/0 label to the outgoing edge without changing the behavior of the controller.

The set of controllable states is $\{p, q, r\}$. Therefore the subsets of it are:

$$\begin{aligned}
\{p\} &: \textit{consistent for inputs 0 and 1} \\
\{q\} &: \textit{consistent for input 1} \\
\{r\} &: \textit{consistent for input 0} \\
\{p, q\} &: \textit{consistent for input 1} \\
\{p, r\} &: \textit{consistent for input 0} \\
\{q, r\} &: \textit{not consistent} \\
\{p, q, r\} &: \textit{not consistent}
\end{aligned}$$

We then construct a machine whose states are the subsets of the controllable states of the plant, and whose state transitions are the transitions under the inputs that make the subsets consistent. Finally we prune the states that either lead to non consistent states or that have no in-going edges. The final controller is therefore:

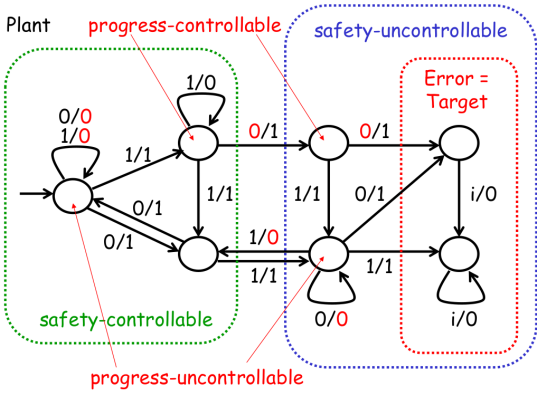


2.7.2 Progress control

Progress control is the dual case of safety control. The **progress objective** is to make the plant get to a state in a **target states set**.

In progress control the roles of the plant and the controller are reversed. This does not however imply that safety-controllable states are also progress-controllable, because the game is not symmetric (the controller always moves first).

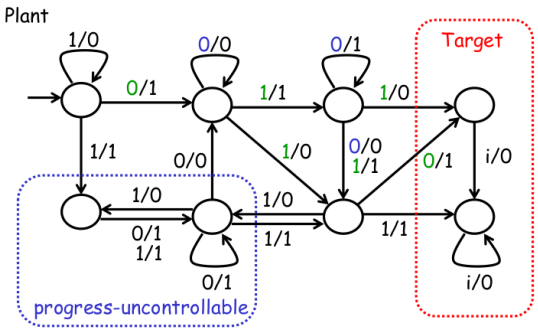
For instance, in the plant we have seen before:



We proceed in a way similar to what we have seen for safety control. First of all we compute the **progress-controllable** set of states as follows:

1. Every state in the *Target* set is progress-controllable.
2. For all states s
   - if there exists an input i such that, for all states s' and outputs o, $(s', o) \in possibleUpdates(s, i)$, then s is progress-controllable.

For example, consider the above plant:



The progress-controllable states are those from which, under a certain number of transitions, the machine can reach the states in the *Target* set. The remaining states are progress-uncontrollable.

We can introduce a distinction in the inputs, between **helpful inputs** (in green) which make the plant progress towards the target states, and **safe inputs** (in blue) which keep the plant away from progress-uncontrollable states.

Then the controller can be designed as follows:

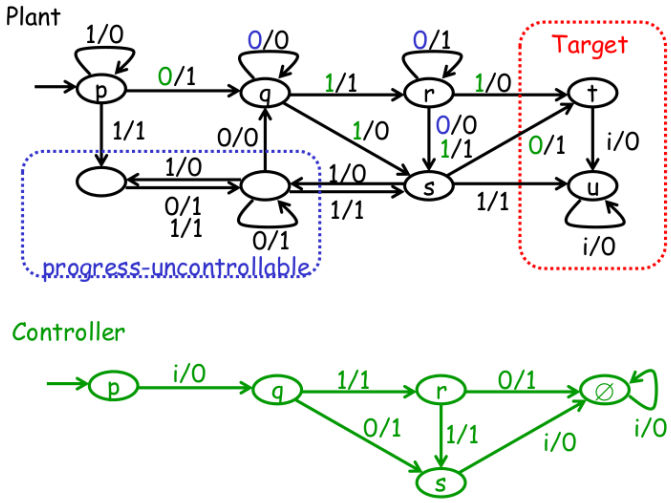
1. A subset $S \subseteq Progress - controllable$ is **consistent** if there is an input i such that, for all states $s \in S$, all states in $possibleUpdates(s, i)$ are progress-controllable.

2. Construct the state machine whose states are the consistent subsets of Progress-controllable without target states (including the empty set $\varnothing$), and whose outputs are safe. A **safe** output is a controller output (and so a plant input) which cannot lead the plant to a progress-uncontrollable state.

3. If the result contains *possibleInitialStates* (of the plant) as a state, and there is an acyclic, output-closed subgraph from *possibleInitialStates* to $\varnothing$, then prune away all states not in the subgraph. The resulting subgraph is the desired Controller. Otherwise, no controller exists. A subgraph is output-closed if transitions with safe but not helpful outputs are removed.

As seen with the safety problem, if the plant is output-deterministic we only need to consider subsets of size 1. In other words the controller always knows the state of the plant.

For example:



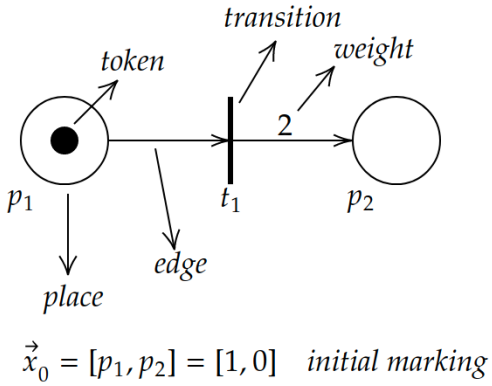
3 Petri nets

Finite state machines are not the only existing model of system behavior. For instance, as we have seen with machine composition, finite state machines are not a good model for **concurrency**, i.e. the out of order execution of parts of the system. Among the mathematical models developed to deal with concurrency, we will now focus on **Petri nets**.

A Petri net is a graph which is **bipartite** (meaning there are two types of nodes and the edges always go from one type to the other), **oriented** (meaning the edges have a direction) and **weighted** (meaning that each edge has an associated weight assigned to it). The two types of node in a Petri net are called **places** and **transitions**. Places act as a holder for **tokens**, which are abstract objects that travel through the net and get consumed when they reach a transition ("firing" a transition), which then produces a new token that will end up in another place.

Definition 3.1 (Petri net)
*A **Petri net** is a 5-tuple $N = (P, T, A, w, \vec{x}_0)$ where:*

- *P is a finite set of nodes called **places**.*
- *T is a finite set of nodes called **transitions**.*
- *A is a set of **edges**: $A \subseteq (P \times T) \cup (T \times P)$.*
- *w is a function that associates a **weight** to each edge: $w := A \to N$. The weight of an edge describes its token capacity.*
- *$\vec{x}_0 = [x(p_1), \ldots, x(p_n)] \in \mathbb{N}^{|P|}$ is an **initial marking vector**, which hold information about the initial distribution of **tokens** in the net.*



Definition 3.2 (Input/output places)
Let $N = (P, T, A, w, \vec{x}_0)$ be a Petri net. The set:

$$I(t) := \{p \in P \mid (p, t) \in A\}$$

*is the set of **input places** of transition $t \in T$, i.e. the set of states that reach t. The set:*

$$O(t) := \{p \in P \mid (t, p) \in A\}$$

*is the set of **output places** of transition $t \in T$, i.e. the set of states reachable by t.*

Definition 3.3 (Firing rule)
*A transition t is **enabled** in state $\vec{x}$ if the **firing rule** is respected, i.e. if:*

$$x(p) \geq w(p, t), \forall p \in I(t)$$

In other words a transition can fire (i.e. consume tokens and produce new ones) if in its input places there are at least as many tokens as the cumulative weight of the ingoing edges.

Observation 3.1
The firing rule tells us whether or not a transition can fire, but not if it will fire. The dynamics of the system are not yet defined.

In the example above, the transition can fire since there is one token in p_1 and the weight of the edge going to t_1 is unspecified (which defaults to 1). The weight of the outgoing edge to p_2 is 2, therefore before a token is generated in p_2, t_1 must fire two times.

The way in which tokens are consumed and produced via the firing of an enabled transition is defined by a function associated to each transition.

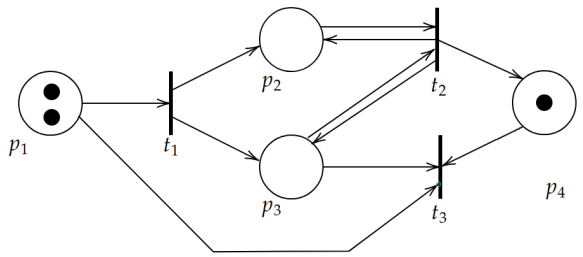
Definition 3.4 (Transition function)
*Let $N = (P, T, A, w, \vec{x}_0)$ be a Petri net with $P = \{p_0, \ldots, p_{n-1}\}$ and $\vec{x} = \{x(p_0), \ldots, x(p_{n-1})\}$ a marking for the n places. The **transition function** for a given transition t is defined as follows:*

$$G(\vec{x}, t) = \begin{cases} \vec{x}' & \text{if } x(p) \geq w(p,t), \forall p \in I(t) \\ \vec{x} & \text{otherwise} \end{cases}$$

$$\text{with} \quad \vec{x}' = [x'(p_0), \ldots, x'(p_{n-1})]$$
$$x'(p_i) = x(p_i) - w(p_i, t) + w(t, p_i) \quad \forall 0 \leq i < n$$

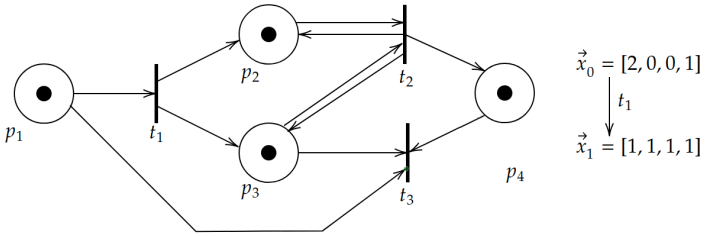
The transition function formally defines the behavior of tokens in the net. When a transition is enabled and fired, the tokens in the input places are consumed by an amount equal to the cumulative weight of the edges connecting the places to the transition. Furthermore the amount of tokens is increased if there are edges going from the transition to the input places, by an amount equal to the weight of each edge.

Example 3.1
Consider the following Petri net:

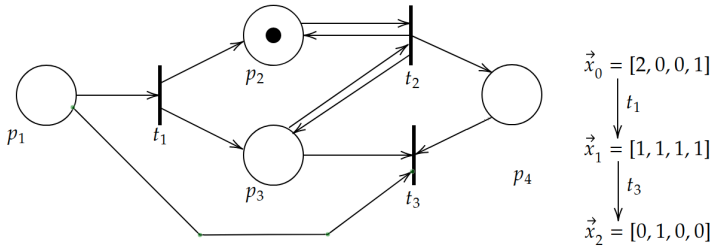


- $P = \{p_1, p_2, p_3, p_4\}$
- $T = \{t_1, t_2, t_3\}$
- $A = \{(p_1, t_1), (p_1, t_3), (p_2, t_2), (p_3, t_2), (p_3, t_3), (p_4, t_3), (t_1, p_2), (t_1, p_3), (t_2, p_2), (t_2, p_3), (t_2, p_4)\}$
- $w(a) = 1 \forall a \in A$
- $\vec{x}_0 = [2, 0, 0, 1]$

We can immediately say that transitions t_2 and t_3 are not enabled, since they have empty input places. Transition t_1 on the other hand is enabled, and when it fires a token from p_1 is consumed and a token each in p_2 and p_3 is generated:



At this point both t_2 and t_3 become enabled. We can arbitrarily choose which one to fire, but we cannot fire them simultaneously. If for example we fire t_3, a token is consumed from p_1, p_3 and p_4 each, and no new token is generated:



At this point no transition is enabled, so no update to the net can be determined. If we chose to fire t_2 instead, the number of tokens in p_2 and p_3 would have been unchanged, since the transition consumes their token but immediately generates a new one in them. At the same time we would have had an accumulation of tokens in p_4, which is a behavior different from that of finite state machines, since we can accumulate an infinite number of tokens in p_4 (which is similar to having an infinite number of states) but we have a finite description of the system.

The last example shows how even a simple Petri net can present a lot of branching in the transitions we can make and the state (in this instance state refers to the marking of a net) of the net we can reach.

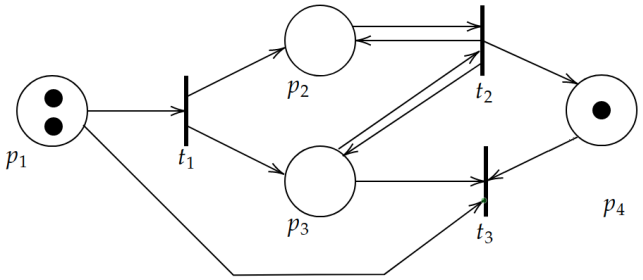
Definition 3.5 (Immediate reachability)
*Let $N = (P, T, A, w, \vec{x}_0)$ be a Petri net and $\vec{x}$ one of its states. A state $\vec{y}$ is **immediately reachable** from $\vec{x}$ if there exists a transition $t \in T$ such that $G(\vec{x}, t) = \vec{y}$.*

Definition 3.6 (Reachability set)
Let $N = (P, T, A, w, \vec{x}_0)$ be a Petri net. The smallest set of states $R(\vec{x})$ defined as:

- $\vec{x} \in R(\vec{x})$
- *if $\vec{y} \in R(\vec{x})$ and $\vec{z} = G(\vec{y}, t)$ for some $t \in T$ then $\vec{z} \in R(\vec{x})$*

*is called **reachability set**, which represents the smallest set of states reachable from state $\vec{x}$.*

Example 3.2
Considering the Petri net of the last example:



We can build its reachability set:

$$
\begin{aligned}
R(\vec{x}_0) &= R_1 \cup R_2 \cup R_3 \cup R_4 \\
R_1 &= \{\vec{x}_0\} \\
t_1 \to t_2 \quad R_2 &= \{\vec{y} \mid \vec{y} = [1,1,1,n], n \geq 1\} \\
t_1 \to t_2 \to t_2 \quad R_3 &= \{\vec{y} \mid \vec{y} = [0,2,2,n], n \geq 1\} \\
t_1 \to t_2 \to t_2 \to t_2 \to \ldots \quad R_4 &= \{\vec{y} \mid \vec{y} = [0,1,0,n], n \geq 0\}
\end{aligned}
$$

The reachability set is built by exploring all possible branchings, until no more transitions that produce a state not produced by previous transitions can be found. For example R_4 could have been found also by running only the t_3 transition.

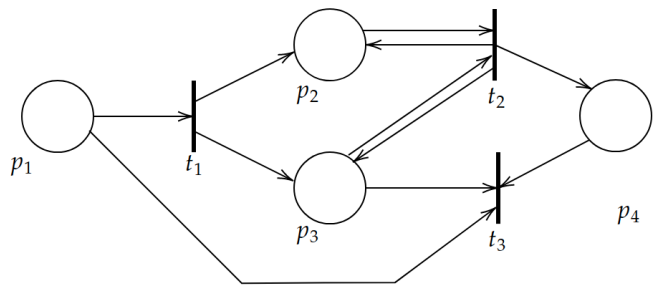
Since the possible markings are potentially infinite, and the exploration of the branches can get quite complicated, we introduce some definitions to tie the firing of transitions to the topology of the net, in order to transform the computation of firings into a computation on the topology of the net, which simplifies the problem by allowing us to utilize linear algebra tools.

Definition 3.7 (Firing vector)
*Let $N = (P, T, A, w, \vec{x}_0)$ be a Petri net with $P = \{p_1, \ldots, p_n\}$ and $T = \{t_1, \ldots, t_m\}$. A **firing vector** $\vec{u} = [0, \ldots, 0, 1, 0, \ldots, 0]$ is a vector of length m where entry $j, 1 \geq j \geq m$ corresponds to transition t_j. All entries of the vector are 0 but one, where it has a value of 1. If entry j is 1, transition t_j fires.*

Definition 3.8 (Incidence matrix)
*The **incidence matrix** $\mathcal{A}$ is an $m \times n$ matrix whose (i, j) entry is:*

$$a_{j,i} = w(t_j, p_i) - w(p_i, t_j)$$

Example 3.3
Consider the Petri net:



By applying the definition of the components of the incidence matrix we can write the incidence matrix for the net as:

$$\mathcal{A} = \begin{bmatrix} -1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -1 & 0 & -1 & -1 \end{bmatrix}$$

The incidence matrix is a representation of the connections between transitions and places (and vice versa) in the net.

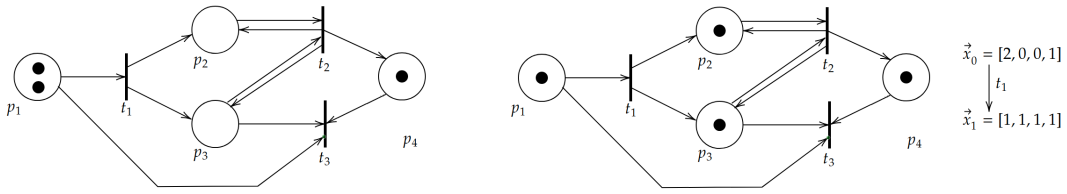
Looking at the first row for example, which is related to t_1, we can see that when t_1 fires p_1 loses a token, since $a_{11} = -1$, p_2 and p_3 acquire a token, since $a_{12} = 1$ and $a_{13} = 1$, while p_4 is unchanged, since $a_{14} = 0$.

The power of this representation is that it lets us write a **state equation** that lets us immediately calculate a state transition given the incidence matrix.

Definition 3.9 (State equation)
*Let $N = (P, T, A, w, \vec{x}_0)$ be a Petri net, $\mathcal{A}$ its incidence matrix, $\vec{x}$ its current state and $\vec{u}_i$ the firing vector of transition t_i. The state $\vec{x}'$ reached after the firing of transition t_i is given by the **state equation**:*

$$\vec{x}' = \vec{x} + \vec{u}\mathcal{A}$$

Example 3.4
Considering the first example we have seen, we want to calculate the state transition using the state equation.



The next state is given by:

$$\begin{aligned} \vec{x}_1 &= \vec{x}_0 + \vec{u}_1\mathcal{A} \\ &= \begin{bmatrix} 2 & 0 & 0 & 1 \end{bmatrix} + \begin{bmatrix} 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} -1 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ -1 & 0 & -1 & -1 \end{bmatrix} \\ &= \begin{bmatrix} 2 & 0 & 0 & 1 \end{bmatrix} + \begin{bmatrix} -1 & 1 & 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 1 \end{bmatrix} \end{aligned}$$

This formalism, while useful, isn't always correct. In fact it doesn't take into account the status of each transition after a state computation, i.e. it doesn't consider that some transitions may become disabled after a state computation.

To correct the formalism we update the definition of the state equation.

Definition 3.10 (Transition sequence evaluation)
Let $N = (P, T, A, w, \vec{x}_0)$ be a Petri net and $T' = \{t_1, \ldots, t_i, \ldots, t_n\}, t_i \in T$ a sequence of n transitions with $\vec{u}_t$ the transition vector for t.

The state after firing all of the transitions in T' is:

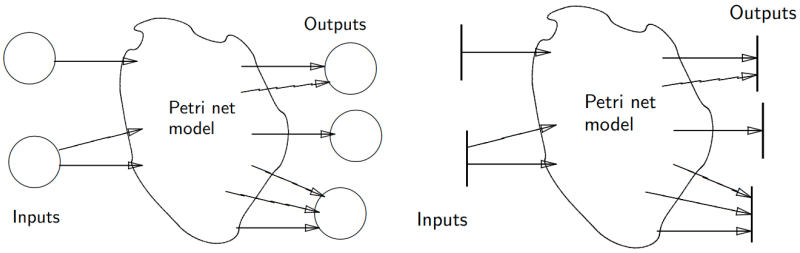
$$\vec{x}_n = \vec{x}_0 + \mathcal{A} \sum_{t \in T'} \vec{u}_t$$

provided that, for all $t_i \in T'$, t_i is enabled in the state:

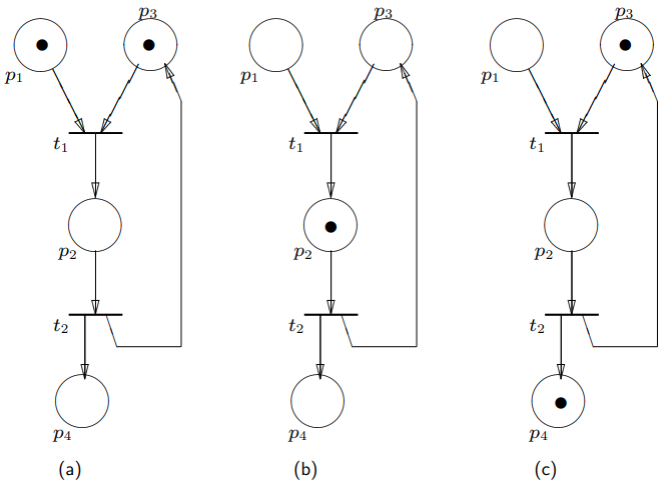
$$\vec{x}_{i-1} = \vec{x}_0 + \mathcal{A} \sum_{t \in (t_1, \ldots, t_{i-1}} \vec{u}_t$$

3.1 System modeling with Petri nets

First of all the external inputs and outputs can either be modeled as places or transitions. In fact there exists a duality between transitions and places that lets us replace either node type with the other.

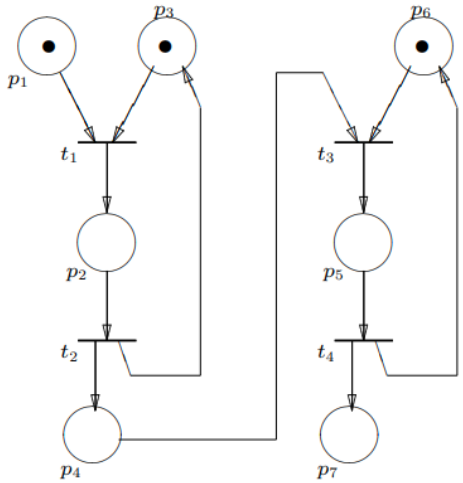


Example 3.5 (Server)
Consider the Petri net on the left:



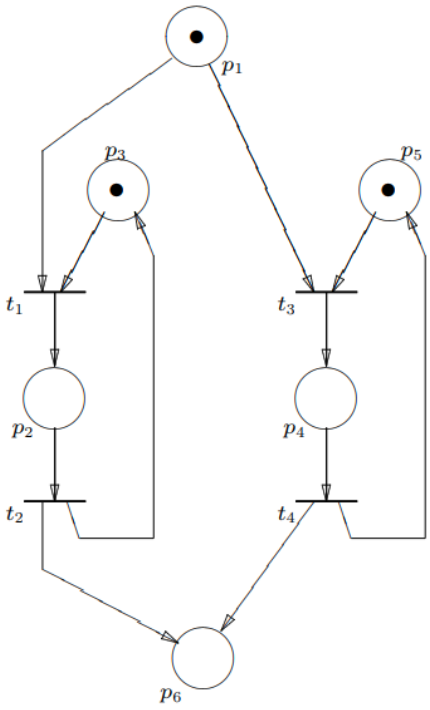
The net models a server, i.e. a service being brought to a customer. The service is modeled via transitions, while the customer is modeled as a token. The loop on the right acts as a locking mechanism to signal a busy server. When a customer arrives in p_1, t_1 fires only if the server is not already busy with another customer, i.e. if there is a token in p_3. Then the customer token and the busy token are consumed, and a token is produced in p_2 (b), after which the service will be delivered via transition t_2, which both produces a token in the output place p_4 and in the "busy" place p_2, resetting the busy lock (c).

Example 3.6 (Sequential Petri net composition)
One of the strong advantages Petri nets have over finite state machine is the ease of machine composition, which with Petri nets is immediate. For example consider the composition of two identical server nets:



The resulting net models a server that serves two service sequentially to the customer. The composition is automatically a new legal Petri net, and all it required was the connection with an edge of the output of the first net to the input of the second.

Example 3.7 (Parallel Petri net composition)
A similar concept to the previous example is the parallel composition of Petri nets:



The resulting net, which once again was immediately obtained, can be seen as the model of a parallel queue, in which multiple customers are served in parallel.

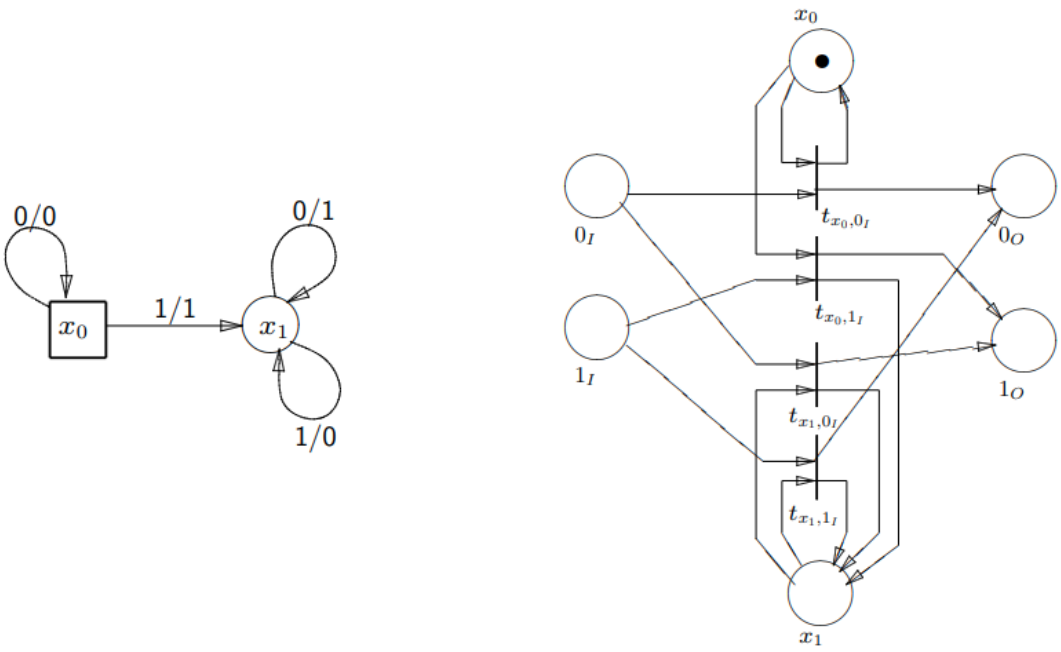
As we have seen, finite state machines are less powerful than Petri nets, because the latter can model infinite states with a finite machine. Intuitively then finite state machines are a subset of Petri nets, and it should be possible to find an equivalent Petri net to any finite state machine.

Definition 3.11 (FSM equivalent Petri net)
Given a finite state machine $M = (\Sigma, \Delta, X, x_0, g, f)$ with mutually exclusive sets Σ and Δ, an equivalent Petri net is $N = (P, T, A, \vec{y}_0)$, with:

- $P = X \cup \Sigma \cup \Delta$
- $T = \{t_{x,a} \mid x \in X, a \in \Sigma\}$
- $A = I(t_{x,a}) \cup O(t_{x,a}) \forall t_{x,a} \in T$
- $I(t_{x,a} = \{x, a\}$
- $O(t_{x,a} = \{g(x, a), f(x, a)\}$
- $\vec{y}_0 = [1, 0, \ldots, 0]$

where Σ are input places, Δ are output places, X are internal places, each $(state, input)$ pair of M becomes a transition of N and the initial marking represents state x_0 with no input.

Example 3.8
Consider the finite state machine on the left, which implements the computation of the 2-complement of a binary number:



The machine has two possible inputs, 0 and 1, so we start from two places, 0_I and 1_I. Then we add as many places as there are states in the finite state machine, so we add two places: x_0 and x_1. The we add as many transitions as there are arcs in the finite state machine, so four. The connectivity between places and transitions reflects the connectivity between states in the finite state machine but also input/output pairs labeling the edges between states. For transition $t_{x_0,0_I}$ for instance, it is related to the $0/0$ self loop of state x_0 of the finite state machine. Therefore we add an edge from the 0 input (0_I) to the transition and an edge from the transition to the 0 output (0_O) to take into the account the input/output pair. Then the $0/0$ edge reflects a self loop of x_0 onto x_0, so we add a pair of edges going to and from place x_0 to and from the transition. Having exhausted all edges for that transition we can move over to the others, for which the process is the same. Finally, since a finite state machine has only one initial state, we put a token in place x_0, which represents the initial marking. When the process is done, the Petri net equivalent to the finite state machine will be similar to the one on the right.

3.2 Analysis methods of Petri nets

There are a number of important properties to deal with when working with Petri nets, such as boundedness, conservation, deadlock, persistence and reachability. A fundamental notion to study these properties is the notion of coverability.

3.2.1 Boundedness

Definition 3.12 (k-bounded place)
*Given a Petri net $N = (P, T, A, w, \vec{x}_0)$, a place $p \in P$ in it is **k-bounded** (or **k-safe**) if:*

$$\forall \vec{y} \in R(\vec{x}_0), \;\; y(p) \leq k$$

Definition 3.13 (k-bounded Petri net)
*A Petri net is **k-bounded** (or **k-safe**) if all places $p_i \in P$ are k-bounded.*

In other words a place is k-bounded if all of its reachable markings don't contain a number of tokens greater of some number k. For example, the place p_4 in the Petri net in example 3.1 is not k-bounded, since it can accumulate an infinite amount of tokens. Therefore the Petri net is also not k-bounded.

3.2.2 Conservation

Definition 3.14 (Strict conservation)
A Petri net $N = (P, T, A, w, \vec{x}_0)$ *is* ***strictly conservative*** *if:*

$$\forall \vec{y} \in R(\vec{x}_0), \quad \sum_{p \in P} y(p) = \sum_{p \in P} x_0(p)$$

In other words conservation means that the total number of tokens is unchanged in all the markings of the net, i.e. that the sum of tokens in any given marking is invariant.

Definition 3.15 (Conservation with respect to a weighting vector)
A Petri net $N = (P, T, A, w, \vec{x}_0)$ *with* n *places is* ***conservative with respect to a weighting vector*** $\vec{\gamma} = [\gamma_1, \ldots, \gamma_n]$ *with* $\gamma_i \in \mathbb{N}$ *if:*

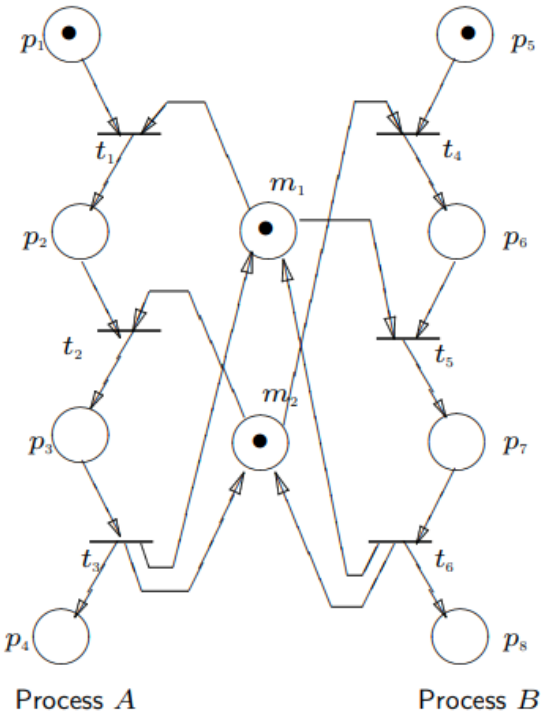
$$\forall \vec{y} \in R(\vec{x}_0), \quad \sum_{i=1}^{n} \gamma_i x(p) = const \qquad with\ \vec{x} \in R(\vec{x}_0)$$

It makes sense to introduce weighting in the definition of conservation, since it lets us separate some places from others by assigning different weights to them. For instance it can be useful to ignore the information on output places, in which accumulation of tokens could happen but it wouldn't influence the net itself.

Definition 3.16 (Conservation)
A Petri net is ***conservative*** *if it's conservative with respect to a weighing vector with strictly positive coordinates.*

3.2.3 Deadlock

Deadlock is the net-locking phenomenon that happens when different parts of a Petri net are blocking the execution of the other by means of consuming tokens. Consider the Petri nets "Process A" and "Process B":



Both processes need to access some resources, modeled as tokens contained in places m_1 and m_2. The problem of deadlock arises because when the processes start and t_1 and t_4 fire, both tokens in m_1 and m_2 are consumed. Then when process A reaches t_2, it cannot fire since the needed token in m_2 was consumed by t_4. Vice versa, when process B reaches t_3 it cannot fire, since the needed token in m_1 was consumed by t_1. We reach a situation in which either net is stuck waiting for the other to release a resource, situation which is called **deadlock**. In general deadlocking happens when some transitions in some sub-nets consume tokens contained in the same places but in different order.

3.2.4 Liveness

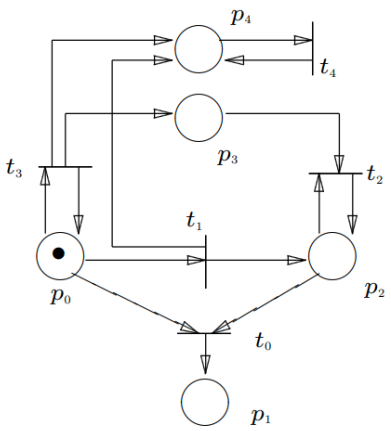
Definition 3.17 (Liveness)
Let $N = (P, T, A, w, \vec{x}_0)$ be a Petri net and $\vec{x}$ a state reachable from $\vec{x}_0$ ($\vec{x} \in R(\vec{x}_0)$). A transition $t \in T$ is called:

- ***Live at level 0 (L0-live)*** *in state $\vec{x}$ if it cannot fire in any state reachable from $\vec{x}$, i.e. it is deadlocked.*
- ***Live at level 1 (L1-live)*** *in state $\vec{x}$ if it is potentially fire-able, i.e. there exists a $\vec{y} \in R(\vec{x})$ such that t is enabled in $\vec{y}$.*
- ***Live at level 2 (L2-live)*** *in state $\vec{x}$ if for every integer n there exists a firing sequence in which t occurs at least n times.*
- ***Live at level 3 (L3-live)*** *in state $\vec{x}$ if there is an infinite firing sequence in which t occurs infinitely often.*
- ***Live at level 4 (L4-live)*** *in state $\vec{x}$ if it is L1-live in every $\vec{y} \in R(\vec{x})$.*

A Petri net is ***live at level i*** *if every transition is live at level i.*

Liveness has to deal with progressiveness, i.e. with the description of what a Petri net (or transition more specifically) "can do next".

Example 3.9
Consider the Petri net:



We study the liveness of each transition in the net:

- *t_0 is dead (i.e. it's L0-live), since it needs a token from both p_0 and p_2 in order to fire, but this is impossible, since the only transition that brings a token to p_2 is t_1, which however also consumes a token from p_0. Since in p_0 there can ever be only one token, it's impossible to have a token in both p_0 and p_2, therefore t_0 can never fire and it is L0-live.*
- *t_1 is L1-live, since there exists a marking in which it is enabled. In fact it can already fire, but by doing so it consumes the only available token in p_0, therefore disabling t_1 in the following states. It is therefore L1-live.*
- *t_2 is L2-live, since we can construct an arbitrary sequence in which t_2 can fire n times. By firing t_3 n times we can accumulate n tokens in p_3, then by firing t_1 we enable t_2, which can then be fired n times consuming the tokens accumulated in p_3. t_2 is therefore L2-live.*
- *t_3 is L3-live, since it starts out enabled and the only condition which disables it is the firing of t_1, which would consume the token in p_0. We can however define a valid firing sequence of an infinite amount of firings of t_3, thus making it L3-live.*
- *t_4 is L4-live, since whatever choice of firing (t_3 or t_1) we make at the beginning, a token ends up in p_4, enabling t_4 which then starts out as L1-live. Then whatever firing happens doesn't change the fact that t_4 stays L1-live, because the token in p_4 is immediately restored when it is consumed by t_4. Since t_4 is L1-live in every possible state, it is L4-live.*

By looking at t_2 we can also see the difference between L2-live and L3-live. Intuitively one could think that the two definitions could be compacted in a single definition, but a problem arises. While it is true that we could accumulate an infinite amount of tokens in p_3 in order to fire t_2 an infinite amount of times, and therefore making it L3-live, there is the issue that t_2 is enabled by the firing of t_1, regardless of how many tokens are in p_3 (given that there is at least one). Therefore we cannot construct a valid firing sequence, because we cannot place the firing of t_1 after an infinite amount of firings of t_3 in a meaningful mathematical way.

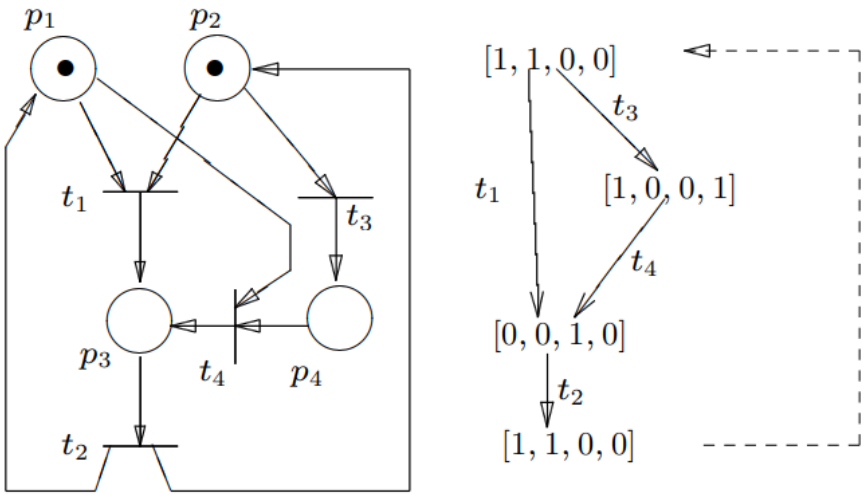
3.2.5 Persistence

Definition 3.18 (Persistence)
*Two transitions are **persistent with respect to each other** if, when both are enabled, the firing of one doesn't disable the other.*

*A Petri net is **persistent** if any two transitions are persistent with respect of to each other.*

3.2.6 Coverability

The notion of coverability has to do with the problem of determining the set of reachable markings of a Petri net.



As we have seen before, this is a complex task even for small nets because of the amount of branching involved. More specifically because of the possibility of accumulation of tokens, the graph of reachable markings, which is called **coverability tree**, can become infinite. The idea is to replace the places where there is an accumulation of tokens with a symbol ω (ω**-enhanched** place), in order to describe an infinite amount of behavior of accumulation of tokens in the place. This approach however results in a loss of information on *how* the tokens are accumulating in the place, therefore the question "is this marking reachable?" cannot be answered in this way.

Definition 3.19 (Coverability)
Let $N = (P, T, A, w, \vec{x}_0)$ be a Petri net and $\vec{x}$ and $\vec{y}$ arbitrary states.

*State $\vec{x}$ **covers** state $\vec{y}$ if in $\vec{x}$ at least all transitions which are enabled in $\vec{y}$ are enabled in $\vec{x}$:*

$$x(p) \geq y(p) \quad \forall p \in P$$

*State $\vec{x}$ **strictly covers** state $\vec{y}$ if $\vec{x}$ covers $\vec{y}$ and, in addition:*

$$\exists p \in P \mid x(p) > y(p)$$

*Let $\vec{x} \in R(\vec{x}_0)$. State $\vec{y}$ is **coverable by** $\tilde{\mathbf{x}}$ if and only if there exists a state $\vec{x}' \in R(\vec{x})$ such that $\vec{x}'(p) \geq y(p)$ for all $p \in P$.*

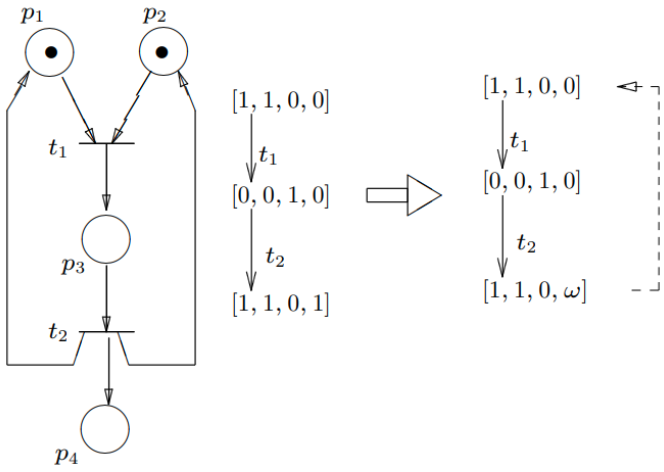
The notion of covering deals with the distribution of tokens in places. A state covers another state if place by place it has more tokens than the other state. A state strictly covers another state if it covers it and if in at least one place it has more tokens than the other. Furthermore the notion of coverability adds reachable states into the mix. A state is coverable by another state if in the reachable set of the latter state there is a state covering the first one.

Definition 3.20 (Coverability tree)
*Let $N = (P, T, A, w, \vec{x}_0)$ be a Petri net. A **coverability tree** is a tree where the arcs denote transitions $t \in T$ and the nodes represent ω-enhanced states of the Petri net.*

*The **root node** of the tree is $\vec{x}_0$. A **terminal node** is a ω-enhanced state in which no transition is enabled. A **duplicate node** is a ω-enhanced state which already exists somewhere else in the coverability tree. An **arc** t connects two nodes $\vec{x}$ and $\vec{y}$ in the tree, if and only if firing of t in state $\vec{x}$ leads to state $\vec{y}$.*

If the Petri net has a finite number of markings, then the coverability tree is coincident with the reachability tree. This can be seen in the example pictured above.

Example 3.10
Consider the Petri net:

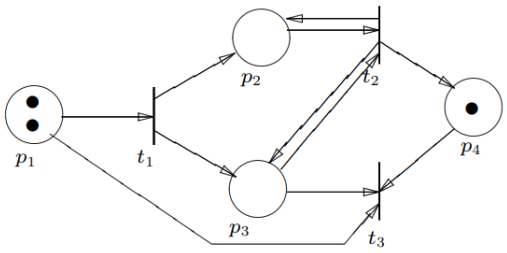


The notion of coverability tree makes sense only in the context of infinite state spaces. In this example, t_1 and t_2 are both L2-live, meaning we can fire them sequentially how many times we want. This results in an accumulation of tokens in p_4, which is then a ω-enhanced place in the coverability tree.

Definition 3.21 (Coverability tree algorithm)
Let $N = (P, T, A, w, \vec{x}_0)$ be a Petri net. Its coverability tree is constructed as follows:

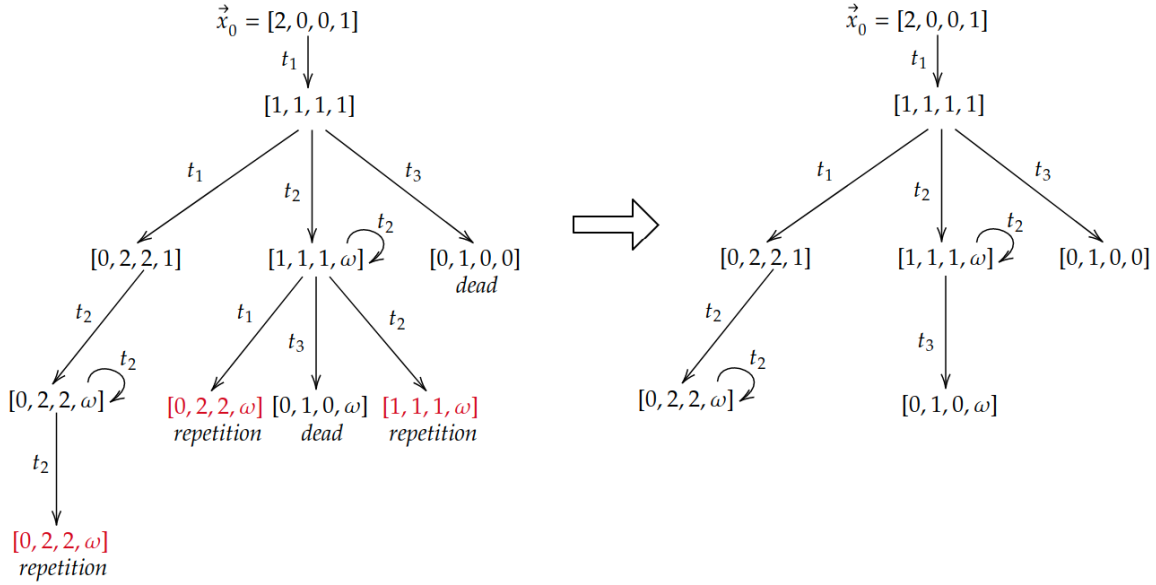
1. *Set L, the list of open nodes, to $L \coloneqq \{\vec{x}_0\}$.*
2. *Take one node from L, called $\vec{x}$, and remove it from L.*
   - *2.1 if $G(\vec{x}, t) = \vec{x}\ \forall t \in T$ then $\vec{x}$ is a terminal node, go to step 3, otherwise continue.*
   - *2.2 for all $\vec{x}' \in G(\vec{x}, t), t \in T, \vec{x} \neq \vec{x}'$:*
     - *2.2.1 do if $x(p) = \omega$ then set $x'(p) = \omega$*
     - *2.2.2 if there is a node $\vec{y}$ already in the tree such that $\vec{x}'$ covers $\vec{y}$, and there is a path from $\vec{y}$ to $\vec{x}'$ then set $x'(p) = \omega$) for all p for which $x'(p) > y(p)$.*
     - *2.2.3 if $\vec{x}'$ is not a duplicate node, then $L \coloneqq L \cup \{\vec{x}'\}$*
3. *if L is not empty, go to step 2.*

Example 3.11
Consider the Petri net:



We start by firing the only enabled transition, t_1. This brings us from state $[2, 0, 0, 1]$ to state $[1, 1, 1, 1]$. Now we have three options. We can fire transition t_3, which brings us to state $[0, 1, 0, 0]$, which is L0-live, i.e. it's dead and no progress can be made, so the tree branch stops here. We can also fire transition t_1 which brings us to state $[0, 2, 2, 1]$. From here we can only fire transition t_2, which would bring us to state $[0, 2, 2, 2]$, but since this state strictly covers one to which it is connected by an arc, we replace the last 2 with an ω. We could fire again t_2, but that would bring us to state $[0, 2, 2, 3]$, which would become for the same reason state $[0, 2, 2, \omega]$, which is a repetition and as such doesn't make up the coverability tree. Returning to state $[1, 1, 1, 1]$, we can finally fire transition t_2, which brings us to state $[1, 1, 1, 2]$. For the same considerations as before (it strictly covers a previous connected state), we substitute an ω in the last place getting state $[1, 1, 1, \omega]$. From here we can either fire t_1, which results in a repetition with state $[0, 2, 2, \omega]$, or fire t_2, which results in a repetition $[1, 1, 1, \omega]$ as well. Finally we can fire t_3, which brings us to state $[0, 1, 0, \omega]$, from which no further transition can be fired (it is dead). Having explored all branches we have finished constructing the coverability tree.

The resulting coverability tree is the following (the markings in red are repetitions which don't make up the tree, but are reported for completeness):



Where there are repetitions that lead from a node to the same node we put a self loop marked with the transition that causes the repetition. If the repetition is of a node seen somewhere else in the tree no self loop is added. Dead nodes are not further marked. The final tree is represented on the right.

With the notion of coverability tree we can define how the properties discussed before (boundedness, safeness, conservation, etc.) are satisfied.

Definition 3.22 (k-boundedness)
A Petri net is k-bounded if the ω symbol never appears in its coverability tree.

If the ω symbol appears, a transition cycle to exceed a given k-bound can always be determined. The coverability tree does not however carry information about the number of cycles required.

Checking for conservation requires setting the weights of the coordinates corresponding to a place that presents a ω symbol in the coverability tree to zero. This is mandatory since the presence of an infinite quantity would make it impossible to apply the rule for conservation we have seen earlier.

Example 3.12
Consider the coverability tree obtained in example 3.11. Is the net conservative for the weighting vector $\vec{\gamma} = [2,3,1,0]$?

The net is not conservative. In fact if we apply the rule to the first node we get: $2*2+3*0+1*0+1*0=4$, but if we apply it for the second node we get: $2*1+3*1+1*1+0*1=6$, therefore the net is not conservative.

A question arises: is it possible to determine a weighting vector that makes the net conservative? The answer is yes, and it requires solving a linear system of equations.

Definition 3.23 (Conservation vector)
*The **conservation vector** is computed as follows:*

- *Set $\gamma_i = 0$ for every unbounded place p_i.*
- *For b bounded places and r nodes in the coverability tree we set up r equations with $b+1$ unknown variables:*

$$\sum_{i=1}^{r} \gamma_i x(p_i) = C$$

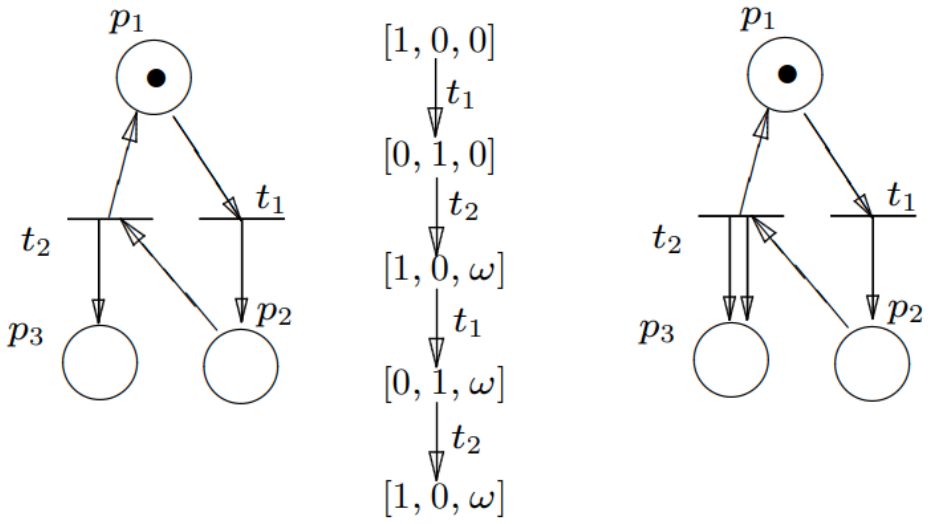
where C is the conservation sum (i.e. the sum of the amount of tokens in any given marking).

Example 3.13
Considering the coverability tree of example 3.11, we can setup four equations as follows:

$$\begin{aligned} \gamma_4 &= 0 \\ 2\gamma_1 + 0\gamma_2 + 0\gamma_3 &= C \\ 1\gamma_1 + 1\gamma_2 + 1\gamma_3 &= C \\ 0\gamma_1 + 2\gamma_2 + 2\gamma_3 &= C \\ 0\gamma_1 + 1\gamma_2 + 0\gamma_3 &= C \end{aligned}$$

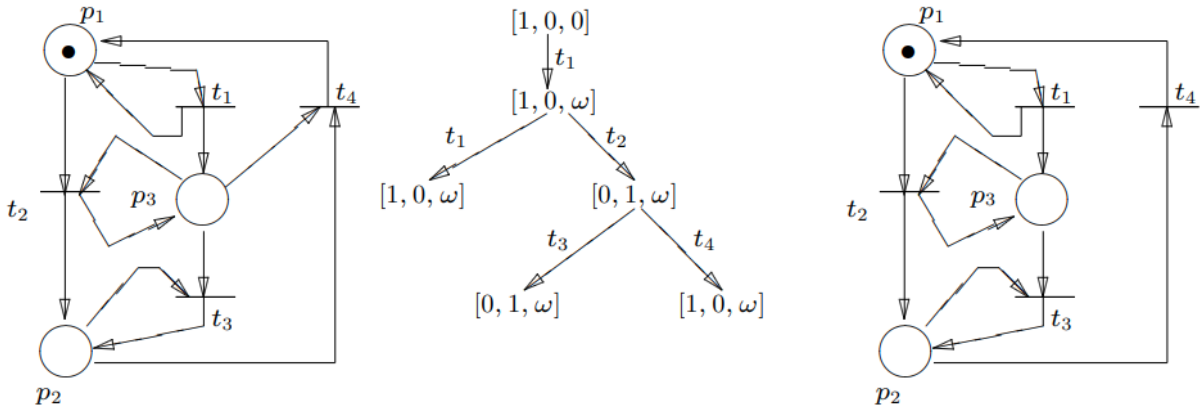
Note that γ_4 is always zero since in p_4 we have an ω symbol.
The only non-negative solution is the trivial one: $\vec{\gamma} = [0, 0, 0, 0]$ and $C = 0$.

The coverability tree is not unique to the net. Consider the following two Petri nets which produce the same coverability tree:



What changes between the two nets is that p_3 is incremented by one token by the left net, but by two tokens by the right net. This shows how the notion of coverability doesn't carry information on *how* a ω-enhanced place is filled with tokens.

Another property that is not described with the coverability tree is deadlocking. Consider the following two nets:



The nets produce the same coverability tree, but the left one can deadlock after t_1, t_2, t_3, while the right one cannot deadlock.

3.3 Decidability, languages, extensions

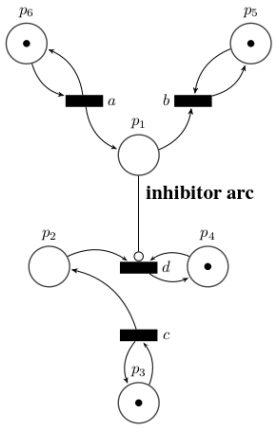
In summary, the main properties of a Petri net are the following:

- Reachability: Given a Petri net, an initial marking M_0 and a marking M, is M reachable from M_0, i.e. is there a sequence of transitions that brings the net from M_0 to M?
- Liveness and deadlocking: For all transitions t and for all markings $M \in R(N, M_0)$, does there exist a marking $M' \in R(N, M)$ in which t is enabled? If there exists a marking $M \in R(N, M_0)$ in which no transition is enabled then the net is deadlocked.
- Boundedness: is there a majorant $k \geq 0$ such that in all places of all reachable markings there can be at most k tokens?
- Conservativity: Is there a weighting vector x such that for every reachable marking the weighted sum of tokens stays the same?
- Repeatability: Is there a transition sequence that can be repeated an infinite number of times?
- Reversibility: Does $M_0 \in R(N, M)$ hold for all markings $M \in R(N, M_0)$?

The answer to the reachability question is that the problem is semi-decidable at first glance, but it turns out that it is decidable in a non-elementary way, meaning it is decidable with a difficult and computationally heavy procedure. We have seen semi-decidability with the coverability tree, which gives us a necessary but not sufficient condition for reachability. Liveness and deadlocking can be determined by reachability. The problem of reversibility can be traced back to the reachability of a particular marking.

Boundedness and conservativity are both decidable with the coverability tree, which gives us a necessary and sufficient condition.

Until now we didn't associate languages to Petri nets. To do so is simply a matter of assigning symbols of a given language to the transitions of the net, which sequences of firings then become sequences of words in the language. Consider for example the Petri net:



The language is made up of four symbols: $L = \{a, b, c, d\}$. c can fire indefinitely, accumulating tokens in p_2. d needs p_1 to have zero tokens in order to fire, and therefore is called **inhibitor arc**. To have zero tokens in p_1, there must be the same number of firings of a and b. If it is enabled, d can fire as many times as c has fired.

The question now is where Petri nets place in the Chomsky hierarchy[1]. Since we have associated Petri nets and languages, from now on we can talk about **Petri languages**. To help us place Petri nets in the Chomsky hierarchy there are a number of theorems that define the properties of Petri languages.

Theorem 3.1 (Regular language)
A regular language is a Petri language.

This means that finite automata are a subclass of Petri nets, since the languages associated to finite automata, which are regular languages, are also Petri languages.

Theorem 3.2 (Context-free language)
The language $\{a^n b^n \mid n > 1\}$ *is both a Petri language and a context-free language, but it is not a regular language.*

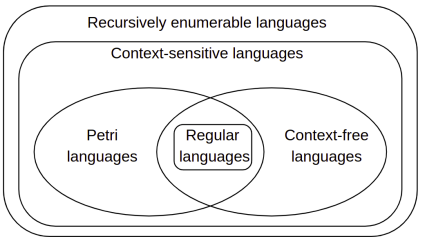
[1] https://en.wikipedia.org/wiki/Chomsky_hierarchy

Theorem 3.3 (Context-sensitive language)
The language $\{a^n b^n c^n \mid n > 1\}$ *is both a Petri language and a context-sensitive language, but it is not a context-free language.*

Theorem 3.4 (Context-free language)
The context-free language $\{ww^R \mid w \in \sum^*\}$ *is not a Petri language.*

Theorem 3.5 (Context-sensitive language)
A Petri language is a context-sensitive language.

In summary the relations between languages and the automata that accept them are:



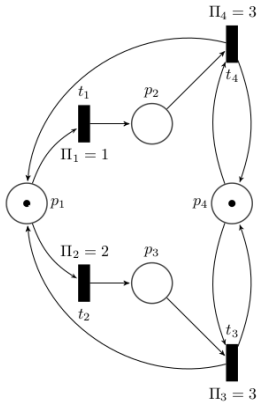
Language	Accepted by
regular	finite automaton
context-free	pushdown ND automaton
Petri	Petri net
context-sensitive	linear bounded automaton
recursively enumerable	Turing machine inhibitor arc-extended Petri net

So Petri languages are a subset of context-sensitive languages, and they partially contain context-free languages and fully contain regular languages. Petri languages can be made equivalent to recursively enumerable languages by introducing the new semantic rule of the inhibitor arcs.

Definition 3.24 (Inhibitor arc)
An ***inhibitor arc*** *is a special arc, denoted with a bubble instead of an arrow, that introduces the concept that a transition* t *can fire only if the origin of the arc is empty.*

The inhibitor arc concept therefore extends the scope of Petri nets, which become more expressive. A "standard" Petri net in fact cannot simulate this "zero-checking" behavior. A Petri net with inhibitor arcs is Turing-equivalent.

Another model that extends the expressiveness of Petri nets is **priority**, i.e. the assignation of priorities to the transitions of the net, which then can only fire in a fixed order.



A priority Petri net can always be redesigned as a inhibitor arc Petri net and vice versa.

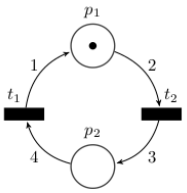
3.4 Matricial representation of Petri nets

The graphic and semi-algorithmic way of analyzing Petri nets is good for understanding the basic topics and for analyzing small simple nets, but it is not a good general approach. We introduce a matricial representation of Petri nets which will allow us to more formally study their properties.

Definition 3.25 (Input and output matrices)
*The **input matrix** I and the **output matrix** O for a Petri net are defined as follows:*

$$I_{|P|,|T|} \quad with\ I(k,j) = w(p_k,t_j), \forall (p_k,t_j) \in P \times T \quad while\ I(k,j) = 0 \forall (p_k,t_j) \notin P \times T$$
$$O_{|P|,|T|} \quad with\ O(k,j) = w(t_j,p_k), \forall (t_j,p_k) \in T \times P \quad while\ O(k,j) = 0 \forall (t_j,p_k) \notin T \times P$$

For example, consider the Petri net:



Its input and output matrices are:

$$I = \begin{array}{c} \begin{matrix} t_1 & t_2 \\ \downarrow & \downarrow \end{matrix} \\ \begin{bmatrix} 0 & 2 \\ 4 & 0 \end{bmatrix} \end{array} \begin{array}{l} \\ \leftarrow\ p_1 \\ \leftarrow\ p_2 \end{array} \qquad O = \begin{array}{c} \begin{matrix} t_1 & t_2 \\ \downarrow & \downarrow \end{matrix} \\ \begin{bmatrix} 1 & 0 \\ 0 & 3 \end{bmatrix} \end{array} \begin{array}{l} \\ \leftarrow\ p_1 \\ \leftarrow\ p_2 \end{array}$$

Definition 3.26 (Incidence matrix)
*The **incidence matrix** C of a Petri net is defined as:*

$$C = O - I$$

where O and I are respectively the output and input matrices of the net.

For the previous example, the incidence matrix of the net would be:

$$C = O - I = \begin{bmatrix} 1 & 0 \\ 0 & 3 \end{bmatrix} - \begin{bmatrix} 0 & 2 \\ 4 & 0 \end{bmatrix} = \begin{bmatrix} 1 & -2 \\ -4 & 3 \end{bmatrix}$$

Definition 3.27 (Marking)
*A **marking** M is a function that assigns to each place p the number of tokens $M(p)$ that it contains. It represents the current state of the Petri net.*

Definition 3.28 (Marking vector)
*Given a Petri net with marking M, its **marking vector** is the column vector of dimension $|P|$ whose components are non negative integers representing the amount of tokens in each place of the net:*

$$m = \begin{bmatrix} m_1 \\ m_2 \\ \vdots \\ m_{|P|} \end{bmatrix} \quad with\ m_i = M(p_i), \forall i = 1,2,\ldots,|P|$$

Definition 3.29 (Firing sequence)
*A **firing sequence** $S = t_1 t_2 \ldots t_n$ enabled in a marking M_0 is a sequence of transitions $t_i, 1 \leq i \leq n, n \in \mathbb{N} - \{0\}$, such that t_1 is enabled in M_0 and the firing of t_i leads to a marking in which t_{i+1} is enabled.*

The notation used for firing sequences is:

$$M_1[t_1 > M_2, M_2[t_2 > M_3 \implies M_1[t_1 t_2 > M_3$$

Or:

$$M_1[S > M_3 \quad with\ S = t_1 t_2$$

Definition 3.30 (Firing vector)
*The **firing vector** s associated to a firing sequence S is a column vector of dimension $|T|$, whose s_i component is equal to the number of occurrences of transition t_i in the S sequence.*

For example, given a Petri net with transitions t_1, t_2, t_3, t_4 and t_5, if $S = t_1 t_1 t_2 t_3$ then $s = \begin{bmatrix} 2 & 1 & 1 & 0 & 0 \end{bmatrix}^T$.

Given the Petri net (N, M_0) with incidence matrix C, suppose it is possible to apply some firing sequence S with firing vector s. If M is the marking reached after applying S, i.e. $M_0[S > M$, we observe that:

$$M = M_0 + Cs$$

The equation above is called **state equation**. The equation expresses the linearity in the evolution of Petri nets. In fact consider the firing sequence $S = t_{j_1} t_{j_2} \dots t_{j_r}$ and let $M_0[S > M$ with $M_0[t_{j_1} > M_1[t_{j_2} > M_2 \dots [t_{j_r} > M_r = M$. Then we have:

$$M = M_r = M_{r-1} + C(\cdot, t_{jr}) = M_{r-2} + C(\cdot, t_{j_{r-1}} + C(\cdot, t_{j_r}) = \dots = M_0 + \sum_{k=1}^{r} C(\cdot, t_{j_k}) = M_0 + Cs$$

where $C(\cdot, t_{j_k})$ is the j_k-th column of C, whose i-th component represents the number of tokens lost or gained by place p_i with the firing of t_{j_k}.

The firing of a transition t_{j_k} from a marking M_0 has the effect of changing the vector M_0, to which are summed algebrically the values of column $C(\cdot, t_{j_k})$, so the firing of multiple transitions has the cumulative effect of summing to M_0 all the column vectors of C corresponding to the individual firings, which is equivalent to summing to M_0 the matrix-column product Cs.

We can observe that:

- The incidence matrix C has as many rows as there are places and as many columns as there are transitions.
- The firing vector s has as many rows as there are transitions and a single column, and its j_k-th component represents how many times transition t_{j_k} is fired in the firing sequence S.
- Cs has as many rows as there are places and a single column.

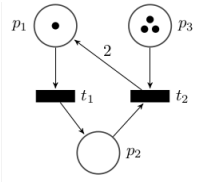
Definition 3.31 (P-invariant)
*A **P-invariant** of a net N is a column vector x of dimension $|P|$ such that $\forall M \in R(N, M_0)$:*

$$x^T M = x^T M_0$$

P-invariants (which stand for Place-invariants), represent sets of places such that the algebric weighted sum of tokens contained in the places remains constant in all reachable markings of the net.

From $M = M_0 + Cs$ we get that $x^T M = x^T M_0 + x^T Cs$ for all enabled firing vectors $s \neq 0$, therefore $x^T M_0 + x^T Cs = x^T M_0$ for all $s \neq 0$. Therefore a P-invariant x must satisfy $x^T Cs = 0$ for all $s \neq 0$, so P-invariants can be computed from the integer solutions of $x^T C = 0$ or $C^T x = 0$.

For example, consider the Petri net:



Its incidence matrix is:

$$C = O - I = \begin{bmatrix} 0 & 2 \\ 1 & 0 \\ 0 & 0 \end{bmatrix} - \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} -1 & 2 \\ 1 & -1 \\ 0 & -1 \end{bmatrix}$$

P-invariants are computed by solving:

$$x^T C = \begin{bmatrix} 0 & 0 \end{bmatrix} \implies \begin{bmatrix} a & b & c \end{bmatrix} \begin{bmatrix} -1 & 2 \\ 1 & -1 \\ 0 & -1 \end{bmatrix} = \begin{bmatrix} 0 & 0 \end{bmatrix} \implies a = 1, b = 1, c = 1 \implies x = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

Note that $x^T C = 0$ has infinite solutions, since the linear combination of P-invariants is a P-invariant. A strategy then is to search for the smallest set of P-invariants that generates all possible solutions.

Definition 3.32 (Support of a P-invariant)
*The **support** of a P-invariant x is the set, denoted with $\|x\|$, of places corresponding to non-null elements of x.*

Definition 3.33 (Minimal support P-invariant)
*A P-invariant is defined as having **minimal support** if its support does not contain the support of any of the other P-invariants of the net.*

Definition 3.34 (Canonical P-invariant)
*A P-invariant is called **canonical** if the maximum common divisor of its non-null elements is 1.*

Definition 3.35 (Positive P-invariant generator set)
*A **positive P-invariant generator set** is the smallest set of positive P-invariants $PI_k, 1 \leq k \leq q$, such that every other P-invariant of the net is obtainable as a linear combination of the PI_k invariants. The elements of the generator sets are called minimal P-invariants.*

Observation 3.2 (Minimal P-invariant)
*A P-invariant is **minimal**, i.e. it belongs to the positive P-invariant generator set, if and only if it is canonical and with minimal support.*

Observation 3.3 (Finiteness and uniqueness)
The P-invariant generator set is finite and unique.

Definition 3.36 (P-invariant covered net)
*A net is **covered by P-invariants** if every place in the net belongs to the support of at least one P-invariant.*

Definition 3.37 (Conservative net)
A non-negative P-invariant covered net, i.e. a net such that:

$$\forall p \in P, \exists x \mid p \in \|x\|, x(p) > 0$$

*is **conservative**.*

Observation 3.4 (Boundedness)
A conservative net is bounded. A bounded net is not necessarily conservative (e.g. a place p with an edge towards a transition t with markings $\{[1],[0]\}$,, for all positive integer x we have $0 = x, 0 \neq x, 1 = x$).

We can compute the minimal P-invariants with the following algorithm:

1. $A = C, Y = I$
2. $P = \begin{bmatrix} Y & A \end{bmatrix}$
3. for $i = 1$ to m do:
   (a) Add to P all rows linear combination with positive coefficients of couples of rows of P, who nullify the i-th column of the A part of the P matrix.
   (b) Prune from P the rows whose i-th column in the A part of the P matrix is not null.
4. The rows in the Y part of the P matrix are the positive P-invariants of the net. Among them are the minimal P-invariants.

Definition 3.38 (T-invariant)
*A **T-invariant** of a net N is a column vector y of dimension $|T|$ solution of the following equation:*

$$Cy = 0$$

Dually to the P-invariants, T-invariants are related to transitions. They represent the possible sequences of firings that bring the net to its initial marking.

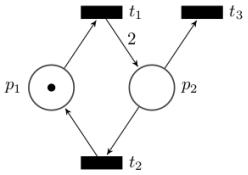
Definition 3.39 (Repetitive firing sequence)
*A firing sequence S is **repetitive** from a marking M in which it is enabled if it can be executed an infinite amount of times from M. A repetitive sequence S enabled from M is called **stationary** if $M[S > M$.*

Given a net N, let S be a firing sequence enabled from an initial marking M_0, let s be the corresponding firing vector and let $M_0[S > M$. The vector s is a T-invariant if and only if $M = M_0$, i.e. if the sequence S is repetitive and stationary.

Observation 3.5
The existence of a T-invariant does not imply the possibility of returning to an initial marking, because there may not exist an enabled firing sequence whose firing vector coincides with the given T-invariant. In other words, the existence of a T-invariant indicates that, if a firing sequence compatible with the T-invariant existed, then the marking of the net would return to the initial marking.

Definition 3.40 (Duality principle)
*The **duality principle of Petri nets** states that the T-invariants of a net with incidence matrix C are the P-invariants of the dual net with incidence matrix C^T. The dual net is obtained by switching places and transitions, and by inverting the direction of edges between them.*

For example, consider the net:



its incidence matrix is:

$$C = O - I = \begin{bmatrix} 0 & 1 & 0 \\ 2 & 0 & 0 \end{bmatrix} - \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 1 \end{bmatrix} = \begin{bmatrix} -1 & 1 & 0 \\ 2 & -1 & -1 \end{bmatrix}$$

To find a T-invariant we solve:

$$Cy = \begin{bmatrix} -1 & 1 & 0 \\ 2 & -1 & -1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \implies a = 1, b = 1, c = 1 \implies y = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}$$

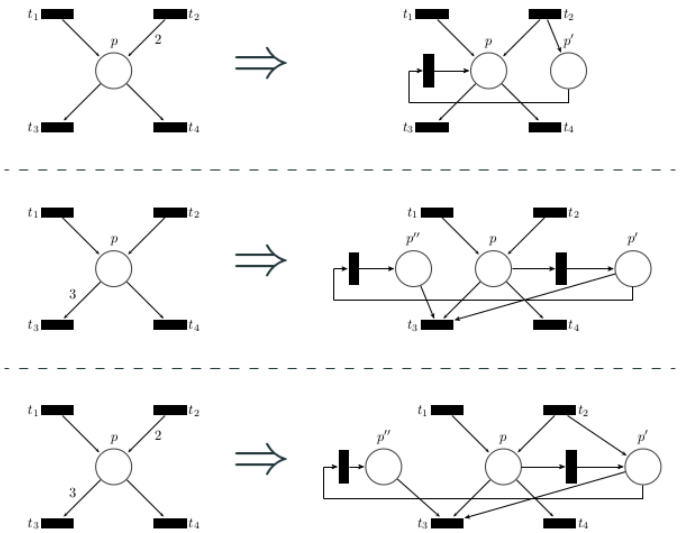
3.5 Classification of Petri nets

Definition 3.41 (Ordinary net)
*A Petri net is **ordinary** if wvery arc has weight 1.*

Definition 3.42 (Pure net)
*A Petri net is **pure** if there are no self-loops, i.e. no oriented cycles composed by only one transition and only one place.*

Definition 3.43 (Restricted net)
*A Petri net is **restricted** if it is ordinary and pure.*

Restricted Petri nets are not less expressive than general nets, but they can be less efficient in the dimension of the representation. Any general net can be transformed in a restricted net with elementary transformations:

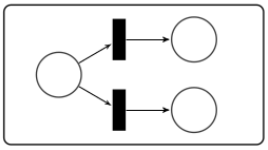


Definition 3.44 (Pre-set and post-set of a place)
*${}^\bullet p, p^\bullet$ are respectively the set of input transitions, or **pre-set**, and the set of output transitions, or **post-set**, of place p.*

Definition 3.45 (Pre-set and post-set of a transition)
*${}^\bullet t, t^\bullet$ are respectively the set of input places, or **pre-set**, and the set of output places, or **post-set**, of transition t.*

Definition 3.46 (State machine or S-graph)
*A **state machine** is a Petri net such that $\forall t_j \in T, |{}^\bullet t_j| = 1$ and $\left|t_j^\bullet\right| = 1$. In other words every transition has exactly one input and one output.*

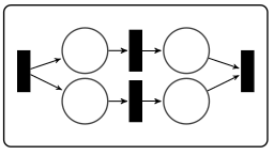
Note that "state machine" here is a new concept, different from what we have seen before.
A state machine can only model choice or conflict. For example conflict is modeled as:



Therefore state machines are less expressive than general Petri nets. If there are multiple tokens the net could represent a limited form of parallelism, but synchronization is not representable since the enabling of a given transition depends on a single place. A state machine with only one token is equivalent to a finite state automaton. They are not the only subclass of Petri nets that are equivalent to finite state automatons, since every Petri net with a finite state space corresponds to a finite state automaton which generates or interprets the same language.

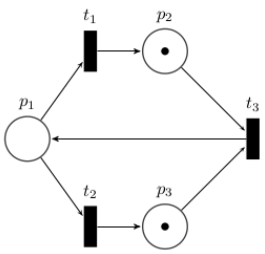
Definition 3.47 (Signal Transition Graph or T-Graph)
*A **signal transition graph (STG)** is a Petri net such that $\forall p_i \in P, |{}^\bullet p| = 1$ and $|p^\bullet| = 1$. In other words every place has exactly one input and one output.*

STGs are the duals of state machines. As such they cannot implement choice or conflict, since every place has only one output transition) but they can implement parallelism, concurrency and synchronization, since the enabling of a transition can depend on multiple places. They are still less expressive than general nets.
For example, the following net implements synchronization of parallel token fluxes:

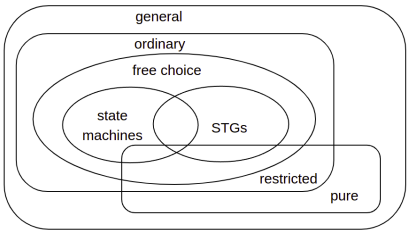


Definition 3.48 (Free choice net)
*A **free choice net** is a Petri net such that $\forall t_j \in T, p_i \in {}^\bullet t_j$ it either has ${}^\bullet t_j = \{p_i\}$ or $p_i^\bullet = \{t_j\}$. In other words for each arc from a place to a transition $(p_i \to t_j)$, either the place is the only input for the transition (no synchronization) or the transition is the only output for the place (no conflict).*

If a place is the input of multiple transitions, it is also the only input place for all those transitions, therefore either all conflicting transitions are enabled or non is. As a consequence, any transition can fire arbitrarily independently from the current marking, hence the name "free choice".
A free choice net can implement choice or parallelism, but not both at the same time. Therefore they are still less expressive than general nets. Furthermore each state machine or STGs a free choice net, but not all free choice nets are either state machines or STGs.
For example, the following net is a free choice net but neither a state machine nor an STG:



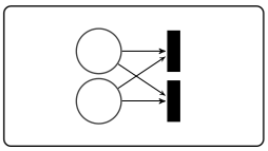
In summary, the relations between each subclass of Petri nets is:



Definition 3.49 (Extended free choice net)
*An **extended free choice net** is a Petri net such that* $\forall t_j \in T, p_i \in {}^\bullet t_j$*, there is an arc from all input places of* t_j *to all output transitions of* p_i*.*

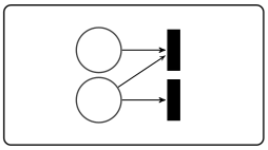
Observation 3.6
If two places share an output transition, they also share all of their output transitions.

Extended free choice nets can implement symmetric choice, conflict and flux synchronization. For example, the following net models symmetric choice:



Definition 3.50 (Asymmetric choice net)
*An **asymmetric choice net** is a Petri net such that* $p_i^\bullet \cap p_j^\bullet \neq \varnothing \implies p_i^\bullet \subseteq p_j^\bullet$ *or* $p_i^\bullet \supseteq p_j^\bullet$*.*

Asymmetric choice nets can implement, as the name suggests, asymmetric choice:



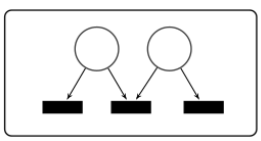
They are therefore more expressive than free choice nets.

Free choice nets and extended free choice nets both satisfy the property that, if t_1 and t_2 share an input place, then no markings in which a transition is enabled and the other is not exists. In other words there is free choice on which transition can fire. An extended choice net can be transformed into a free choice net.

In summary, all subclasses of nets just described cannot implement:

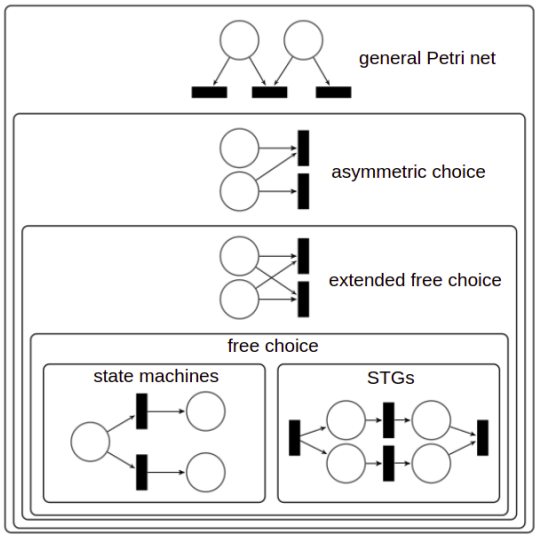
- State machines: synchronization
- STGs: conflict
- Free choice nets: confusion
- Asymmetric nets: symmetric confusion

Confusion refers to the situation in which the choice of firing a transition influences the possibility of firing other transitions:



Only general Petri nets can implement confusion (asymmetrically and symmetrically).

In summary, the relations between the expressiveness of each class of net is:



3.6 S-component and T-component decomposition

The idea is to study whether or not we can separate a Petri net in useful subnets, in order to study Petri nets as compositions of these subnets.

Definition 3.51 (Net)
*A **net** $N = (P, T, A)$ is a triple of places P, transitions T and arcs $A \subseteq P \times T \cup T \times P$.*

Definition 3.52 (Subnet)
*The triple $N' = (P', T', A')$ is a **subnet** of the net $N = (P, T, A)$ if and only if $P' \subseteq P$, $T' \subseteq T$ and $A' = A \cap ((P' \times T') \cup (T' \times P'))$.*

Definition 3.53 (Subnet generation)
*A subnet N' is **generated** by the set $X' = P' \cup T' \subseteq P \cup T$ if and only if $A' = A \cap ((P' \times T') \cup (T' \times P'))$.*

Intuitively, we can define subnets generated by places (transitions) of a state machine (STG).

Definition 3.54 (S-component (T-component))
*A subnet N' is a **S-component (T-component)** of the net N if and only if N' is a strongly connected S-graph (T-Graph).*

Definition 3.55 (Bounded net)
*A Petri net (N, M_0) is **bounded** if and only if $\forall p \in S, \exists k \in N$ such that $\forall M \in R(N, M_0), M(p) \leq k$. If $k = 1$ the net is **safe**.*

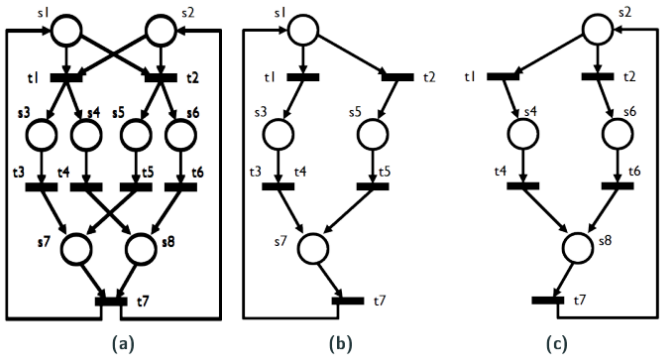
Definition 3.56 (Structurally bounded net)
*A Petri net N is **structurally bounded** if and only if it is bounded for each initial marking M_0.*

Definition 3.57 (Liveness)
*A transition t is **live** in (N, M_0) if and only if $\forall M \in R(N, M_0), \exists M' \in R(N, M)$ such that t is enabled in M. A Petri net (N, M_0) is **live** if and only if all of its transitions are live. A Petri net (N, M_0) is **structurally live** if and only if there exists only one initial marking M_0 such that (N, M_0) is live.*

Definition 3.58 (Well formed net)
*A Petri net N is **well formed** if there exists a marking M_0 of N such that (N, M_0) is live and safe.*

Theorem 3.6 (S-components (T-components) and well formed nets)
A well formed Petri net N can be covered with S-components (T-components).

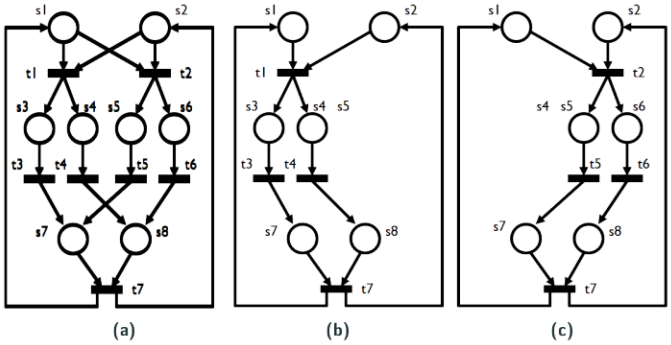
The meaning of covering a net with S-components is evident in the following example:



Both (b) and (c) are S-graphs (i.e. state machines).

The original net (a) is separated into two subnets (b) and (c) which cover, up to repetition of transitions, all of its places and transitions. The usefulness of this decomposition is obviously that it is simpler to study smaller subnets than bigger and more complicated nets.

The same can be done with T-components



Both (b) and (c) are T-graphs (i.e. STGs).

3.7 Siphons and traps

Siphons and traps are constructs that help us define some properties of free choice Petri nets.

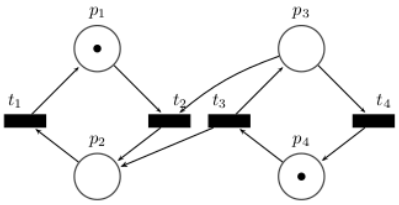
Definition 3.59 (Siphon)
*A set of places S is a **siphon** if and only if ${}^\bullet S \subseteq S^\bullet$.*

Definition 3.60 (Minimal siphon)
*A siphon S is **minimal** if and only if there exist no other siphon S' such that $S' \subseteq S$.*

Definition 3.61 (Base siphon)
*The union of siphons is a siphon. A **base siphon** is a siphon that cannot be obtained as a union of other siphons.*

A siphon is a set of places that generates a subset of the tokens it consumes, therefore if it empties it cannot acquire tokens.

For example:



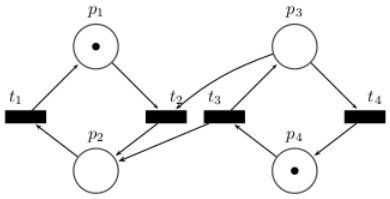
$S = \{p_3, p_4\}$ is both a minimal and a base siphon. Both $S' = \{p_2, p_3, p_4\}$ and the set of all places of the net are siphons: ${}^\bullet S' = \{t_2, t_3, t_4\} \subseteq S'^\bullet = \{t_1, t_2, t_3, t_4\}$.

Definition 3.62 (Trap)
*A set of places S is a **trap** if and only if $S^\bullet \subseteq {}^\bullet S$.*

Observation 3.7
The union of traps (siphons) is a trap (siphon).

A trap is the dual object to a siphon. It is a set of places that consumes a subset of the tokens it generates, and therefore once it acquires at least one token it cannot empty.

For example:



$S = \{p_1, p_2\}$ is a trap: $S^\bullet = \{t_1, t_2\} \subseteq {}^\bullet S = \{t_1, t_2, t_3\}$. $S' = \{p_1, p_2, p_3\}$ is a trap: $S^\bullet = \{t_3, t_2, t_1\} \subseteq {}^\bullet S = \{t_4, t_2, t_1, t_3\}$. $S" = \{p_1, p_2, p_3, p_4\}$ is a trap: $S"^\bullet = {}^\bullet S" = \{t_4, t_3, t_2, t_1\}$.

Siphons and traps can be present simultaneously in Petri nets.

Using these constructs we can define a series of theorems about the properties of free choice Petri nets.

Theorem 3.7 (Commoner's theorem)
A free choice Petri net (N, M_0) is live if and only if each siphon contains a marked (i.e. containing tokens) trap.

Theorem 3.8
A live free choice Petri net (N, M_0) is safe if and only if it is covered with strongly connected S-components, each of which has exactly one token.

Theorem 3.9
Let (N, M_0) be a live, safe free choice Petri net. Then N is covered by strongly connected T-components. Furthermore there exists a marking $M \in R(N, M_0)$ such that each component (N_1, M_1) is a marked graph that is both live and safe.

Theorem 3.10
An asymmetric choice Petri net (N, M_0) is live if each siphon contains a marked trap.

Some of these results are collected in a single theorem that links the liveness and safeness of a free choice Petri net to the possibility of decomposing it in interconnected S-components and T-components:

Theorem 3.11 (Hack's theorem)
A free choice Petri net N is well formed (i.e. there exists a marking M_0 of N such that (N, M_0) is safe and live) if and only if:

- *every T-component is strongly connected and not empty and the set of T-components covers the net.*
- *every S-component is strongly connected and not empty and the set of S-components covers the net.*

3.8 Further extensions of Petri nets

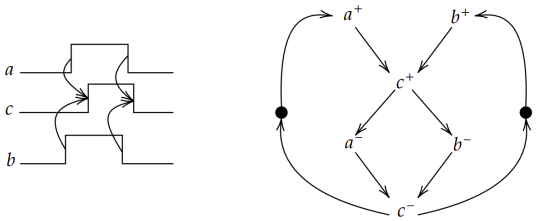
In general Petri nets the tokens are indistinguishable from one another, i.e. they carry no information. In high-level nets we can associate information to each token, in order to expand the capabilities of representation of the net. For example we can assign colors to tokens, which then become carriers of information that is interpreted by transitions, to which we need therefore to assign logical conditions that cause their firing.

Other extensions to the theory include hierarchic Petri nets, in which transitions are replaced by entire Petri nets, temporized Petri nets, which introduce timing in the firing of transitions, and stochastic Petri nets, which introduce delays in the transitions based on probabilistic distributions.

3.9 Signal transition graph

A **signal transition graph (STG)** is a formalism utilized in asynchronous circuits to analyze and synthesize them.

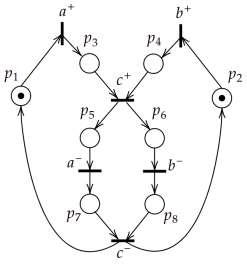
Example 3.14
Consider the C-element, which is an asynchronous circuit that, given two input signals a and b, has an output signal c that goes up when both inputs go up, and goes down when both inputs go down:

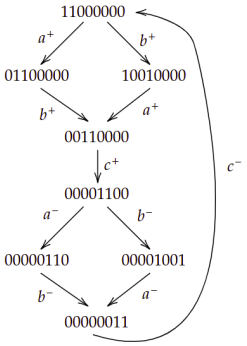


The signal transition graph for the circuit is constructed by placing the signals behaviors (up and down) as nodes, and connecting the nodes with edges that represent the causal link between signal actions. Then special nodes represented by black dots are added, which represent the starting condition of the system.

Looking at the STG on the right, we can see that, starting from the initial state, we go to the a^+ and b^+ nodes, which represent the signals a and b going up. After those nodes we go to the c^+ node, which represents the signal c going up. Then we repeat the same graph but for the signals going down, and finally we return to the initial condition.

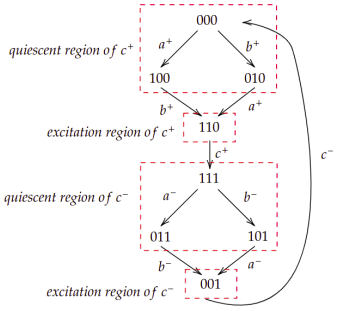
We can transform the STG into a Petri net, by replacing the nodes that represent signal actions with transitions, the starting nodes with places containing a token, and the edges between signal action nodes with a place. The corresponding Petri net is:





We can then build the reachability graph for the Petri net, which in the field of STGs is called **transition system**. The result is shown on the right.

We could then encode the states in the transition system with 3 variables, representing whether or not a signal is high or low, obtaining:



The **quiescent region** of a signal is a region on the encoded transition system where its value remains constant. The **excitation region** of a signal is a region on the encoded transition system where the signal changes its value.

Finally we could use Karnaugh maps using the encoded signals to design a logic circuit that implements the C-element, but this is outside the scope of the course.

We now define formally a transition system.

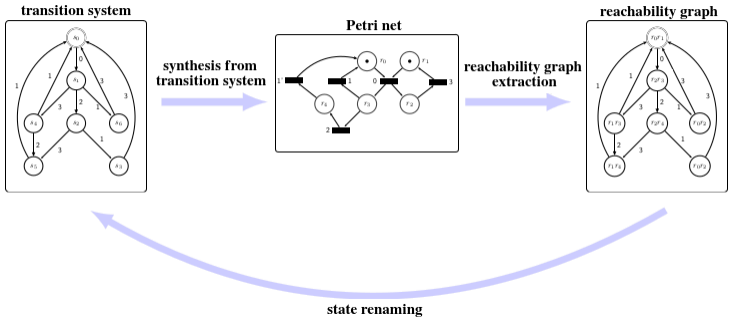
Definition 3.63 (Transition system)
*A **transition system** TS is defined as a 4-tuple $TS = (S, E, T, s_0)$ where:*

- *S is a non-empty set of states.*
- *E is a set of events.*
- *$T \subseteq S \times E \times S$ is a **transition relation**.*
- *s_0 is the initial state.*

Observation 3.8 (Properties of a transition system)
We assume that the transition system satisfies the following properties:

- *There are no self-loops: $\forall (s, e, s') \in TS \neq s'$*
- *Each event occurs at leas once: $\forall e \in E \exists \mid (s, e, s') \in T$*
- *Each state is reachable from the initial state: $\forall s \in S \mid s_0 \rightarrow {}^* s$*

We study transition systems because they are related to Petri nets. In fact transition systems are equivalent to the reachability graph of a Petri net, up the renaming of states.



So now we want to know how to determine a Petri net equivalent to a given transition system. We want to do so because with Petri nets we can intuitively model concurrent systems, and also implement control and validation of those systems more easily because of the limited dimensions of Petri nets compared to transition systems.

Definition 3.64 (Region)
*Given a transition system $TS = (S, E, T, s_0)$ we define **region** a set of states $r \subseteq S$ such that the following properties are satisfied for each event $e \in E$:*

1. $enter(e, r) \rightarrow \neg in(e, r) \wedge \neg out(e, r) \wedge \neg exit(e, r)$
2. $exit(e, r) \rightarrow \neg in(e, r) \wedge \neg out(e, r) \wedge \neg enter(e, r)$

where;

$$no\_cross \begin{cases} in(e, r) \equiv \exists (s, e, s') \in T \mid s, s' \in r \\ out(e, r) \equiv \exists (s, e, s') \in T \mid s, s' \notin r \end{cases}$$
$$enter(e, r) \equiv \exists (s, e, s') \in T \mid s \notin r \wedge s' \in r$$
$$exit(e, r) \equiv \exists (s, e, s') \in T \mid s \in r \wedge s' \notin r$$

A region is therefore a subset of states such that every transition marked with the same event satisfy the property of entering the region ($enter$), exiting it ($exit$) or either being contained in the region or not (no_cross).

Definition 3.65 (Pre-region, post-region)
*A region r is a **pre-region** of an event e if there exists a transition marked with e exiting from r. A region r is a **post-region** of an event e if there exists a transition marked with e entering from r.*

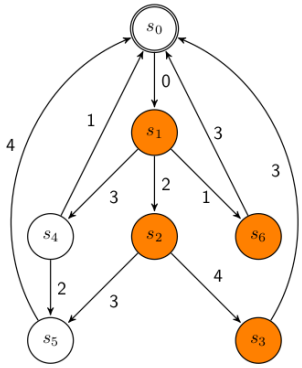
The sets of all pre-regions and post regions of e are denoted with ${}^\circ e$ and e° respectively. By definition, if $r \in {}^\circ e$ ($r \in e^\circ$), all transitions marked with e exit r (enter r).

Definition 3.66 (Sub-region)
*A region r' is a **sub-region** of a region r if $r' \subset r$.*

Definition 3.67 (Minimal region)
*A region r is a **minimal region** if no other region r' is a sub-region of r.*

Observation 3.9 (Region union and intersection)
If r and r' are regions with r' sub-region of r, then $r - r'$ is a region. Any region can be represented with the union of disjointed minimal regions.

For example:

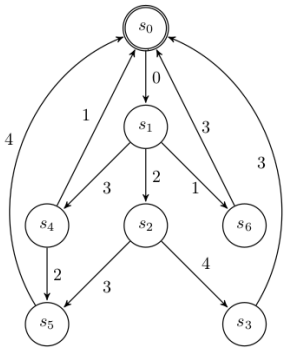


Event	Property
0	enter
1	no_cross
2	no_cross
3	exit
4	no_cross

$\{s_1, s_2, s_3, s_6\}$ is a region, since all the transitions of each event satisfy the same property.

Definition 3.68 (Excitation region)
*An **excitation region** for event e, denoted with $ER(e)$, is a maximal set of states S such that for each $s \in S$ there exists a transition $s \xrightarrow{e}$.*

For example:



Event	Excitation Region
0	$\{s_0\}$
1	$\{s_1, s_4\}$
2	$\{s_1, s_4\}$
3	$\{s_1, s_2, s_3, s_6\}$
4	$\{s_2, s_5\}$

To generate a Petri net from a transition system we first compute the set of minimal pre-regions. We consider only the non-trivial regions, i.e. all regions except the region containing all states and the region containing no states. From the non-trivial region set we then compute the set R_{TS} of minimal pre-regions. We can observe that, if the transition system is strongly connected, all non-trivial regions are also pre-regions.

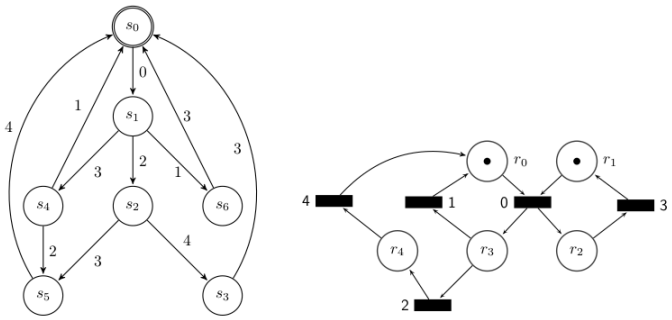
Definition 3.69 (Closure)
*A transition system is **closed with respect to excitation** if the following properties are satisfied:*

1. *Closure with respect to excitation:*
$$\forall a : \cap_{r \in {}^\circ a} r = ER(a)$$
2. *Efficacy of the event:*
$$\forall a : {}^\circ a \neq \varnothing$$

If the transition system is closed with respect to excitation, then an equivalent Petri net having one transition for each of the transition system's events can be obtained.

The Petri net is made up of as many places as there are regions in the transition system, as many transitions as there are events, and edges placed according to the post- and pre-regions of each event.

For example:



Region	States
r_0	$\{s_0, s_3, s_6\}$
r_1	$\{s_0, s_4, s_5\}$
r_2	$\{s_1, s_2, s_3, s_6\}$
r_3	$\{s_1, s_4\}$
r_4	$\{s_2, s_5\}$

Event	Pre-region	Post-region
0	r_0, r_1	r_2, r_3
1	r_3	r_0
2	r_3	r_4
3	r_2	r_1
4	r_4	r_0

If the transition system is not closed with respect to excitation the procedure can still be carried out, but first a step of **label splitting**, i.e. a division of events in copies of themselves in order to align them with the property of a given region. Furthermore some optimization procedures exist, that simplify and reduce the resulting Petri net. Both label splitting and optimization are not subjects of this course.

4 Supervisory control

We introduce a summary of properties of languages and automata, that will help us analyze the problem of designing a formalism called supervisory control.

4.1 Languages

Definition 4.1 (Alphabet)
*An **alphabet** is a finite set of **event** symbols:*

$$E = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$$

Definition 4.2 (Trace)
*A **trace**, or word, is a finite sequence of events from an alphabet E:*

$$s = \sigma_i \sigma_j \sigma_j \sigma_k \ldots$$

*$\sigma_i \in s$ denotes that the event σ_i appears in s. $|s|$ denotes the length of the trace, i.e. the number of events including repetitions that it contains. ϵ denotes the **empty trace**, i.e. the trace with $|\epsilon| = 0$.*

Observation 4.1 (Properties of traces)
Some properties of traces are:

- ***Concatenation** of traces: if $s_2 = \sigma_5\sigma_4$ and $s_1 = \sigma_2\sigma_3\sigma_1\sigma_1\sigma_5$, then $s_1 s_2 = \sigma_2\sigma_3\sigma_1\sigma_1\sigma_5\sigma_5\sigma_4$. ϵ is the identity element for concatenation: $s_1\epsilon = \epsilon s_1 = s_1$. σ^n denotes the repetition of the event σ n times.*
- ***Prefix, suffix** and **subtrace**:*

$$\begin{aligned} \sigma_2\sigma_3\sigma_1 &\quad \text{is a prefix of } s_1 \\ \sigma_1\sigma_5 &\quad \text{is a suffix of } s_1 \\ \sigma_3\sigma_1\sigma_1 &\quad \text{is a subtrace of } s_1 \end{aligned}$$

- ***Prefix-closure** of a trace: it is the set that contains all prefixes of the trace:*

$$\overline{s_2} \coloneqq \overline{\{s_2\}} = \{\epsilon, \sigma_5, s_2\}$$

Definition 4.3 (Kleene closure)
*The **Kleene closure** of the alphabet E, denoted with E^*, is the set of all finite traces of elements of E, including ϵ. Such set is countably infinite.*

Definition 4.4 (Language)
*A **language** over an alphabet E is a subset of the Kleene closure of the alphabet, i.e. any:*

$$L \subseteq E^*$$

is a language. Therefore $\varnothing$, E and E^* are languages.

Observation 4.2
$\epsilon \notin \varnothing$, therefore $\{\epsilon\}$ is a language.

Observation 4.3 (Properties of languages)
Some properties of languages are:

- ***Set operations**: union, intersection, difference (denoted with "\") and complement with respect to E^*.*
- ***Concatenation**: let $L_1, L_2 \subseteq E^*$. Then:*

$$L_1 L_2 \coloneqq \{s \in E^* \mid (s = s_1 s_2) \wedge (s_1 \in L_1) \wedge (s_2 \in L_2)\}$$

- ***Prefix-closure**: let $L \subseteq E^*$. Then:*

$$\overline{L} \coloneqq \{s \in E^* \mid (\exists t \in E^*), st \in L\}$$

*Therefore the prefix closure of L is the language consisting of all the prefixes of all the traces in L. For example if $L = \{abc, cde\}$, then $\overline{L} = \{\epsilon, a, ab, abc, c, cd, cde\}$. If $L = \varnothing$ then $\overline{L} = \varnothing$, and if $L \neq \varnothing$ then $\epsilon \in \overline{L}$. In general $L \subseteq \overline{L}$. L is **prefix-closed** if $L = \overline{L}$.*

- ***Kleene-closure**: let $L \subseteq E^*$. Then:*

$$L^* \coloneqq \{\omega \in E^* \mid \omega = \omega_1\omega_2 \dots \omega_k, k \geq 0, \omega_i \in L\}$$

The * operation is idempotent, i.e. $(L^*)^* = L^*$. Also $\varnothing^* = \{\epsilon\}$ and $\{\epsilon\}^* = \{\epsilon\}$.

- ***Post-language** of L after trace s:*

$$L \setminus s \coloneqq \{t \in E^* \mid st \in L\}$$

By definition, $L \setminus s = \varnothing$ if $s \notin \overline{L}$.

- ***Concatenation with Kleene-closure**:*

$$L^+ \coloneqq LL^*$$

- ***Nonconflicting languages**: L_1 and L_2 are nonconflicting if $\overline{L_1 \cap L_2} = \overline{L}_1 \cap \overline{L}_2$.*
- ***M-closure**: L is M-closed if $\overline{L} \cap M = L$.*

As we have seen, E is finite while E^* is countably infinite. The power set of E^*, i.e. the set of all languages over the alphabet E, has cardinality 2^{E^*}, which is uncountable.

We want to represent languages in a finite way. If a language is finite we can always list all its element, albeit rarely in a practical way. If a language L is infinite, we can represent it as:

$$L = \{s \in E^* \mid s \text{ has property} P\}$$

Where P is some property, like for instance that a trace should have the same number of σ_i events as s_j events. This representation can be useful, but it is not particularly for supervisory control we will need to find subsets and supersets of L.

More preferably we would like to use **discrete event modeling formalism**, such as finite state automata and Petri nets, that require us to specify only a finite number of "objects" in order to represent a particular language. Then we are interested in determining how many languages in 2^{e^*} the formalism can represent, since we cannot represent an uncountable number of languages with a finite number of objects.

4.2 Automata

Definition 4.5 (Deterministic automaton)
*A **deterministic automaton**, or automaton for short, is a six-tuple:*

$$G = (X, E, f, \Gamma, x_0, X_m)$$

where:

- *X is the **set of states**.*
- *E is the **set of events** associated to the transitions in G.*
- *$f : X \times E \to X$ is the **transition function** $f(x, e) = y$, that describes the presence of a transition labeled by event e from state x to state y. In general, f is a partial function on its domain.*
- *$\Gamma : X \to 2^E$ is the **active event function**. $\Gamma(x)$ is the set of all events e for which $f(x, e)$ is defined, and it is called **active event set** of G at x.*
- *x_0 is the **initial state**.*
- *$X_m \subseteq X$ is the set of **marked states**.*

Observation 4.4 (Properties of automatons)
We can observe that:

- *If X is a finite set, we call G a **deterministic finite-state automaton**, or DFA for short.*
- *The automaton is deterministic because f is a function over $X \times E$.*
- *The fact that f is only partially defined over $X \times E$ is a variation over the standard computer science definition of automaton.*
- *The definition of Γ is superfluous, since it derives from f.*
- *The selection of which states to mark is a modeling issue that depends on the problem at hand.*

The automaton G operates as follows. It starts in the initial state x_0 and, upon the occurrence of an event $e \in \Gamma(x_0) \subseteq E$, it will make a transition to state $f(x_0, e) \in X$. This process then continues based on the transitions for which f is defined.

For convenience reasons, f is always extended from domain $X \times E$ to domain $X \times E^*$ in the following way:

$$\begin{aligned} f(x, \epsilon) &\coloneqq x \\ f(x, se) &\coloneqq f(f(x, s), e) \quad \text{for } s \in E^* \text{ and } e \in E \end{aligned}$$

Considering now the automaton as a directed graph, we can consider the subset of directed paths that can be followed from initial state and that lead to a marked state. This leads us to introduce the notions of languages generated and marked by the automaton.

Definition 4.6 (Generated language)
*The language **generated** by G is:*

$$L(G) \coloneqq \{s \in E^* \mid f(x_0, s) \text{ is defined}\}$$

Definition 4.7 (Marked language)
*The language **marked** by G is:*

$$L_m(G) \coloneqq \{s \in L(G) \mid f(x_0, s) \in X_m\}$$

Observation 4.5
$L(G)$ is always prefix-closed. $L(G) = E^*$ when f is a total function (i.e. defined on all $X \times E$.

Definition 4.8 (Automata equivalence)
*Automata G_1 and G_2 are **equivalent** if:*

$$L(G_1) = L(G_2) \quad \text{and} \quad L_m(G_1) = L_m(G_2)$$

So in general G represents two languages, $L(G)$ and $L_m(G)$. In general $L_m(G) \subseteq \overline{L_m(G)} \subseteq L(G)$.

Since we use an automaton to model two languages, we can delete all the states that are not accessible or **reachable** from x_0 by some trace in $L(G)$. When we delete states, we also delete the transitions that are attached to them.

Definition 4.9 (Accessibility)
*A state x is **accessible** if it can be reached from the initial marking x_0 under some trace s: $f(x_0, s) = x$.*

*We define the **accessible part of an automaton** as:*

$$\begin{aligned} Ac(G) &\coloneqq (X_{ac}, E, f_{ac}, x_0, X_{ac,m}) \\ X_{ac} &= \{x \in X \mid \exists s \in E^*, (f(x_0, s) = x)\} \\ X_{ac,m} &= X_m \cap X_{ac} \\ f_{ac} &= f\mid_{X_{ac} \times E \to X_{ac}} \end{aligned}$$

The notion of accessibility has no effect on the languages $L(G)$ and $L_m(G)$. From now on, we'll assume that every automaton is **accessible**, i.e. that $G = Ac(G)$.

The dual notion is coaccessibility, which deals with reaching markings from states.

Definition 4.10 (Coaccessibility)
*A state x is **coaccessible** if it can reach a marked state in X_m: $f(x, s) \in X_m$.*

*We define the **accessible part of an automaton** as:*

$$\begin{aligned} CoAc(G) &\coloneqq (X_{coac}, E, f_{coac}, x_{0,coac}, X_m) \\ X_{coac} &= \{x \in X \mid \exists s \in E^*, (f(x, s) \in X_m)\} \\ x_{0,coac} &= \begin{cases} x_0 & \text{if } x_0 \in X_{coac} \\ undefined & \text{otherwise} \end{cases} \\ f_{coac} &= f\mid_{X_{coac} \times E \to X_{coac}} \end{aligned}$$

Coaccessibility shrinks the language $L(G)$, but it does not affect $L_m(G)$. If G is **coaccessible**, i.e. if $G = CoAc(G)$, then $L(G) = \overline{L_m(G)}$.

Definition 4.11 (Trim automaton)
*An automaton is **trim** if it is both accessible and coaccessible:*

$$Trim(G) \coloneqq CoAc[Ac(G)] = Ac[CoAc(G)]$$

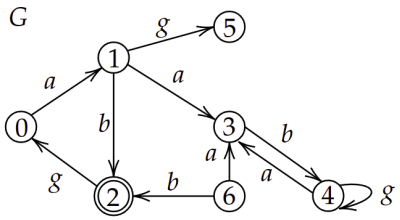
Definition 4.12 (Blocking automaton)
*An automaton is **blocking** if:*

$$L(G) \neq \overline{L_m(G)}$$

which implies that $\overline{L_m(G)}$ is a proper subset of $L(G)$.

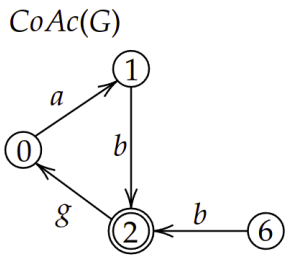
In other words coaccessibility lets us model the deadlocking behavior.

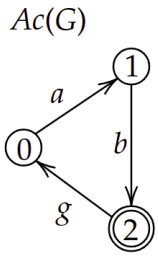
Example 4.1
Consider the following automaton:



*All behavior must be analyzed under the condition that, at some point, we reach state 2, which is indicated by the double circle as an accepting4 state (i.e. a state in which "things happen"). So we can already say that state 5 is in deadlock, while states 3 and 4 are in what is called **livelock**. Livelock means that progress can be made, for example by going from 3 to 4 and vice versa or self looping in 4, but it canno be made towards an accepting state, as there are no traces that bring us out of state 3. So by this intuitive process we determine that the coaccessible part of G is made up of states 0, 1, 2, and 6. We can also observe that state 6 is not reachable under any trace, therefore the accessible part of G is made up of 0, 1 and 2. Note that when we remove states we also remove the transitions attached to them.*

The resulting accessible and coaccessible parts of G are then:



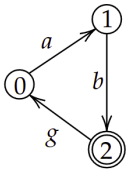


Notice that $Ac(G)$ is also the $Trim(G)$.

Definition 4.13 (Complement)
Given the automaton $G = X, E, f, \Gamma, x_0, X_m)$ that marks the language $K \subseteq E^*$, the automaton G^{comp} that marks the language $E^* \setminus K$ is built as follows:

1. *Complete the transition function f of G making it a total function, f_{tot}.*
   (a) *$X \cup \{x_d\}$, where x_d is a special "dead" or "dump" state.*
   (b)
   $$f_{tot} = \begin{cases} f(x,e) & \text{if } e \in \Gamma(x) \\ x_d & \text{otherwise} \end{cases}$$
   Furthermore, set $f_{tot}(x_d, e) = x_d$ for all $e \in E$.
   (c) *$G_{tot} = (X \cup \{x_d\}, E, f_{tot}, x_0, X_m)$ and $L(G_{tot}) = E^*$ and $L_m(G_tot) = K$.*
2. *$G^{comp})(X \cup \{x_d\}, E, f_{tot}, x_0, (X \cup \{x_d\}) \setminus X_m)$. $L(G^{comp}) = E^*$ and $L_m(G^{comp}) = E^* \setminus L_m(G)$ as desired.*

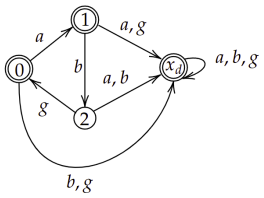
Example 4.2
Consider $Trim(G)$ from the previous example:



We want to compute the complement automaton that marks the language complement of $L(G)$.

To do so we apply the previous definition. Operatively, we switch all non-accepting states to accepting states and vice versa and add an accepting "dead" state x_d. Then we go through all states and add transitions from them to state x_d, marked with all the events not yet accepted by the state. For instance in state 0 only events starting with a are accepted, so we add a transition to x_d marked with b, g, meaning that events starting with b or g will now be accepted by x_d. Note that we do this also for x_d, in which we add a self loop marked with all of the events contained in the alphabet.

The resulting complement automaton is:



So the complement automaton of an automaton G is the automaton that accepts the complement of the language marked by G. For instance, strings a, g and abg are not accepted by G, but they are by its complement. On the other hand abgab is accepted by G but not by its complement.

As we have seen in finite state machines, determinism restricts the range of behaviors we can model. Therefore we extend the definition of automata to allow for two new elements:

1. The event set is augmented to:
$$E_\varepsilon = E \cup \{\varepsilon\}$$
Where a transition labeled ε is interpreted as an internal event of the automaton not observable by the outside world.

2. $f(x, \sigma)$ is no longer single state, but can now be a set of states.

Definition 4.14 (Nondeterministic automaton)
*A **nondeterministic automaton** G_{nd} is a six-tuple:*
$$G_{nd} = (X, E_\varepsilon, f_{nd}, \Gamma, x_0, X_m)$$
where all objects have the same interpretation as in DFAs, except for:

- *f_{nd} is a function $f_{nd} : X \times E_\varepsilon \to 2^X$, i.e. $f_{nd}(x, e) \subseteq X$ wherever it is defined.*
- *The initial state may be itself a set of states, i.e. $x_0 \subseteq X$.*

Nondeterministic automata generate and mark languages the same way as automata do. To describe these languages we need to extend the domain of f_{nd} to traces of events. Let u be a trace of events and e an event, then:
$$f_{nd}(x, ue) \coloneqq \{z \mid z \in f_{nd}(y, e) \text{ for some state } y \in f_{nd}(x, u)\}$$
note that, by convention, $x \in f_{nd}(x, \varepsilon)$.
We define:
$$L(G_{nd}) = \{s \in E^* \mid \exists x \in x_0, (f_{nd}(x, s) \text{ is defined}\}$$
$$L_m(G_{nd}) = \{s \in L(G_{nd}) \mid \exists x \in x_0, (f_{nd}(x, s) \cap X_m \neq \varnothing)\}$$

As we have seen for finite state machines, nondeterministic automata are not more expressive than automata. In fact any nondeterministic automaton can be transformed in an equivalent automaton, i.e. an automaton that generates and marks the same languages. This can be proved formally, and it will be when we'll discuss observer automata.

4.2.1 Composition of automata

Composition of automata is necessary because often systems can be viewed as the composition of many sub-systems.

Definition 4.15 (Product composition of automata)
*The **product composition** of two automatons $G_1 = (X_1, E_1, f_1, \Gamma_1, x_{01}, X_{m1})$ and $G_2 = (X_2, E_2, f_2, \Gamma_2, x_{02}, X_{m2})$, denoted with the "$\times$" symbol, is defined as:*
$$G_1 \times G_2 \coloneqq Ac(X_1 \times X_2, E_1 \cap E_2, f, \Gamma_1 \cap \Gamma_2, (x_{01}, x_{02}), X_{m1} \times X_{m2})$$
where:
$$f((x_1, x_2), \sigma) \coloneqq \begin{cases} (f_1(x_1, \sigma), f_2(x_2, \sigma)) & \text{if } \sigma \in \Gamma_1(x_1) \cap \Gamma_2(x_2) \\ undefined & \text{otherwise} \end{cases}$$

Observation 4.6 (Properties of product composition of automata)
The product composition of automata has the following properties:

1. $L(G_1 \times G_2) = L(G_1) \cap L(G_2)$
2. $L_m(G_1 \times G_2) = L_m(G_1) \cap L_m(G_2)$

Property (2) shows how automata composition implements the intersection of languages.

Product composition is not the only type of composition we need to study automata composition. With the product of automata we enforce the synchronization between the two on all the events in $E_1 \cap E_2$, but this does not cover all possible cases. In fact two automata could work partly synchronized on some events, and independently on other events.

Definition 4.16 (Parallel composition of automata)
*The **parallel composition** of two automatons $G_1 = (X_1, E_1, f_1, \Gamma_1, x_{01}, X_{m1})$ and $G_2 = (X_2, E_2, f_2, \Gamma_2, x_{02}, X_{m2})$, denoted with the "$\|$" symbol, is defined as:*

$$G_1 \parallel G_2 \coloneqq Ac(X_1 \times X_2, E_1 \cup E_2, f, \Gamma_1 \cup \Gamma_2, (x_{01}, x_{02}), X_{m1} \times X_{m2})$$

where:

$$f((x_1, x_2), \sigma) \coloneqq \begin{cases} (f_1(x_1, \sigma), f_2(x_2, \sigma)) & \text{if } \sigma \in \Gamma_1(x_1) \cap \Gamma_2(x_2) \\ (f_1(x_1, \sigma), x_2) & \text{if } \sigma \in \Gamma_1(x_1) \setminus E_2 \\ (x_1, f_2(x_2, \sigma)) & \text{if } \sigma \in \Gamma_2(x_2) \setminus E_1 \\ undefined & otherwise \end{cases}$$

In a parallel composition a common event, i.e. an event $e \in E_1 \cap E_2$, can only be executed if both automata execute it simultaneously. For this reason this operation is also called synchronous composition. The other events are not subject to this constraint and can therefore be executed whenever possible. To analyze the language resulting from the composition we need a new operator.

Definition 4.17 (Natural projections)
*We define **natural projections** $P_i : (E_1 \cup E_2)^* \to E_i^*, i = 1, 2$ as follows:*

$$P_i(\epsilon) = \epsilon$$
$$P_i(\sigma) = \begin{cases} \sigma & \text{if } \sigma \in E_i \\ \epsilon & \text{if } \sigma \notin E_i \end{cases}$$
$$P_i(s\sigma) = P_i(s)P_i(\sigma) \text{ for } s \in (E_1 \cup E_2)^*, \sigma \in (E_1 \cup E_2)$$

We also define the corresponding inverse maps $P_i^{-1} : E_i^* \to 2^{(E_1 \cup E_2)^*}$ as follows:

$$P_i^{-1}(t) = \{s \in (E_1 \cup E_2)^* \mid P_i(s) = t\}$$

The projections P_i and their inverses P_i^{-1} are extended to languages as follows. For $L \subseteq (E_1 \cup E_2)^*$:

$$P_i(L) \coloneqq \{t \in E_i^* \mid \exists s \in L, (P_i(s) = t)\}$$

And for $L_i \subseteq E_i^*$:

$$P_i^{-1}(L_i) \coloneqq \{s \in (E_1 \cup E_2)^* \mid \exists t \in L_i, (P_i(s) = t)\}$$

Inverse natural projections allow us to extend the alphabet associated to an automaton, by means of injecting an arbitrary amount of repeated events t after each event of each string recognized by the language. This injection can be viewed on the automaton as the adding of self loops on each state that make the injected strings be accepted by the automaton. The "direct" projection performs the inverse process, so they remove the injected events.

Observation 4.7 (Properties of parallel composition of automata)
The parallel composition of automata has the following properties:

1. $P_i[P_i^{-1}(L)] = L$, *but* $L \subseteq P_i^{-1}[P_i(L)]$.
2. $P_i[L(G_1 \parallel G_2)] \subseteq L(G_i), i = 1, 2$.
3. $L(G_1 \parallel G_2) = P_1^{-1}[L(G_1)] \cap P_2^{-1}[L(G_2)]$.
4. $L_m(G_1 \parallel G_2) = P_1^{-1}[L_m(G_1)] \cap P_2^{-1}[L_m(G_2)]$.

5. $G_1 \parallel G_2 = G_2 \parallel G_1$, *up to the renaming of states.*

6. $G_1 \parallel (G_2 \parallel G_3) = (G_1 \parallel G_2) \parallel G_3$.

Note that if $E_1 = E_2$, then the parallel composition reduces to the product composition, since all transitions are forced to be synchronized. If $E_1 \cap E_2 = \varnothing$ then there are no synchronized transitions, and thus G is the **concurrent** behavior of G_1 and G_2.

We can furthermore define the $\parallel$ operation on languages. For $L_i \subseteq E_i^*$ and P_i defined as above:

$$L_1 \parallel L_2 = P_1^{-1}(L_1) \cap P_2^{-1}(L_2)$$

Example 4.3
Consider the following alphabets:

$$E_1 = \{a, b\} \quad E_2 = \{b, c\} \quad E_l = E_1 \cup E_2 = \{a, b, c\}$$

And the language:

$$L = \{c, ccb, abc, cacb, cabcbbca\} \subseteq E_l^*$$

We now consider the natural projections of of L from E_e^* to either E_1 or E_2:

$$P_i : E_e^* \to E_i, i = 1, 2$$

We obtain:

$$P_1(L) = \{\epsilon, b, ab, abbba\}$$
$$P_2(L) = \{c, ccb, bc, ccb, cbcbbc\}$$

The projections are the shrinking of the language with respect to the string not accepted by either of the alphabets, i.e. P_1 restricts L by removing c, while P_2 restricts L by removing a.
The inverse projections are:

$$P_1^{-1}[P_i(L)] = \{c^*, c^*bc^*, c^*ac^*bc^*, c^*ac^*bc^*bc^*bc^*ac^*\}$$
$$P_2^{-1}[P_2(L)] = \{a^*ca^*, a^*ca^*ca^*ba^*, a^*ba^*ca^*, a^*ca^*ca^*ba^*, a^*ca^*ba^*ca^*ba^*ba^*ca^*\}$$

Notice that property (1) holds, i.e. we have $P_i^{-1}[P_i(L)] \subseteq L$. In other words what we get from the inversion is a subset of the original language, not a set equal to it.

4.2.2 Observer automata

Consider a DES modeled by a (possibly nondeterministic) automaton:

$$G_{nd} = (X, E \cup \{\varepsilon\}, f_{nd}, \Gamma, x_0, X_m)$$

We now partition the set of events E of G as:

$$E = E_o \cup E_{uo}$$

where E_0 is the set of **observable states**, i.e. the states that in a real system would be registered by sensors, and E_{uo} is the set of **unobservable states**. Note that, following from their definition, ε states are not observable.

Our objective now is to estimate the state of G_{nd} from traces of observed events only, i.e. to design an **observer automaton**.

Definition 4.18 (Observer automaton construction algorithm)
*Let $G_{nd} = (X, E \cup \{\varepsilon\}, f_{nd}, \Gamma, x_0, X_m)$ be a nondeterministic automaton and let $E = E_o \cup E_{uo}$. Then $G_{obs} = (X_{obs}, E_o, f_{obs}, x_{0,obs}, X_{m,obs})$ is the associated **observer automaton**, and it is constructed as follows:*

1. *Replace all transitions of G_{nd} labeled by events in E_{uo} with ε-transitions.*

2. *Start with $X_{obs} = 2^X \setminus \varnothing$.*

3. *For each state $x \in X$ define its **unobservable reach**:*

$$UR(x) \coloneqq f_{nd}(x, \varepsilon)$$

4. *Define* $x_{0,obs} = UR(x_0)$.

5. *For each* $S \subseteq X$ *and* $e \in E$, *define:*

$$f_{obs}(S, e) = UR(\{x \in X \mid \exists x_e \in S, [x \in f_{nd}(x_e, e)]\}$$

6. $X_{m,obs} = \{S \subseteq X \mid S \cap X_m \neq \varnothing\}$

7. *Repeat the above steps in a breadth-first manner, so only the accessible part of* G_{obs} *is constructed. The resulting state space* X_{obs} *is a subset of* 2^X.

Observation 4.8 (Properties of the observer automaton)
The properties of the observer automaton are:

1. G_{obs} *is a deterministic automaton with event set* E_0.

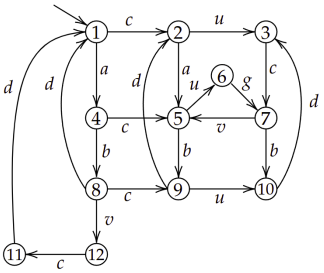
2. $L(G_{obs}) = P_o[L(G_{nd})]$, *where* P_o *is the natural projection* $P_o : E \to E_o$.

3. $L_m(G_{obs}) = P_o[L_m(G_{nd})]$.

4. *Let* $f_{obs}(x_{0,obs}, t) = S$, *where* $t \in P_0[L(G_{nd})]$. *Then* $x \in S$ *if and only if there exists* $s \in L(G_{gnd})$ *such that* $x \in f_{nd}(y, s)$ *for some* $y \in x_0$ *and* $P_o(s) = t$. *Hence,* S *is the set of all states* G_{nd} *could be in after observing* t, *namely* S *is the* ***state estimate*** *of* G_{nd} *after* t.

A consequence of properties (2) and (3) is that nondeterministic automata have the same modeling power as deterministic automata.

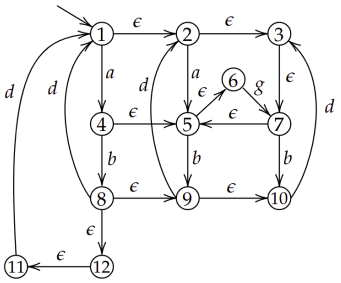
Example 4.4
Consider the automaton:



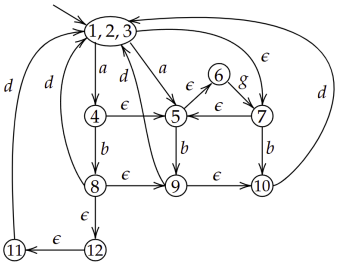
It is known that the alphabet:

$$E_{uo} = \{c, u, v\}$$

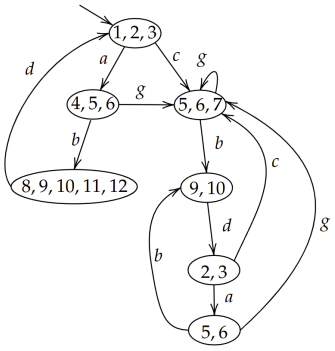
is unobservable.
First of all we replace all transitions in the unobservable alphabet with ϵ*-transitions:*



Then we apply epsilon-closure. First of all we replace the initial state 1 with an initial state 1,2,3 that takes into account the ϵ*-transitions:*



We then apply ϵ-closure again, by following the graph and replacing the states reached via an ϵ-transition with a single state made up of all of them. For instance, from state 1,2,3 with transition a we reach both states 4 and 5. From there via an ϵ-transition we reach either state 5 or state 6. Therefore we put a transition a from state 1,2,3 to state 4,5,6. We repeat the process for the rest of the graph, obtaining:



The resulting automaton is the observer automaton.

Note that when the starting transition is already an ϵ-transition, we replace it with the corresponding unobservable event.

4.3 Regular languages and finite-state automata

Definition 4.19 (Regular language)
*A language K is **regular**, i.e. $K \in \mathcal{R}$, if there exists a (deterministic) finite state automaton G that marks it, i.e. $L_m(G) = K$.*

Not all languages are regular, and we generally need to keep track of an arbitrarily large number of events that happen before another event. Therefore, by Pumping's Lemma, any finite number of state is not sufficient.

Definition 4.20 (Pumping's Lemma)
Let L be an infinite regular language. Then there exist subtraces x, y and z such that (i) $y \neq \epsilon$ and (ii) $xy^n z \in L$ for all $n \geq 0$.

Theorem 4.1 (Closeness of $\mathcal{R}$)
The class of regular languages $\mathcal{R}$ is closed under:

1. *Union*
2. *Concatenation*
3. *Kleene-closure*
4. *Complementation with respect to E^**
5. *Intersection*

The above theorem points can be proved by considering two automata, G_1 and G_2, and then, respectively:

1. Creating a new initial state and connecting it with ϵ-transitions to the two initial states of the respective automata.
2. Connecting the marked states of G_1 to the initial state of G_2 with ϵ-transitions, and then unmarking all states of G_1.
3. Adding a new initial state, marking it and connecting it to the old initial state with an ϵ-transition. Then add ϵ-transitions from every marked state to the old initial state.
4. Use the complement operation.
5. Take the product of the two automata.

4.4 The feedback loop of supervisory control

We'll now consider a discrete event system (DES) modeled by a pair of languages, L and L_m, where $L = \overline{L}$ is the set of all traces the DES can generate and $L_m \subseteq L$ is the language of marked traces that is used to represent the completion of some tasks. Both L and L_m are defined over an event set E. We'll assume that both languages are generated by an automaton $G = (X, E, f, \Gamma, x_0, X_m)$: $L(G) = L, L_m(G) = L_m$. We will from now on refer to this model as "DES G".

Control is necessary because an uncontrolled DES G does not in general satisfy **safety** or **nonblocking** specifications. The specifications of safety and "nonblocking-ness" are collectively referred to as **legal behavior**.

Legal behavior is generally defined as a subset of $L(G)$, usually denoted with L_a where "a" stands for "admissible". In blocking problems legal behavior is a subset of $L_m(G)$, usually denoted with L_{am}.

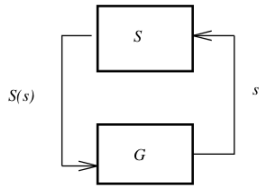
We now partition the event set E:

$$E = E_c \cup E_{uc}$$

where E_c is the **set of controllable events**, which are events that can be disabled from happening by control, and E_{uc} is the **set of uncontrollable events**, which are events not preventable by control.

There are reasons for having events modeled as uncontrollable. For instance an event could be inherently unpreventable, such as plant failure, unpreventable due to hardware limitations or unpreventable by design, such as high priority events, or if the event represents the ticking of a clock.

We now assume that the transition function of G can be controlled by an external agent, i.e. a **controller** or **supervisor** S that can enable or disable the controllable events of G.



The control paradigm then is made up of a feedback loop, in which G represents the uncontrolled behavior of the DES, and S the controller that controls the controllable behavior.

Definition 4.21 (Supervisor)
*A **supervisor** is a function:*

$$S \coloneqq L(G) \mapsto 2^E$$

Definition 4.22 (Enabled events set)
For each $s \in L(G)$ generated so far by G (under the control of S):

$$S(s) \cap \Gamma(f(x_0, s))$$

*is the **set of enabled events** that G can execute in its current state $f(x_0, s)$. In other words no event in the current active event set of G can be executed if it is not also found in $S(s)$.*

Definition 4.23 (Admissible supervisor)
*A supervisor is **admissible** if, for all $s \in L(G)$:*

$$E_{uc} \cap \Gamma(f(x_0, s)) \subseteq S(s)$$

that is to say S is never allowed to disable a feasible uncontrollable event.

Definition 4.24 (Notation and nomenclature)
*$S(s)$ is called **control action** at s. S is called **control policy**. We denote the feedback loop of S and G as $S \setminus G$.*

Definition 4.25 (Language generated by $S \setminus G$)
*The **language generated** by $S \setminus G$ is defined recursively as:*

1. $\varepsilon \in L(S \setminus G)$
2. $[s \in L(S \setminus G)) \wedge (s\sigma \in L(G)) \wedge (\sigma \in S(s))] \iff [s\sigma \in L(S \setminus G)]$

We note that $L(S \setminus G) \subseteq L(G)$, and it is prefix-closed by definition.

Definition 4.26 (Language marked by $S \setminus G$)
*The **language marked** by $S \setminus G$ is defined as follows:*

$$L_m(S \setminus G) \coloneqq L(S \setminus G) \cap L_m(G)$$

In other words it consists exactly of the marked traces of G that survive under the control of S.

Observation 4.9
We observe that:

$$\varnothing \subseteq L_m(S \setminus G) \subseteq \overline{L_m(S \setminus G)} \subseteq L(S \setminus G) \subseteq L(G)$$

So we can think of the controlled DES $S \setminus G$ as an automaton that generates $L(S \setminus G)$ and marks $L_m(S \setminus G)$.

Definition 4.27 (Nonblocking supervisor)
*S is said to be **nonblocking** if $S \setminus G$ is nonblocking, i.e. if:*

$$L(S \setminus G) = \overline{L_m(S \setminus G)}$$

*otherwise S is said to be **blocking**.*

Since marked traces represent completed tasks or the record of the completion of tasks, blocking means that the controlled system cannot terminate the current task. Therefore the notion of marked trace and blocking allow us to model deadlock and livelock.

4.4.1 P-supervisor

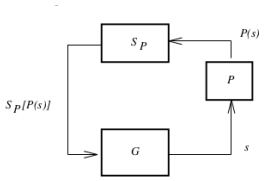
Until now we have assumed all events in G as observable. This is not always the case, as some events could not be recorded by available sensors, could be happening remotely and so on.

So, we partition the event set in two subsets:

$$E = E_o \cup E_{up}$$

where E_o is the **set of observable events**, that are available to the supervisor, and E_{uo} is the **set of unobservable events**, which are the events not accessible to the supervisor.

The feedback loop then changes to account for the unobservable events:



In the feedback loop we add the natural projection $P : E^* \to E_o^*$. Its function is to erase all observable events that might appear in a trace.

Due to the presence of P the supervisor cannot distinguish between traces that have the same projection, $P(s_1) = P(s_2), s_1, s_2 \in L(G)$. Therefore the supervisor must take the same control action for both traces. TO account for this fact we introduce the notion of **control action under partial observation**:

$$S_P : P[L(G)] \to 2^E$$

S_P is called **P-supervisor**.

When an enabled observable event occurs, $P(s)$ changes and the control action is instantaneously updated, that is to say that the control action $S_P(t)$ for $t \in P[L(G)]$ is applied by S_P immediately after the last execution by G of the observable event of t, and it remains until another observable event is executed by G.

Definition 4.28 (Admissible P-supervisor)
*A P-supervisor is **admissible** if it never disables a feasible uncontrollable event.*
We define:

$$L_t = P^{-1}(t')\{\sigma\}(S_P(t) \cap E_{uo})^* \cap L(G)$$

L_t contains all the strings in $L(G)$ that are effectively subject of the control action $S_P(t)$, when S_P controls G.
A P-supervisor is admissible if, for all $t = t'\sigma \in P[L(G)]$:

$$E_{uc} \cap \left[\bigcup_{s \in L_t} \Gamma(f(x_0, s)) \right] \subseteq S_P(t)$$

The term in the brackets represents all the feasible continuations in $L(G)$ of all the strings that $S_P(t)$ applies to.

Definition 4.29 (Language generated by $S_P \setminus G$)
*The **language generated** by $S_P \setminus G$ is defined recursively as follows:*

1. $\varepsilon \in L(S_P \setminus G)$
2. $[(s \in L(S_P \setminus G)) \wedge (s\sigma \in L(G)) \wedge (\sigma \in S_P[P(s)])] \iff [s\sigma \in L(S_P \setminus G)]$

Definition 4.30 (Language marked by $S_P \setminus G$)
*The **language marked** by $S_P \setminus G$ is defined as follows:*

$$L_m(S_P \setminus G) \coloneqq L(S_P \setminus G) \cap L_m(G)$$

Observation 4.10 (Event set)
Note that both $L(S_P \setminus G)$ and $L_m(S_P \setminus G)$ are defined over E and not E_o, i.e. they correspond to the closed-loop behavior of G before the effect of P.

Observation 4.11 (Observable vs controllable)
Controllability and observability have not been related, i.e. an unobservable event could be controllable and vice versa.

4.4.2 Legal behavior

After accounting for all the specifications imposed on the system, we obtain L_a (or L_{am}). These specifications are described by a series of (possibly marked) languages:

$$K_{s,i}, i = 1, \ldots, m$$

If a specification language $K_{s,i}$ is not given as a subset of either $L(G)$ or $L_m(G)$, then we take either the intersection:

$$L_{a,i} = L(G) \cap K_{s,i} \quad \text{or} \quad L_{am,i} = L_m(G) \cap K_{s,i}$$

Or the parallel composition:

$$L_{a,i} = L(G) \parallel K_{s,i} \quad \text{or} \quad L_{am,i} = L_m(G) \parallel K_{s,i}$$

The choice of intersection or parallel composition depends on the respective event sets of $L(G)$ and $K_{s,i}$. If the events that appear $L(G) \cap K_{s,i}$ are irrelevant to $K_{s,i}$, then we take the parallel composition. On the other hand, if these events are absent from $K_{s,i}$ because they should not happen in the legal behavior, then we take the intersection.

The question now becomes to determine under what conditions any given languages L_a and L_{am} can be achieved exactly by a supervisor S or a P-supervisor S_P. This can be done in a number of ways, depending on the specification:

- **Illegal states**: If a specification identifies some states of G as illegal, then it is enough to delete these states from G, i.e. remove the illegal states and the transitions attached to them and perform the Ac operation.
- **State splitting**: If a specification requires remembering how a certain state of G was reached, in order to determine the legality of future behavior, then that state has to be splitted as many times as necessary, adjusting the active event set of each new state according to the respective legal continuations.
- **Event alternance**: If a specification requires the alternance of two events, then build a two-state automaton that implements this alternance. Then take the parallel composition with G in order to get the generator of the legal language.
- **Illegal subtrace**: If a specification identifies as illegal all traces of $L(G)$ that contain a forbidden subtrace $s_f = \sigma_1 \ldots \sigma_n \in \Sigma^*$, then we build the automaton $H_{spec} = (X, E, f, x_0, X)$ in the following way:
  1. $X = \{\varepsilon, \sigma_1, \sigma_1\sigma_2, \ldots, \sigma_1 \ldots, \sigma_{n-1}\}$, i.e. we associate a state of H_{spec} to each proper prefix of s_f.
  2. The transition function f is constructed as follows:
     - $f(\sigma_1, \ldots, \sigma_i, \sigma_{i+1}) = \sigma_1 \ldots \sigma_{i+1}, i = 0, \ldots, n-2$
     - Complete f to E for all states in X, except for state $\sigma_1 \ldots \sigma_{n-1}$ which is completed to $E \setminus \{\sigma_n\}$ (since that event is illegal in that state), in the following way:

       $$f(\sigma_1 \ldots \sigma_i, \gamma) = \text{state in } X \text{ corresponding to the longest suffix of } \sigma_1 \ldots \sigma_i\gamma$$

  3. Take $x_0 = \varepsilon$

  $$L(H_{spec}) = L_m(H_{spec}) = E^* \setminus E^*\{s_f\}E^*$$

  Finally, the desired H_a is obtained as $H_a = H_{spec} \times G$. This procedure can be extended to a finite set of illegal subtraces, i.e. a single H_{spec} is constructed at once instead of repeating the procedure for all illegal subtraces.

4.5 Controllability and observability

The results of the theory discussed so far are summarized in three theorems.

Theorem 4.2 (Controllability theorem)
Consider a DES G where $E_{uc} \subseteq E$ is the subset of uncontrollable events. Let $K \subseteq L(G), K \neq \varnothing$.
There exists a supervisor S such that:

$$L(S \setminus G) = \overline{K}$$

if and only if the following condition (controllabilty) holds:

$$\overline{K}E_{uc} \cap L(G) \subseteq \overline{K}$$

Proof 4.2
We now prove the controllability theorem:

- $\Longrightarrow$ *: For $s \in L(G)$ define $S(s)$ according to:*

$$S(s) = [E_{uc} \cap \Gamma(f(x_0, s))] \cup \{\sigma \in E_c \mid s\sigma \in \overline{K}\}$$

This supervisor enables after trace s: (i) all uncontrollable events feasible in G after trace s and (ii) all controllable events that extend s inside of $\overline{K}$. Part (i) ensures that S is admissible, i.e. that it never disables a feasible uncontrollable event.

We now prove that with such S, $(L(S \setminus G) = \overline{K}$. The proof is by induction on the length of traces in the two languages.

  - *The base case is for traces of length 0. But $\varepsilon \in L(S \setminus G)$ by definition, and $\varepsilon \in \overline{K}$ since $K \neq \varnothing$, thus the base case holds.*
  - *The induction hypothesis is that for all traces s such that $|s| \leq n, s \in L(S \setminus G)$ if and only if $s \in \overline{K}$. We now prove the same for traces in the form $s\sigma$:*
    - *Let $s\sigma \in L(S \setminus G)$. By definition of $L(S \setminus G)$, this implies that:*

$$[s \in L(S \setminus G)] \wedge [\sigma \in S(s)] \wedge [s\sigma \in L(G)]$$

which in turn implies, using the induction hypothesis, that:

$$[s \in \overline{K}] \wedge [\sigma \in S(s)] \wedge [s\sigma \in L(G)]$$

If $\sigma \in E_{uc}$, then the controllability condition immediately yields $s\sigma \in \overline{K}$. If $\sigma \in E_c$, by the definition of S we also obtain that $s\sigma \in \overline{K}$. This proved the "if" part of the induction step.
    - *Let $s\sigma \in \overline{K}$. Then $s\sigma \in L(G)$ since by assumption $\overline{K} \subseteq L(G)$. If $\sigma \in E_{uc}$ then $\sigma \in S(s)$ by the definition of $S(s)$. If $\sigma \in E_c$ then once again $\sigma \in S(s)$ by the definition of $S(s)$.*
Overall we have that:

$$[s \in \overline{K} \wedge [\sigma \in S(s)] \wedge [s\sigma \in L(G)]$$

which using the induction hypothesis implies that:

$$[s \in L(S \setminus G)] \wedge [\sigma \in S(s)] \wedge [s\sigma \in L(G)]$$

It immediately follows that $s\sigma \in L(S \setminus G)$. This has proved the "only if" part of the induction step.

- $\Longleftarrow$ *: Let there exist an admissible S such that $L(S \setminus G) = \overline{K}$. Let $s \in \overline{K}$, $\sigma \in E_{uc}$ and $s\sigma \in L(G)$.*

  - *$\sigma \in S(s)$ since any admissible supervisor is not allowed to disable a feasible uncontrollable event.*
  - *By definition of $L(S \setminus G)$ we have however that:*

$$[s \in \overline{K} = L(S \setminus G)] \wedge [s\sigma \in L(G)] \wedge [\sigma \in S(s)] \implies s\sigma \in L(S \setminus G) = \overline{K}$$

  - *Therefore we have shown that:*

$$[s \in \overline{K}] \wedge [\sigma \in E_{uc}] \wedge [s\sigma \in L(G)] \implies s\sigma \in \overline{K}$$

or, in terms of languages:

$$\overline{K}E_{uc} \cap L(G) \subseteq \overline{K}$$

which is the controllability condition.

QED

Observation 4.12 (Construction of the supervisor)
The proof of the controllability theorem is constructive, because if the controllability condition holds, the supervisor that achieves the required behavior is:

$$S(s) = [E_{uc} \cap \Gamma(f(x_0, s))] \cup \{\sigma \in E_c \mid s\sigma \in \overline{K}\}$$

The controllability condition is intuitive, it can be summarized as: "if you cannot prevent it, it should be legal". We define more formally this notion.

Definition 4.31 (Controllability)
*Let K and $M = \overline{M}$ be languages over an event set E. Let E_{uc} be a designated subset of E. K is said to be **controllable** with respect to M and E_{uc} if and only if $\forall s \in \overline{K}$ and $\forall \sigma \in E_{uc}$:*

$$\overline{K}E_{uc} \cap M \subseteq \overline{K}$$

By definition, controllability is a property of the prefix-closure of a language. Thus K is controllable if and only if $\overline{K}$ is controllable. The definition also provides a condition to verify if this property holds or not.

Theorem 4.3 (Nonblocking controllability theorem)
Consider a DES G where $E_{uc} \subseteq E$ is the set of uncontrollable events. Consider also the language $K \subseteq L_m(G)$ where $K \neq \varnothing$.

There exists a nonblocking supervisor S for G such that:

$$L_m(S \setminus G) = K \implies L(S \setminus G) = \overline{K}$$

if and only if the following conditions hold:

- *Controllability: $\overline{K}E_{uc} \cap L(G) \subseteq \overline{K}$*
- *$L_m(G)$-closure: K is $L_m(G)$-closed, i.e. $K = \overline{K} \cap L_m(G)$.*

Proof 4.3
We now prove the nonblocking controllability theorem:

- *$\implies$: For $s \in L(G)$, as in the proof of the controllability theorem, define $S(s)$ as:*

  $$S(s) = [E_{uc} \cap \Gamma(f(x_0, s))] \cup \{\sigma \in E_c \mid s\sigma \in \overline{K}\}$$

  The proof of the controllability theorem already establishes that with this S, $S \setminus G = \overline{K}$. Therefore $L_m(S \setminus G) = K$ follows by applying the $L_m(G)$-closure condition. It follows that S is nonblocking.

- *$\impliedby$: Let there exist an admissible S such that $L_m(S \setminus G) = K$ and $L(S \setminus G) = \overline{K}$. Then by the definition of $L_m(S \setminus G)$ we have that $K = \overline{K} \cap L_m(G)$, which is the $L_m(G)$-closure condition. The remainder of the proof is the same as the proof of the controllability theorem.*

QED

Observation 4.13 (Construction of the supervisor)
Just like the controllability theorem, the nonblocking controllability theorem gives us a supervisor:

$$S(s) = [E_{uc} \cap \Gamma(f(x_0, s))] \cup \{\sigma \in E_c \mid s\sigma \in \overline{K}\}$$

We will now generalize the nonblocking controllability theorem to the case of P-supervisors. It is clear that we need another condition beyond controllability, i.e. observability, which intuitively means: "if you cannot differentiate between two traces, then they require the same control action".

Definition 4.32 (Observability)
*Let K and $M = \overline{M}$ be languages over an event set E. Let E_c be a designated subset of E. Let E_o be another designated subset of E with P as the corresponding natural projection from E^* to E_o^*. K is said to be **observable** with respect to M, P and E_c if and only if $\forall s \in \overline{K}$ and $\forall \sigma \in E_c$:*

$$[s\sigma \notin \overline{K}] \wedge [s\sigma \in M] \implies P^{-1}[P(s)]\sigma \cap \overline{K} = \varnothing$$

The right hand side of the implication identifies all traces in $\overline{K}$ that have the same projection as s and that can be continued with event σ. If the set is not empty, then $\overline{K}$ contains two traces s and s' such that $P(s) = P(s')$, and where $s\sigma \notin \overline{K}$ while $s'\sigma \in \overline{K}$. If this happens, then no P-supervisor can exactly achieve language $\overline{K}$.

As we have seen for controllability, observability is a property of the prefix-closure of a language. Therefore K is observable if $\overline{K}$ is.

Theorem 4.4 (Controllability and observability theorem)
Consider a DES G where $E_{uc} \subseteq E$ is the set of uncontrollable events and $E_o \subseteq E$ is the set of observable events. Let P be the natural projection form E^* to E_o^*. Consider also the language $K \subseteq L_m(G)$ where $K \neq \varnothing$.

There exists a nonblocking P-supervisor S_P for G such that:

$$L(S_P \setminus G) = K$$

if and only if the following conditions hold:

1. *K is controllable with respect to $L(G)$ and E_{uc}*
2. *K is observable with respect to $L(G), P$ and E_{uc}*
3. *K is $L_m(G)$-closed*

Proof 4.4
We now prove the controllability and observability theorem:

- $\Longrightarrow$ *: For $t \in P[L(G)]$ define $S_P(t)$ as: $S_P(t) = E_{uc} \cup \{\sigma \in E_c \mid \exists s'\sigma \in \overline{K}(P(s') = t)\}$.*

  This supervisor enables after trace s: (i) all uncontrollable events and (ii) all controllable events that extend any trace s', that projects to t, inside $\overline{K}$. Part (i) ensures that S_P is admissible, i.e. that it never disables a feasible uncontrollable event.

  The proof is by induction on the length of the traces in the two languages.

  - *The base case is for traces of length 0. But $\varepsilon \in L(S_P \setminus G)$ by definition, and $\varepsilon \in \overline{K}$ since $K \neq \varnothing$, thus the base case holds.*
  - *The induction hypothesis is that for all traces s such that $|s| \leq n, s \in L(S_P \setminus G)$ if and only if $s \in \overline{K}$. We now prove the same for traces in the form $s\sigma$ where $|s| = n$:*
    - *Let $s\sigma \in L(S_P \setminus G)$. By definition of $L(S_P \setminus G)$, this implies that:*

      $$[s \in L(S_P \setminus G)] \wedge [\sigma \in S_P[P(s)]] \wedge [s\sigma \in L(G)]$$

      which in turn implies, using the induction hypothesis, that:

      $$[s \in \overline{K}] \wedge [\sigma \in S_P[P(s)]] \wedge [s\sigma \in L(G)]$$

      If $\sigma \in E_{uc}$, then the controllability condition immediately yields $s\sigma \in \overline{K}$. If $\sigma \in E_c$, by the definition of S_P and with $t = P(s)$ we obtain that there exists $s'\sigma \in \overline{K}$ such that $P(s') = t = P(s)$. So we have that:

      $$P^{-1}[P(s)]\sigma \cap \overline{K} \neq \varnothing$$

      But since $s\sigma \in L(G)$ we must have that $s\sigma \in \overline{K}$, otherwise observability would be contradicted. This proved the "if" part of the induction step.
    - *Let $s\sigma \in \overline{K}$. Then $s\sigma \in L(G)$ since by assumption $\overline{K} \subseteq \overline{L_m(G)} \subseteq L(G)$. If $\sigma \in E_{uc}$ then $\sigma \in S_P[P(s)]$ by the admissibility of S_P. If $\sigma \in E_c$ then once again $\sigma \in S_P[P(s)]$ by the definition of $S_P[P(s)]$. Overall we have that:*

      $$[s \in \overline{K} \wedge [\sigma \in S_P[P(s)]] \wedge [s\sigma \in L(G)]$$

      which using the induction hypothesis implies that:

      $$[s \in L(S_P[P(s)] \setminus G)] \wedge [\sigma \in S_P[P(s)]] \wedge [s\sigma \in L(G)]$$

      It immediately follows that $s\sigma \in L(S_P[P(s)] \setminus G)$. This has proved the "only if" part of the induction step.

- $\Longleftarrow$ *: Let S_P be an admissible P-supervisor such that $L(S_P \setminus G) = \overline{K}$ and $L_m(S_P \setminus G) = K$. The proof that controllability and $L_m(G)$-closure hold is the same as the nonblocking controllability theorem, except that $S(s)$ is replaced by $S_P[P(s)]$.*

  To prove that observability must hold, take $s \in \overline{K}$ and $\sigma \in E_c$ such that $s\sigma \notin \overline{K}$ and $s\sigma \in L(G)$.

  - *From $s \in \overline{K}$,$\sigma \in E_c$,$s\sigma \notin \overline{K}$, and $L(S_P \setminus G) = \overline{K}$, it must be that $\sigma \notin S_P[P(s)]$, otherwise $L(S_P \setminus G) \neq \overline{K}$.*
  - *This means that there cannot exist $s'\sigma \in \overline{K}$ such that $P(s') = P(s)$, otherwise $L(S_P \setminus G) \neq \overline{K}$, since S_P does not distinguish between s and s'.*

QED

Observation 4.14 (Construction of the supervisor)
Once again, the proof is constructive, i.e. it also gives us the supervisor:

$$S_P(t) = E_{uc} \cup \{\sigma \in E_c \mid \exists s'\sigma \in \overline{K}(P(s') = t)\}$$

Corollary 4.4.1
Consider a DES G where $E_{uc} \subseteq E$ is the set of uncontrollable events and $E_o \subseteq E$ is the set of observable events. Let P be the natural projection form E^* to E_o^*. Consider also the language $K \subseteq L_m(G)$ where $K \neq \varnothing$. Then there exists S_P such that $L(S_P \setminus G) = \overline{K}$ if and only if K is controllable with respect to $L(G)$ and E_{uc} and observable with respect to $L(G), P$ and E_c.

4.6 Realization and design of controllers

Let there exist a supervisor S such that $L(S \setminus G) = \overline{K}$. Thus K is a **controllable sublanguage** of $L(G)$. If we want to extend the notion to marked languages, then under $L_m(G)$-closure we get that $L_m(S \setminus G) = K$ and S is nonblocking.

We also rule out the trivial cases $\overline{K} = \varnothing$, which is controllable by definition but unachievable by control unless the system never executes events, and $\overline{K} = L(G)$, which is also controllable by definition but S plays no part in it, so it doesn't interest us.

The issue is that we need a convenient representation of the function S for implementation purposes, rather than simply listing all $S(s)$ for all $s \in L(S \setminus G)$. Since the system is represented by an automaton, it makes sense to represent the supervisor as an automaton. This automaton is called **realization** of S.

An easy way to realize S is to build an automaton that recognizes the language $\overline{K}$ in the following way:

- Let R be the realization of S, i.e. let:
$$R = (Y, E, g, \Gamma_R, y_0, Y)$$
where R is trim and:
$$L_m(R) = L(R) = \overline{K}$$
- If we now connect the realization of S to G using the product operation, the resulting $R \times G$ has exactly the behavior required for the closed loop system $S \setminus G$:
$$L(R \times G) = L(R) \cap L(G) = \overline{K} \cap L(G) = \overline{K} = L(S \setminus G)$$
$$L_m(R \times G) = L_m(R) \cap L_m(G) = \overline{K} \cap L_m(G) = L(S \setminus G) \cap L_m(G) = L_m(S \setminus G)$$
- Note that, since R is defined on the same event set as G, then $R \parallel G = R \times G$.

The above means that the control action $S(s)$ is encoded in the transition structure of R, namely:

$$S(s) = \Gamma_{R \times G}(g \times f((y_0, x_0), s = \Gamma_R(g(y_0, s))$$

Where the last equality follows from $\overline{K} \subseteq L(G)$, and $g \times f$ denotes the transition function of $R \times G$.

$R \times G$ is a composition of automata defined without reference to a control mechanism, unlike $S \setminus G$. Its interpretation is the following: let G be in state x and R in state y following the execution of $s \in L(S \setminus G)$. G generates an event σ that is currently enabled. This means that this event is also present in the active event set of R at y. Thus R also executes the event, as a passive observer of G. Let x' and y' be the new states of G and R after the execution of σ. The set of enabled events of G at $s\sigma$ is now given by the active event set of R at y'.

Thus we have built a representation of S that, if K is regular, requires only finite memory. We call the R derived above the **standard realization** of S.

If we are given an automaton C and form its product $C \times G$, it cannot always be interpreted as the control of G by some supervisor. The property of "controlling" depends on whether or not $L(C)$ is controllable.

Definition 4.33 (Induced supervisor)
*Let $C = (Y, E, g, \Gamma_R, y_0, Y)$ be a trim automaton. We define the **supervisor for G induced by C** as follows. For all $s \in L(G)$:*

$$S_i^C(s) = \begin{cases} [E_{uc} \cap \Gamma(f(x_0, s))] \cup \{\sigma \in E_c \mid s\sigma \in L(C)\} & \text{if } s \in L(G) \cap L(C) \\ E_{uc} & \text{otherwise} \end{cases}$$

Note that we need $E_{uc} \cap \Gamma(f(x_0, s))$ in order to make sure that S_i^C is an admissible supervisor.

From the previous definition we can derive the fact that:

$$L(S_i^C \setminus G) = L(C \times G)$$

if and only if $L(C)$ is controllable with respect to $L(G)$ and E_{uc}.

The process for building the realization of a P-supervisor is slightly more complex sue to the presence of unobservable events.

A realization S_P can be built as follows:

1. Build a trim automaton R that generates $\overline{K}$.
2. Build R_{obs}, the observer for R.

   At this point, we can no longer say that the active event set of R_{obs} encodes the set of enabled events by the function S_P, since the event set of R_{obs} is E_o and thus contains no information on the desired control action with respect to the unobservable events, which is contained in R, but since each state of R_{obs} is a set of states of R we can recover the required control action.
3. Let t be the current trace of observable events, and let $x_{obs,current}$ be the state of R_{obs} after the execution of t. Therefore after the last observable event in t, but before the next one, R can be in any of the states contained in $x_{obs,current}$.
4. Then we have that:
$$S_P^{real}(t) = \bigcup_{x \in x_{obs,current}} [\Gamma_R(x)]$$

In this interpretation R_{obs} is a "passive observer", that follows only the observable transitions of G. The desired control action, $S_P(t)$, is then obtained by looking at the current state of R_{obs} and considering the corresponding active event sets in R of all states in this set. As before, we call this the **standard realization** of S_P.

4.7 Uncontrollability

Observation 4.15 (Properties of controllability)
Some properties of controllability:

- *If K_1 and K_2 are controllable, then $K_1 \cup K_2$ is controllable. In fact:*
$$\overline{(K_1 \cup K_2)}E_{uc} \cap M = (\overline{K}_1 \cup \overline{K}_2)E_{uc} \cap M = (\overline{K}_1 E_{uc} \cap M) \cup (\overline{K}_2 E_{uc} \cap M) \subseteq \overline{K}_1 \cup \overline{K}_2 = \overline{K_1 \cup K_2}$$
- *If K_1 and K_2 are controllable, then $K_1 \cap K_2$ is not generally controllable. A counterexample is $E_{uc} = \{\alpha\}$, $E = \{\alpha, \beta, \gamma\}$, $M = \{\varepsilon, \alpha, \alpha\beta, \alpha\gamma\}$, $K_1 = \{\varepsilon, \alpha\beta\}$ and $K_2 = \{\varepsilon, \alpha\gamma\}$.*
- *If K_1 and K_2 are nonconflicting and controllable, then $K_1 \cap K_2$ is controllable. In fact:*
$$\overline{(K_1 \cap K_2)}E_{uc} \cap M = (\overline{K}_1 \cap \overline{K}_2)E_{uc} \cap M = (\overline{K}_1 E_{uc} \cap M) \cap (\overline{K}_2 E_{uc} \cap M) \subseteq \overline{K}_1 \cap \overline{K}_2 = \overline{K}_1 \cap \overline{K}_2$$
  Which contradicts the nonconflicting property $K_1 \cap \overline{K}_2 = \overline{(K_1 \cap K_2)}$.
- *If K_1 and K_2 are prefix-closed and controllable, then $K_1 \cap K_2$ is prefix-closed and controllable. This is an immediate consequence of the previous property, since all prefix-closed languages are nonconflicting.*

Definition 4.34 (Controllable sublanguages and superlanguages)
Given a language K we define:
$$C_{in}(K) \coloneqq \{L \subseteq K \mid \overline{L}E_{uc} \cap M \subseteq \overline{L}\}$$
*the **set of controllable sublanguages of** K, and:*
$$CC_{out}(K) \coloneqq \{L \subseteq E^* \mid (K \subseteq L \subseteq M) \wedge (\overline{L} = L) \wedge (\overline{L}E_{uc} \cap M \subseteq \overline{L})\}$$
*the **set of controllable superlanguages of** K, and:*

Definition 4.35 (Supremal controllable language)
We define:
$$K^{\uparrow C} \coloneqq \bigcup_{L \in C_{in}(K)} L$$
*the **supremal controllable language of** K. In the worst case, $K^{\uparrow C} = \varnothing$, since $\varnothing \in C_{in}(K)$. If K is controllable, then $K^{\uparrow C} = K$. $K^{\uparrow C}$ is not prefix-closed in general.*

Observation 4.16 (Properties of the $\uparrow C$ operation)
The properties of the $\uparrow C$ operation are:

- *If K is prefix-closed, then so is $K^{\uparrow C}$.*

- *If $K \subseteq L_m(G)$ is $L_m(G)$-closed, then so is $K^{\uparrow C}$.*
- *In general, $\overline{K^{\uparrow C}} \subseteq (\overline{K})^{\uparrow C}$.*
- $(K_1 \cap K_2)^{\uparrow C} \subseteq K_1^{\uparrow C} \cap K_2^{\uparrow C}$.
- $(K_1 \cap K_2)^{\uparrow C} = (K_1^{\uparrow C} \cap K_2^{\uparrow C})^{\uparrow C}$.
- *If $K_1^{\uparrow C}$ and $K_2^{\uparrow C}$ are nonconflicting, then $(K_1 \cap K_2)^{\uparrow C} = K_1^{\uparrow C} \cap K_2^{\uparrow C}$.*
- $(K_1 \cup K_2)^{\uparrow C} \supseteq K_1^{\uparrow C} \cup K_2^{\uparrow C}$.

Definition 4.36 (Infimal prefix-closed controllable superlanguage)
We define:

$$K^{\downarrow C} := \bigcap_{L \in CC_{in}(K)} L$$

*the **infimal prefix-closed controllable superlanguage of** K. In the worst case $K^{\downarrow C} = M$, since $M \in CC_{out}(K)$. If K is controllable, then $K^{\downarrow C} = \overline{K}$.*

Observation 4.17 (Properties of the $\downarrow C$ operation)
The properties of the $\downarrow C$ operation are:

- $(K_1 \cap K_2)^{\downarrow C} \subseteq K_1^{\downarrow C} \cap K_2^{\downarrow C}$.
- *If K_1 and K_2 are nonconflicting, then $(K_1 \cap K_2)^{\downarrow C} = K_1^{\downarrow C} \cap K_2^{\downarrow C}$.*
- $(K_1 \cup K_2)^{\downarrow C} = K_1^{\downarrow C} \cup K_2^{\downarrow C}$.