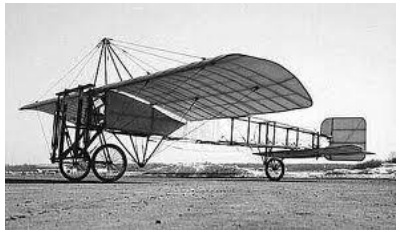
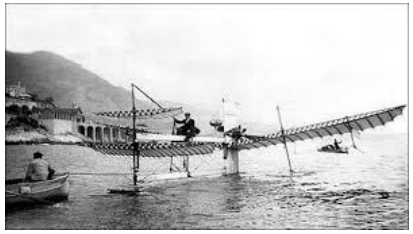
CentraleSupélec

Elements of Complex System Engineering

Antoine B. Rauzy
Department of Mechanical and Production Engineering (MTP)
Norwegian Science and Technology University (NTNU)
and
Chaire Blériot-Fabre
Centrale-Supélec, SAFRAN


CentraleSupélec

LECTURE 5. AUTOMATA

Notions:

- Automata
- Model-Checking

CentraleSupélec

LECTURE 5. PART 1. INTRODUCTION

CentraleSupélec

Objective of this lecture

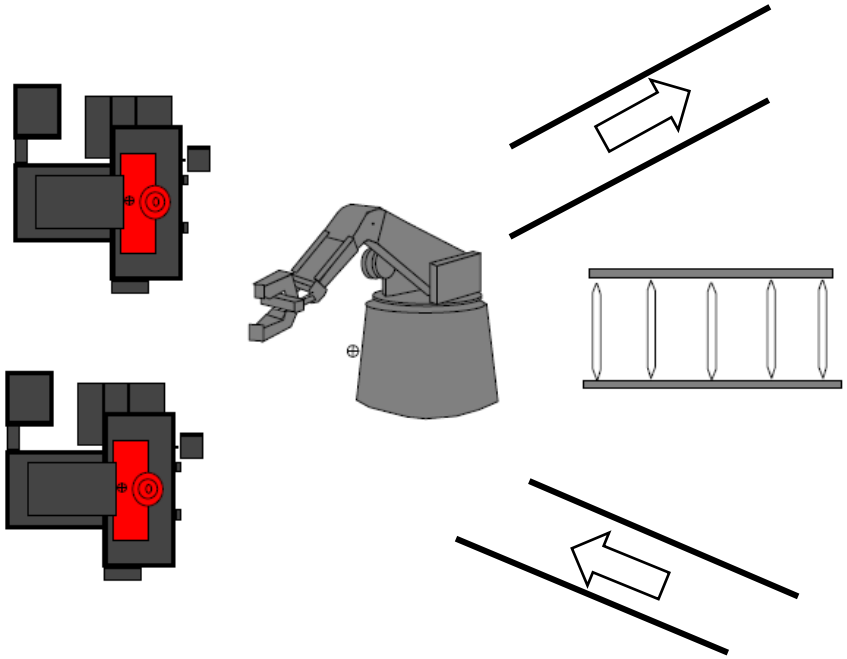
Automata play a central role in complex systems engineering. They are an essential **modeling tool** to describe the behavior of systems. They are also a fundamental **algorithmic tool** of software engineering.

The objective of this lecture is:

- to introduce the vocabulary and the main concepts of **automata theory**;
- to present the various types of automata that are used to describe the **behavior** of complex systems;
- to present their use in the context of **model-checking**;
- to discover a specific application of automata via the design of a **controller**.

Use Case: the Robot*

A production workshop is made of two drilling machines, a robot and a storage shelf (with a limited capacity). Products brought in and taken out of the workshop by means of two conveyor belts. The robot is in charge of taking items on the first conveyor belt, moving them to the drilling machine or the storage shelf and to put them on the second conveyor belt when they are treated.



We want:

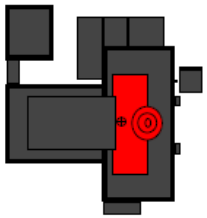
- to program (correctly) the robot;
- to check that the workshop reaches the expected production level;
- …

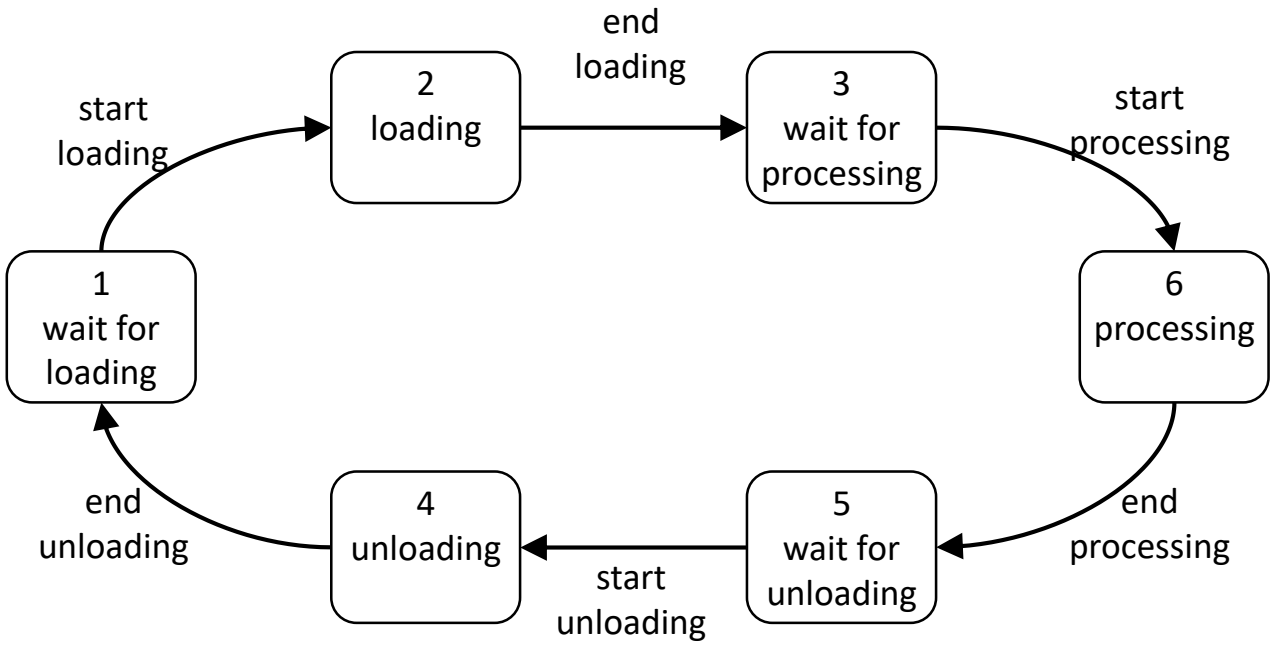
This could be done by long, costly (and potentially dangerous) tries and errors. But it is of course better to perform the tests virtually, i.e. to design a model.

(*) This case study is freely inspired from a course delivered by Prof. Edward Lin at Maryland University.

What to do?

At a high level of **abstraction**, the **behavior** of a system can almost always be described as a set of sequences of **actions** or **events** that modify the **state** of the system, i.e. eventually by a graph of states and actions.

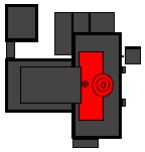
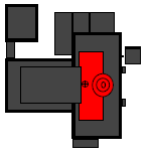
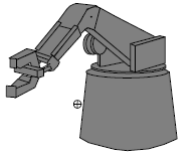
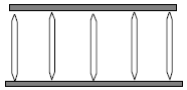




Such a graph is called a **(finite) state automaton**.
It can be more or less detailed depending on the needs.

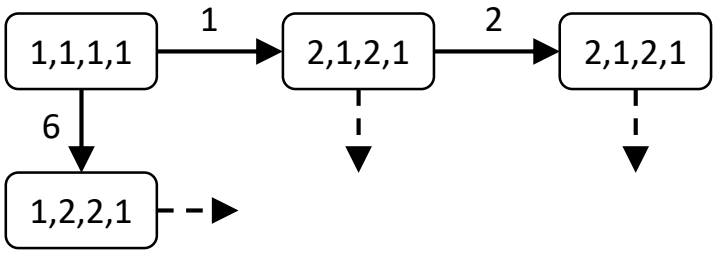
What to do?

Once the behavior of each component of the system described by a finite state automaton, it possible to **compose** these automata. The composition has to take into account that some actions are performed locally to a component and that some other are **synchronized**, i.e. **performed simultaneously** on two or more components.

				
star loading machine 1	start loading		take item on conveyor 1	
end loading machine 1	end loading		release item	
start processing machine 1	start processing			
…				

What to do?

The **global behavior** of the system is also a (finite) state automaton. It is the "**product**" of local (component) automata via the **synchronized actions**. For this reason, it is called a **synchronized product**.



1	start loading machine 1
2	end loading machine 1
	…
6	start loading machine 2

- The **states** of the synchronized product are **vectors of states** of compound automata. Its **transitions** are **vectors of synchronized actions**.
- The synchronized product can be **extremely large**. In practice, it is never built "by hand", and even not built at all, but just implicitly traversed.
- This mechanism is **recursive**: as the synchronized product of automata is itself an automaton, it is possible to compose it with other automata. This makes it possible to describe the behavior of hierarchical systems.

CentraleSupélec

What to do?

Once its global behavior described by an automaton, it is possible to check **behavioral properties** of the system. These behavioral properties are translated into **graph properties**, i.e. are checked by **graph algorithms**.

Behavioral properties	**Graph properties**
The production chain will never be blocked.	There is no state with no out-transition.
The shelf has sufficient capacity.	There is no state where the shelf is full having a out-transition adding an item to the shelf.
The second drilling machine will be eventually loaded.	There is no infinite path on which the second drilling machine is not loaded.
The system can always be go back to its initial state.	There is a path from any state to the initial state.
…	…

CentraleSupélec

LECTURE 5. PART 2. DEFINITIONS

CentraleSupélec

Finite State Automata

A **finite state automaton** is a quadruple ⟨S, E, T, i⟩ where:

- S is a finite set of symbols called **states**;
- E is a finite set of symbols called **labels**; E is called the **alphabet** of the automaton.
- T is a subset of the Cartesian product S×E×S. Elements of T are called **transitions**. A transition (s,e,t) is often denoted: $e: s \rightarrow t$ or $s \xrightarrow{e} t$. The states s are called respectively the **source** and the **target** of the transition ;
- i is at state of S called the **initial state**.

The finite state automaton ⟨S, E, T, i⟩ is **deterministic**:

$$\forall\, s, t, t' \in S, \forall e \in E, s \xrightarrow{e} t \in T \text{ et } s \xrightarrow{e} t' \in T \Rightarrow t = t'$$

In other word, if each state s and each event e, there is at most one out-transition of s labelled by e.

Automata that describe system behaviors are not always deterministic.

Reachable States

The previous definition makes it possible to define automata with states that are not reachable from the initial state.

The set of reachable states from a state i of a finite state automaton ⟨S, E, T, i⟩ (and more generally a graph G(S, T)) is the **least fixpoint** of the following equation:

$$Reachable(i) = \{i\} \cup \{t; s \to t \in T \wedge Reachable(s)\}$$

The algorithm we have seen last lecture is derived from the above equation.

In general, we are only interested by the automaton restricted to states that are reachable from the initial state. Let A: ⟨S, E, T, i⟩ be a finite state automaton. The restriction of A to the states that are reachable from A is the automaton A': ⟨S', E, T', i⟩ with:

- $S' = Reachable(i)$
- $T' = \left\{ s \xrightarrow{e} t \in T; s, t \in S' \right\}$

This may lead to remove labels that are not used in T'.

Synchronized Products

Let A_1: $\langle S_1, E_1, T_1, i_1 \rangle$, … A_k: $\langle S_k, E_k, T_k, i_k \rangle$ be k finite state automata and let V be a set of synchronization vectors, i.e. a subset of the **Cartesian product** $E_1 \cup \{\varepsilon\} \times \ldots \times E_k \cup \{\varepsilon\}$ (the symbol ε represents the absence of transition).

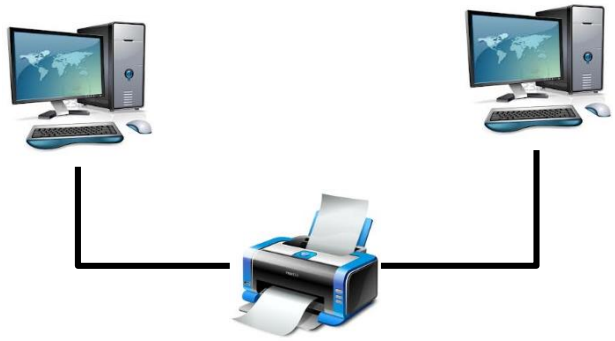
The **synchronized product** of A_1,…, A_k by V is the automaton A: $\langle S, E, T, i \rangle$ such that:

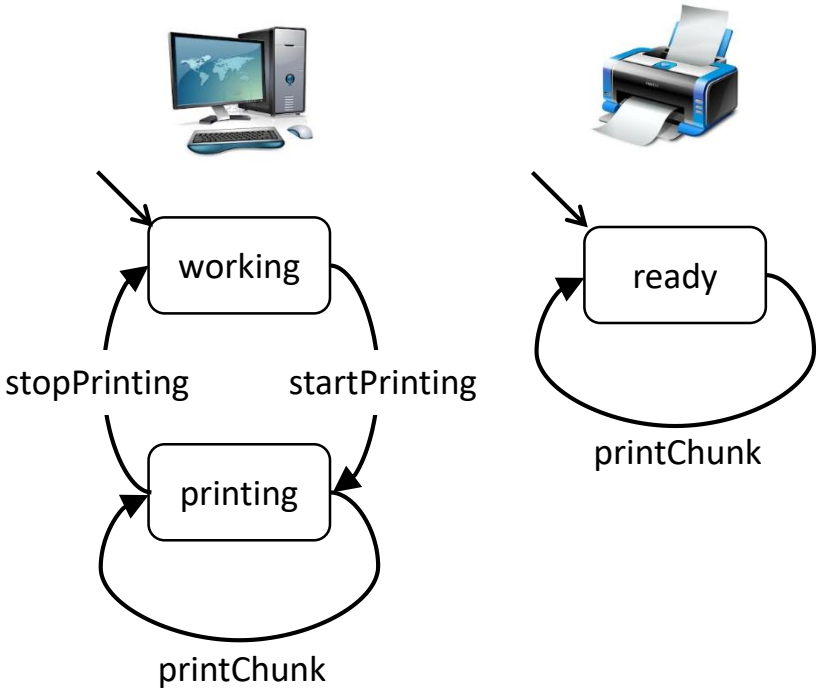
- $S = S_1 \times \cdots \times S_k$
- $E = E_1 \cup \{\varepsilon\} \times \cdots \times E_k \cup \{\epsilon\}$
- $T = \left\{ \begin{array}{c} s = \langle s_1 \times \cdots \times s_k \rangle \in S \\ s \xrightarrow{e} t;\ t = \langle t_1 \times \cdots \times t_k \rangle \in S \\ e = \langle e_1 \times \cdots \times e_k \rangle \in E \end{array} \right.$ where for each i = 1..k $\left. \begin{array}{l} s_i \xrightarrow{e_i} t_i \in T_i \text{ si } e_i \neq \epsilon \\ s_i = t_i \text{ si } e_i = \varepsilon \end{array} \right\}$
- $i = \langle i_1 \times \cdots \times i_k \rangle$

The restriction of the automaton to its reachable states makes indeed fully sense with the notion of synchronized product: the synchronized product is built by means of a fixpoint mechanism from its initial state.

Example : Shared Resources

Consider a network with two computers and a printer.
We want to model what can happen in this network.
A first (not very good) model could be:



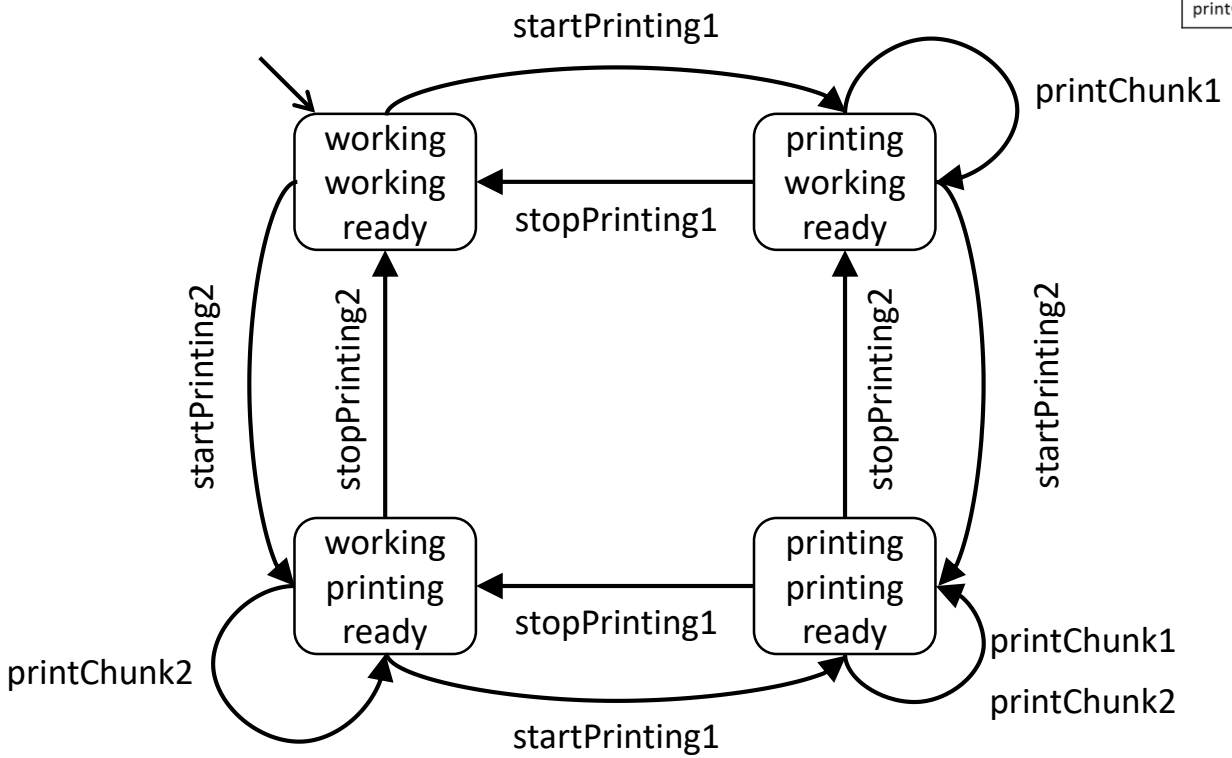


	computer 1	computer 2	printer
startPrinting1	startPrinting		
startPrinting2		startPrinting	
stopPrinting1	stopPrinting		
stopPrinting2		stopPrinting	
printChunk1	printChunk		printChunk
printChunk2		printChunk	printChunk

Example: Resource Sharing

	computer 1	computer 2	printer
startPrinting1	startPrinting		
startPrinting2		startPrinting	
stopPrinting1	stopPrinting		
stopPrinting2		stopPrinting	
printChunk1	printChunk		printChunk
printChunk2		printChunk	printChunk

The synchronized product:



CentraleSupélec

LECTURE 5. PART 3. INTERLUDE

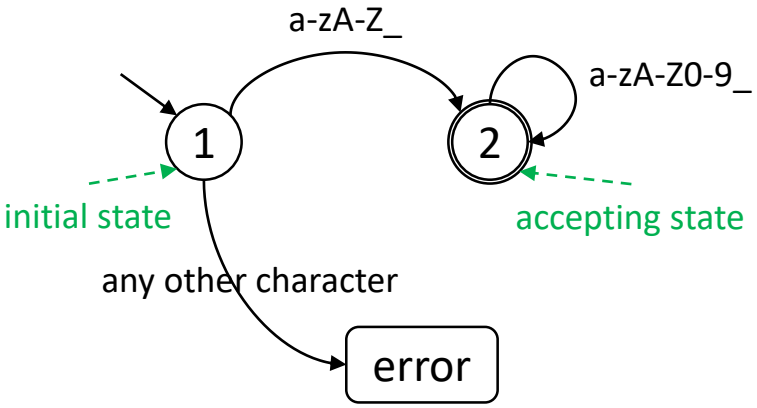
Parsing Automata

Automata play a central role in algorithms that read a text (**parsing algorithms**).

E.g. an identifier is a letter or an underscore followed by any number of letters, digits or underscores.

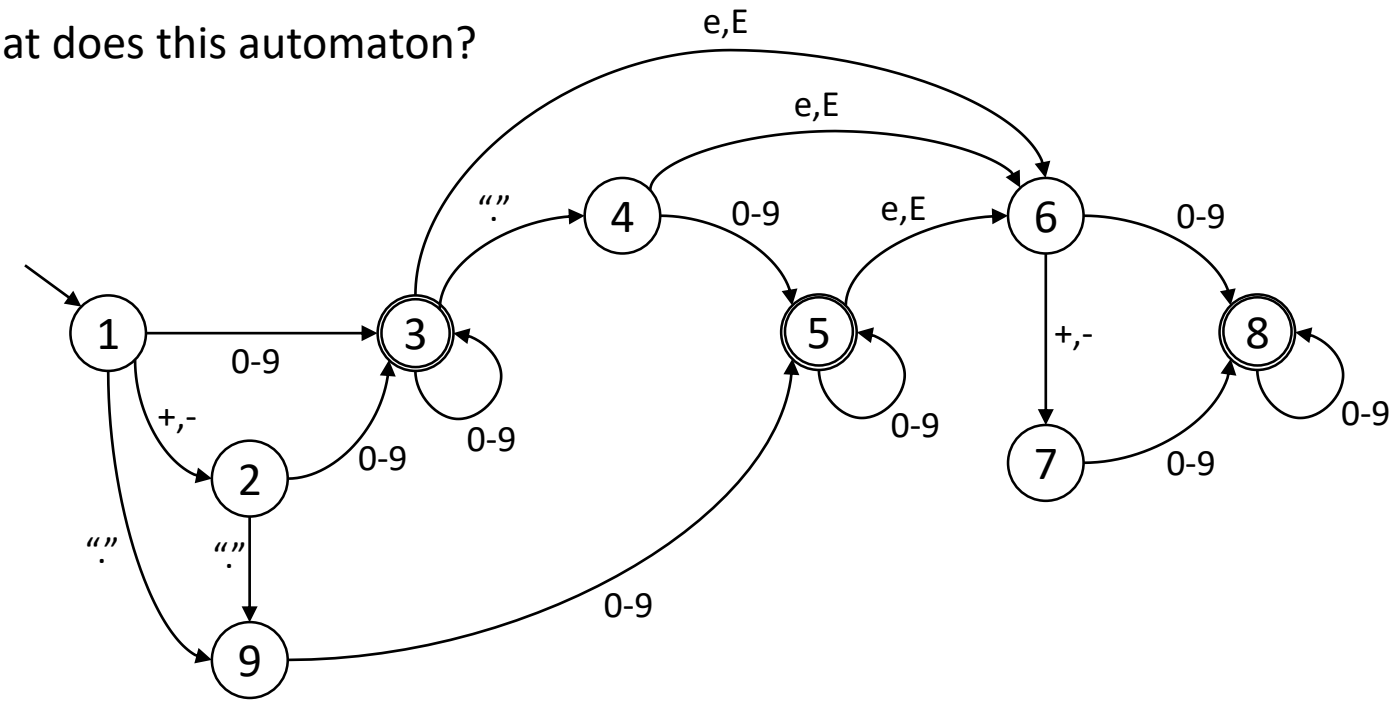
Rational expression: [a-zA-Z_] [a-zA-Z0-9_]*

Automaton:



Parsing Automata

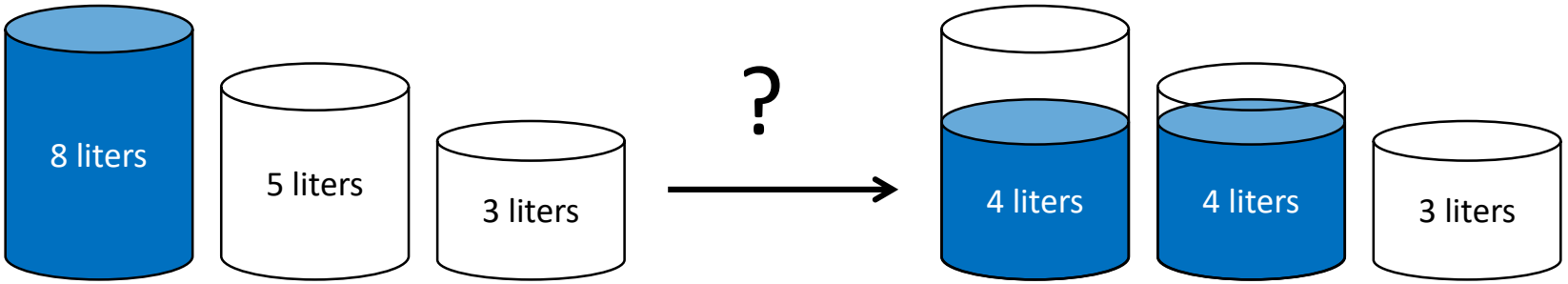
What does this automaton?



Find “words” recognized by this automaton (at least one per non looped path).

The three buckets

Automata and synchronized products are hidden in many places, including into famous puzzles for scholars: Assume we have three buckets that can contain respectively 8, 5 and 3 liters. Assume moreover that the 8 liters one is full at the beginning. Is there a way to transfer exactly 4 liters from to the 5 liters one, without using any other device than the three buckets?



Three Buckets Problem: Model

The automaton representing a bucket containing n liters is made of n+1 states (0, 1, 2,…, n) and four series of transitions:

Name	Source state	Target state
get_k (k=1,2,..,n)	$s \leq n-k$	s+k
put_k (k=1,2,..,n)	$s \geq k$	s-k
empty_k (k=1,2,..,n)	k	0
fill_k (k=1,2,..,n)	s = n-k	n

Synchronization vectors are then obtained by coupling local transitions “put_k” and “fill_k” on the one hand, and “get_k” and “empty_k”:

Name	S1 (8l)	S2 (5l)	S3 (3l)
…			
PF_3	put_3	fill_3	ε
EG_3	empty_3	get_3	ε
…			

$$422 \xrightarrow{PF\_3} 152$$

$$323 \xrightarrow{EG\_3} 053$$

Three Buckets Problem: 1st Solution

state	successors			
800	350	503		
503	053	800	530	
530	350	233	800	503
233	053	503	530	251
251	053	701	233	350
701	251	503	800	710
710	350	413	800	701
413	053	503	710	440
440	350	143	800	413
143	053	503	440	152
152	053	602	143	350
602	152	503	800	620
620	350	323	800	602
323	053	503	620	350
053	503	350		
350	053	800	323	

800
↓ P_F3
503
↓ _GE3
530
↓ P_F3
233
↓ _FP2
251
↓ GE_5
701
↓ _GE1
710
↓ P_F3
413
↓ _GE3
440

Three Buckets Problem: 2nd Solution

state	successors			
800	350	503		
503	053	800	530	
530	350	233	800	503
233	053	503	530	251
251	053	701	233	350
701	251	503	800	710
710	350	413	800	701
413	053	503	710	440
440	350	143	800	413
143	053	503	440	152
152	053	602	143	350
602	152	503	800	620
620	350	323	800	602
323	053	503	620	350
053	503	350		
350	053	800	323	

800 ↓ 350 ↓ 323 ↓ 620 ↓ 602 ↓ 152 ↓ 143 ↓ 440

CentraleSupélec

LECTURE 5. PART 4. PROPERTIES

CentraleSupélec

Safety Properties

It is possible to write associate Boolean properties with states. For instance:

- A **deadlock** is a state with no out-transition (sink nodes).
- A dangerous state of the network with several computers and a printer is a state in which several computers print at the same time.

Safety properties are properties that can be expressed as the reachability of (at least one state) of a set of states having a certain Boolean property. For instance:

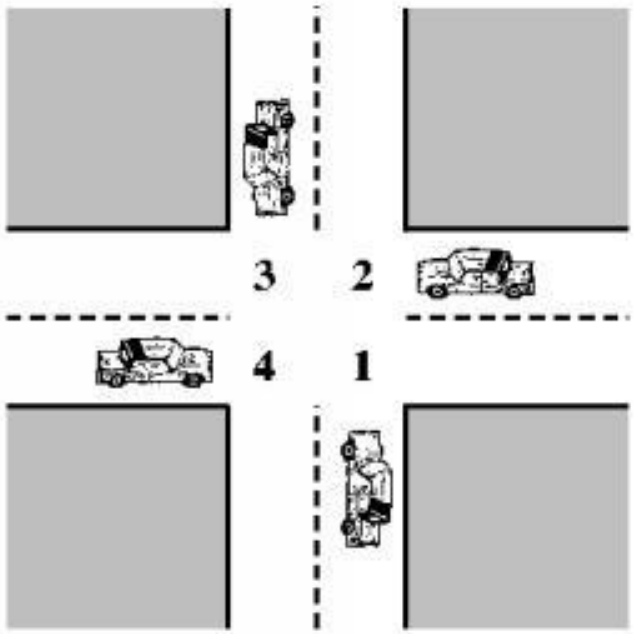
- The existence of a reachable deadlocks.
- The existence of reachable dangerous states.

Safety property play a very important role. First, many expected or unwanted properties of systems can be expressed as safety properties. Second, they are relatively easy to check (see lecture on graphs).

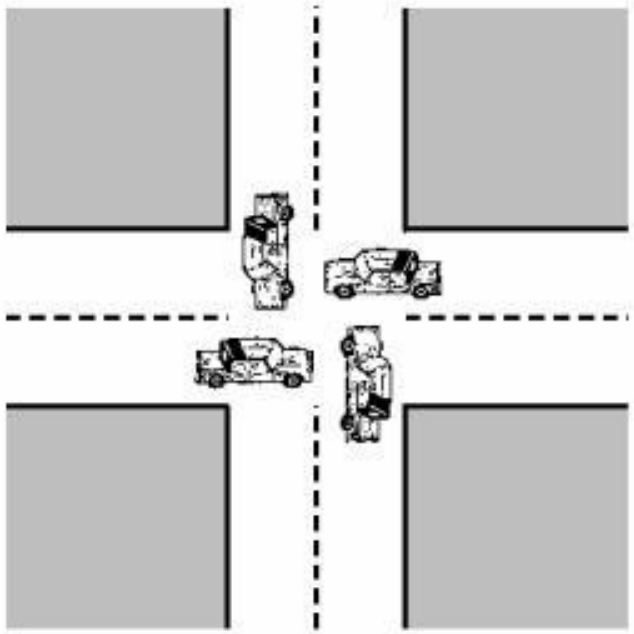
It is sometimes possible (not always) to calculate **co-reachable** states, i.e. states s such that a given state t (or set of states T) is reachable from s. The calculation of co-reachable states make it possible to check interesting properties such as the capacity of the system to come back to its initial state.

CentraleSupélec

Example of Deadlocks



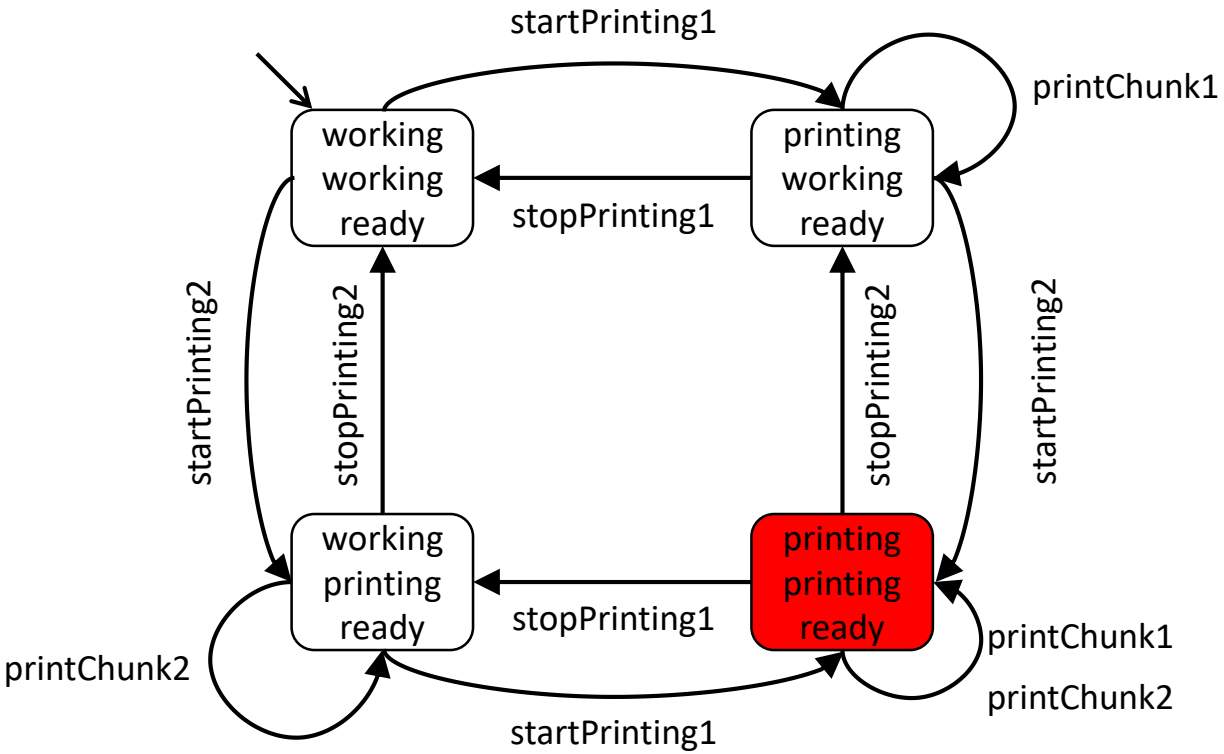
(a) Deadlock possible



(b) Deadlock

Reachable Unsafe States

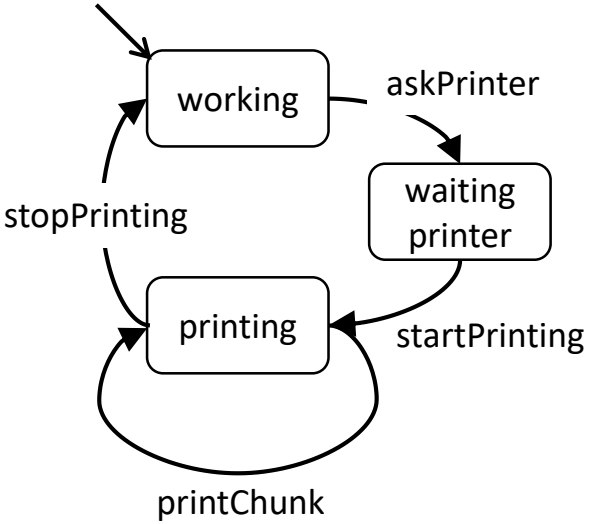
Synchronized product

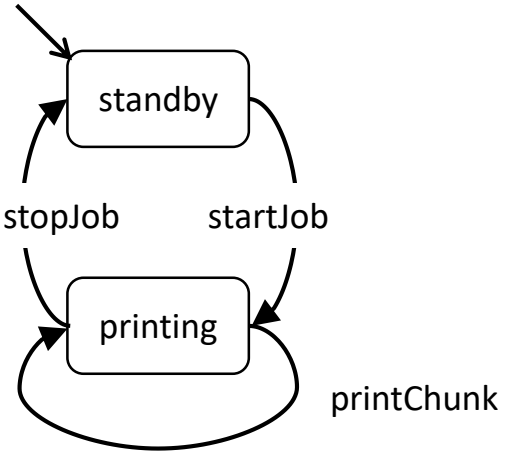


There is a reachable state in which both computer 1 and computer 2 are printing.

Resource Sharing

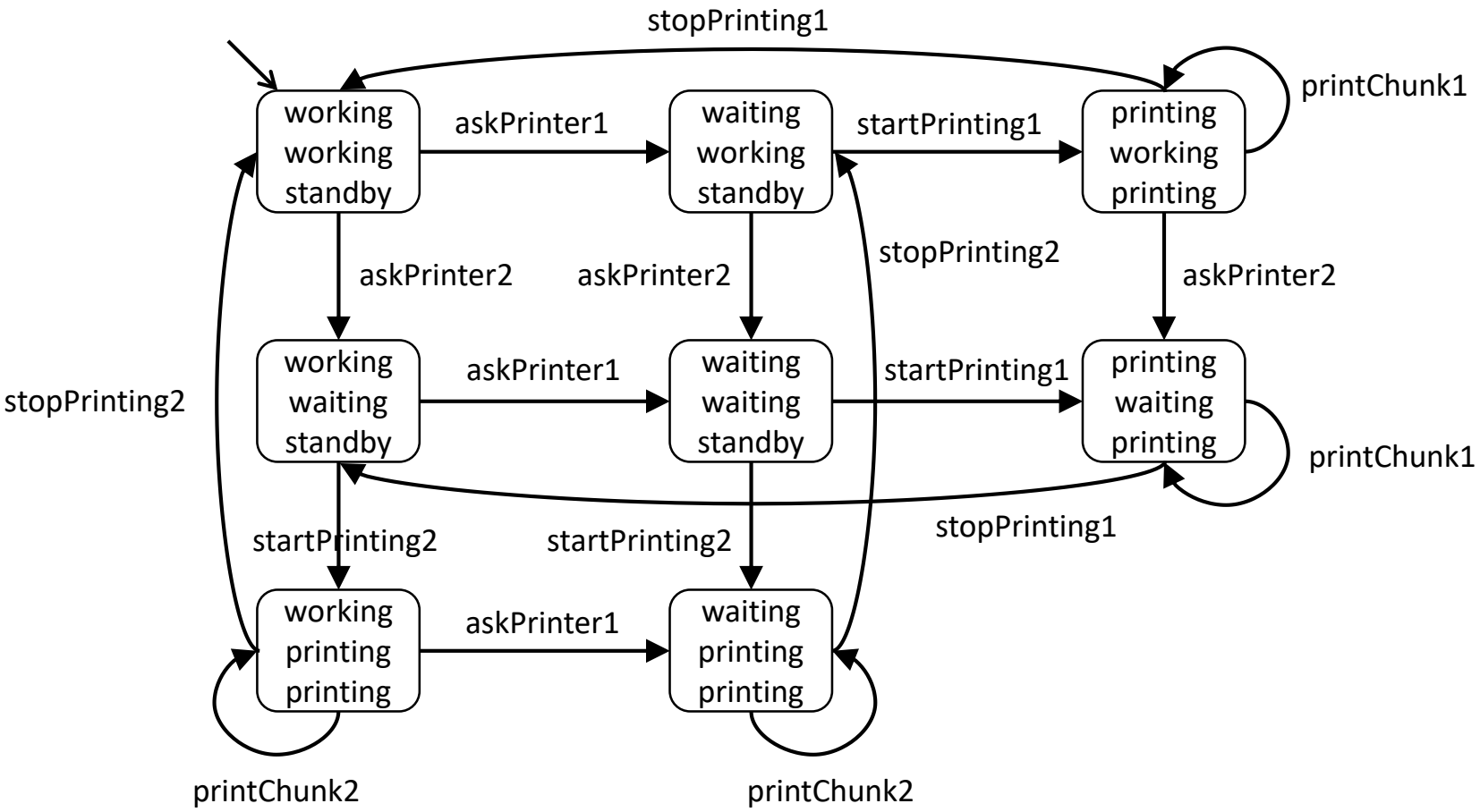
A slightly more realistic model





	computer 1	computer 2	printer
askPrinter1	askPrinter		
askPrinter2		askPrinter	
startPrinting1	startPrinting		startJob
startPrinting2		startPrinting	startJob
stopPrinting1	stopPrinting		stopJob
stopPrinting2		stopPrinting	stopJob
printChunk1	printChunk		printChunk
printChunk2		printChunk	printChunk

Resource Sharing



In this new model, there is no unsafe state

CentraleSupélec

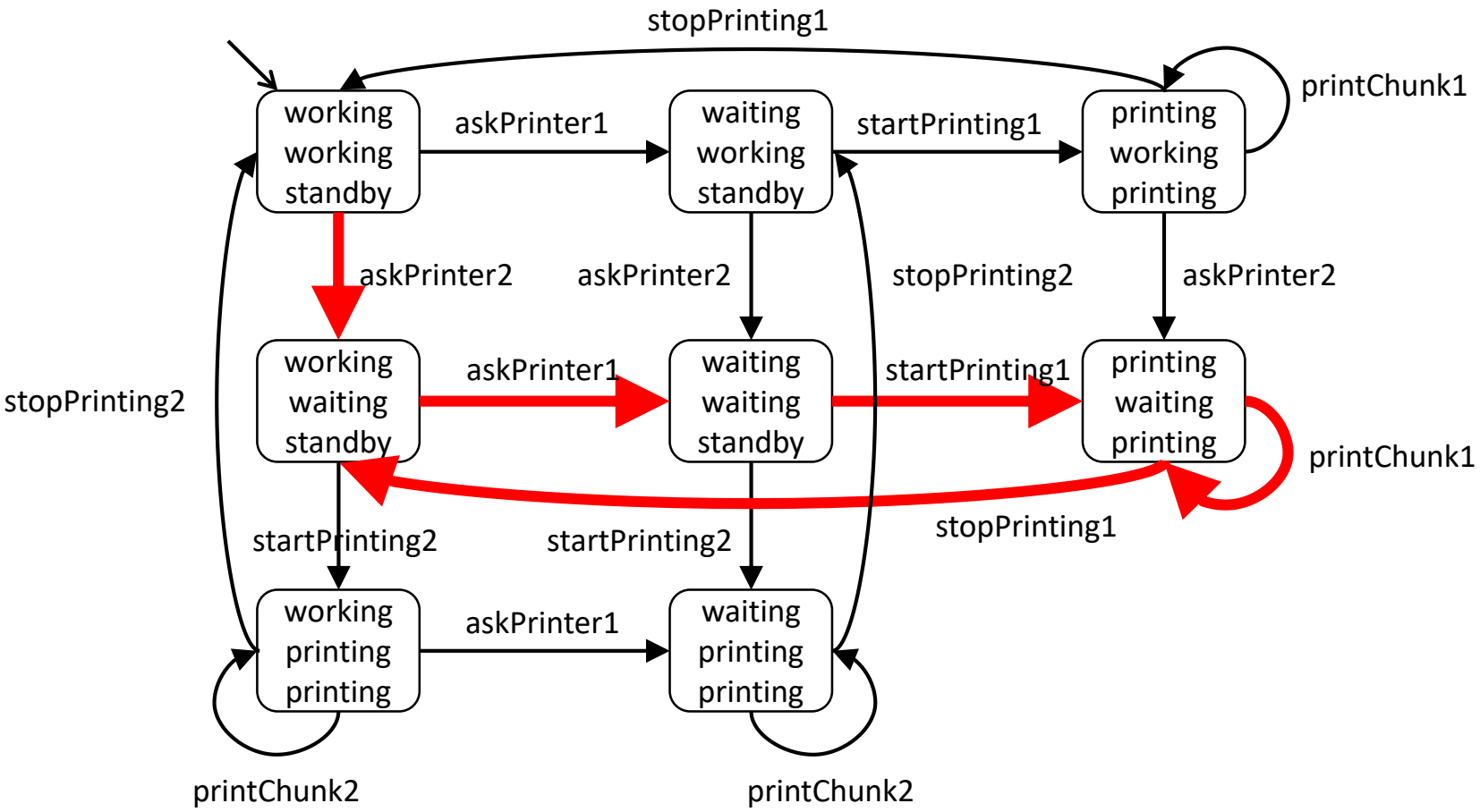
Liveness Properties

Liveness properties are a second category of interesting properties. A liveness property asserts that for all reachable state s, all possible executions starting from s will **eventually** (i.e. within a finite number of steps) go through a state t verifying a certain Boolean property P. In other words, there is no **circuit** going through states that do not verify the property P. For instance:

- If one of the computer is waiting for the printer, it will eventually get it.
- If an alarm is raised, it will eventually be handled.
- If a client enters into the queue (e.g. of a call center), his demand will be eventually treated.

Liveness properties play also an important role in model and system verification/validation.
They are however more **costly** to check than safety properties. Nevertheless, it exists relatively efficient data structures and algorithms to do so.

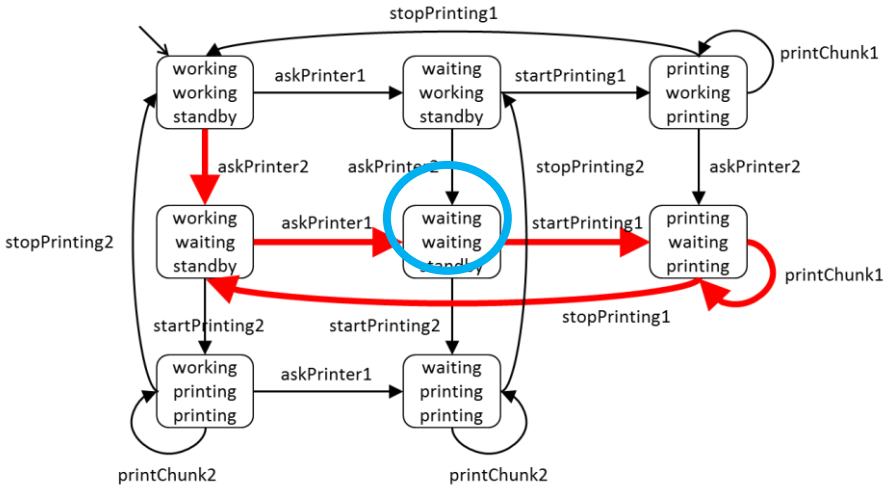
Resource Sharing



There is an execution in which the second computer waits forever the printer.

Fairness

Liveness properties are intimately related to time: they are always in the form “in the future, something expected will necessarily happen”.



The infinite loop detected in the network assumes that:

- the first computer prints an infinite document or that it starts immediately printing a new document after the printing of current one is completed;
- its requests are systematically given the highest priority.

These hypotheses may be not very realistic…

This is the reason why liveness properties are often studied under **fairness** conditions or assumptions.

An execution involving a set of components is fair if none of these components stays idle infinitely. Many real life systems are verify liveness properties only if under fairness assumptions.

Beyond Safety and Liveness

There exists properties that are neither safety nor liveness properties.

For instance, given two automata A and B, are these two automata equivalent, i.e. does any execution of A correspond to an execution of B (i.e. is labelled with the same actions).

Such a property can be used to check that a system meets its specification.

The general principle to verify such a property is to build (or at least to traverse) the synchronized product of A and B and to check a safety (or a liveness) property on the product. Unfortunately, this turns out to be extremely computationally costly because the product can be astronomically large.

CentraleSupélec

Temporal Logics

Temporal logics are often used to describe behavioral properties of the system under study. Temporal logic formulas characterize sets of executions of the system under study (i.e. paths in the automaton).

We assume given (implicitly or explicitly) a graph/automaton G : (S, T), $T \subseteq S \times S$. Moreover, we assume atomic Boolean properties $P = \{p_1, \ldots, p_n\}$ that either true or false in each state of S. G and P are called a **Kripke structure**.

The set Φ of state properties is defined as follows.

- Φ = true| false | p_i | $\Phi \vee \Phi$ | $\Phi \wedge \Phi$ | $\neg\Phi$ | AΦ | EΦ

The formula Af (respectively Ef) characterizes the states s such that all paths (respectively it exists a path) starting from s that verify the path property f.

The set ϕ of path properties is defined as follows.

- ϕ = Φ | | $\Phi \vee \Phi$ | $\Phi \wedge \Phi$ | $\neg\Phi$ | XΦ | FΦ | GΦ | ΦUΦ

Xf characterizes the paths starting from a state s whose immediate successor verifies the state property f.

Ff characterizes the paths starting from a state s whose at least one successors verifies the state property f.

And so on…

CentraleSupélec

LECTURE 5. PART 5.
BEYOND STATE AUTOMATA

CentraleSupélec

Beyond State Automata

Synchronized products describe the behavior of systems in a **implicit** and **hierarchical** way.

There exist actually many formalisms to describe, to specify and to study behaviors by means of automata. These formalisms can be characterized along several directions:

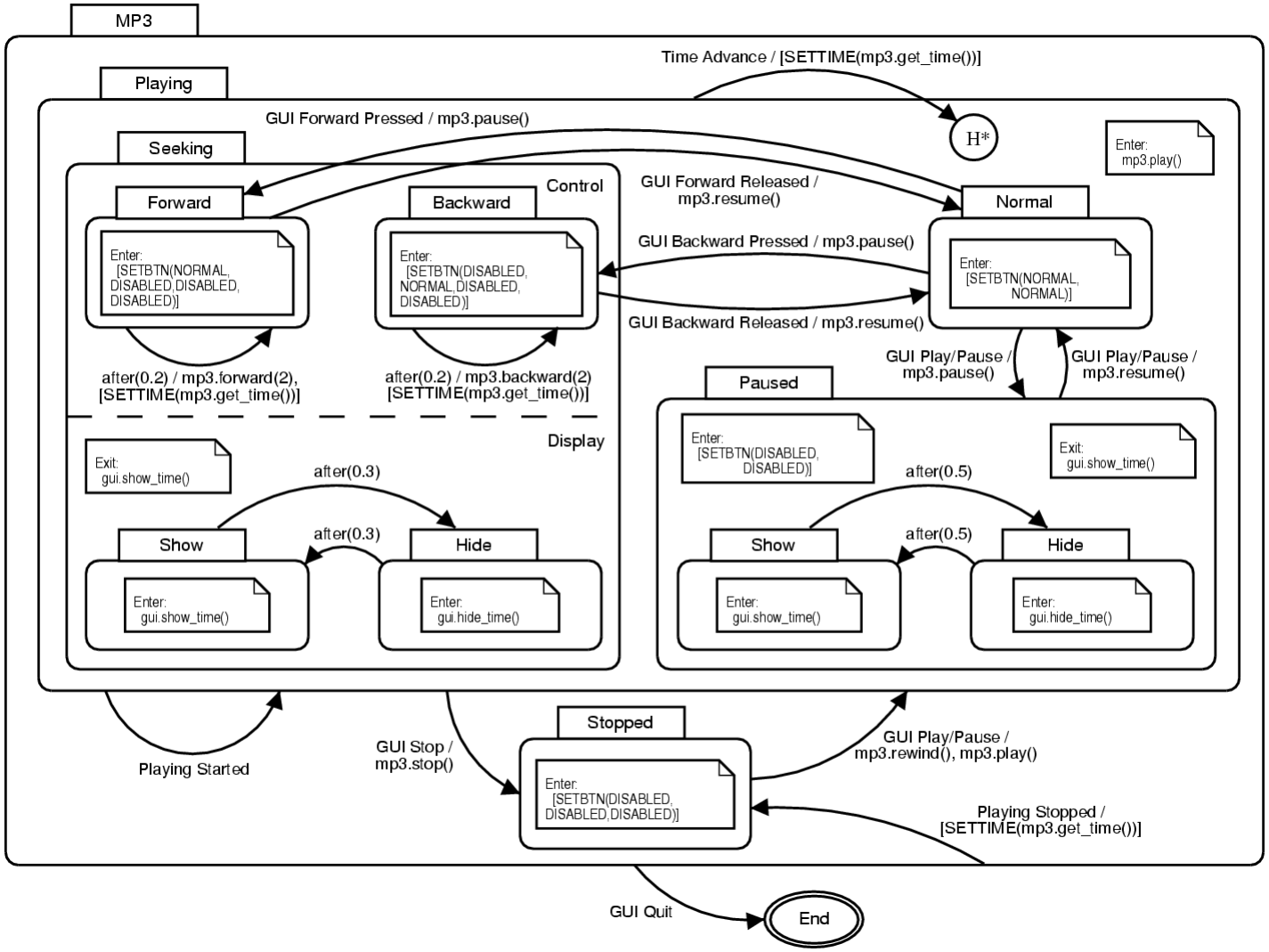
- The **representation** of the state space (**explicit or implicit**),
- Description of **finite** or **infinite behaviors**,
- The means provided for the **hierarchical composition**,
- Whether the accent is put on **states** or **events**
- The **graphical representation** of models.

It is also possible to associate **delays** and **probabilities** to transitions.

Last, it is possible to use automata to generate embedded software (controller). This technique is now well mastered and more and more applied in industry.

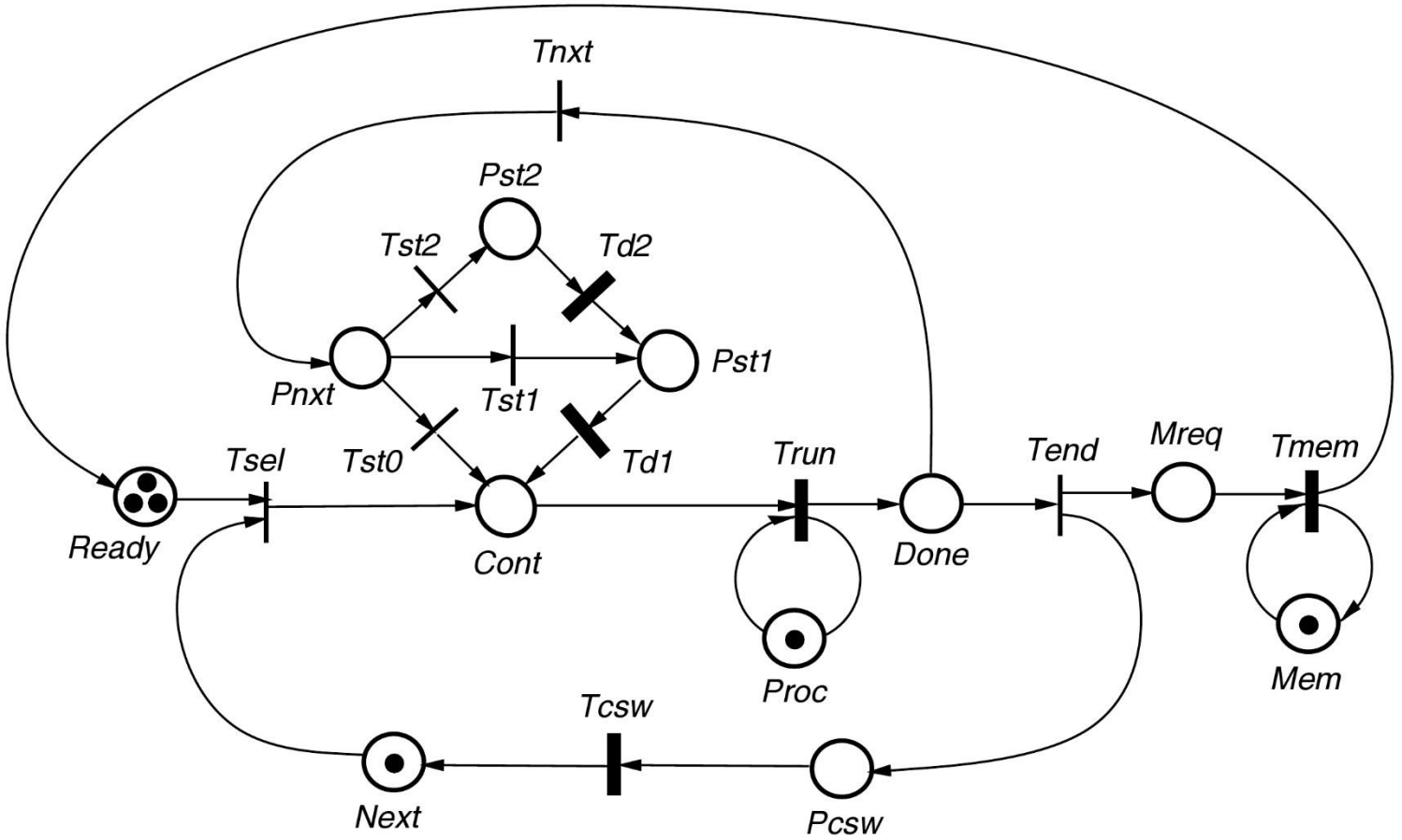
CentraleSupélec

State-charts



CentraleSupélec

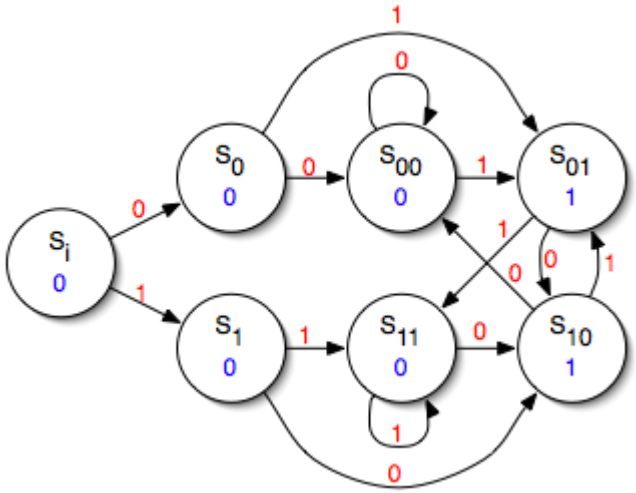
Petri Nets



Moore Machines

A Moore machine is made of:

- A finite set of states Q;
- An initial state i, member of Q;
- A finite set of symbols A, called the input alphabet;
- A finite set of symbols B, called the output alphabet;
- A transition function δ: $Q \times A \rightarrow Q$;
- An output function : $Q \rightarrow B$.



CentraleSupélec

LECTURE 5. PART 6.
BEYOND STATE AUTOMATA

CentraleSupélec

Modeling Language(s)

Except for toy examples, it is not possible to build the synchronized product by hand and in many cases not possible to build it at all. Properties are thus in general verified by exploring the synchronized product without building it (and a fortiori representing it graphically).

Therefore, we need languages to describe automata and synchronized products and a tool to verify properties.

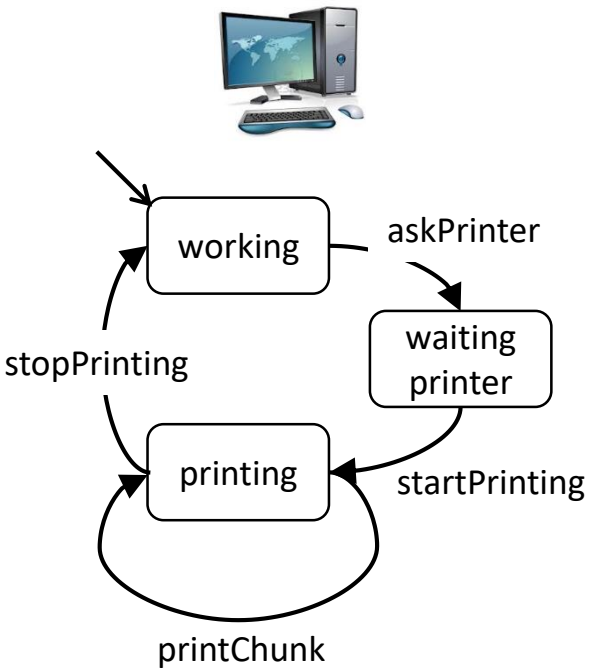
A minimal language to do so should contain:

- Constructs to describe automata, i.e. states, events and transitions;
- Constructs to describe synchronized products, i.e. composed automata and synchronization vectors;
- Constructs to describe models made of automata and synchronization models.

A Textual Format for Automata

Automaton ::= “automaton” Identifier State* Event* Transition* “end”
State ::= “state” Identifier
Event ::= “event” Identifier
Transition ::= “transition” Identifier Identifier Identifier

```
automaton Computer
    state working
    state waiting
    state printing
    event askPrinter
    event startPrinting
    event printChunk
    event stopPrinting
    transition askPrinter working waiting
    transition startPrinting waiting printing
    transition printChunk printing printing
    transition stopPrinting printing working
end
```



A Textual Format for Products

Product ::= “product” Identifier ComposedAutomaton* SynchronizationVector* “end”
ComposedAutomaton ::= Identifier Identifier
SynchronizationVector ::= “event” Identifier “(“ LocalEvent (“,” LocalEvent)* “)”
LocalEvent ::= Identifier “.” Identifier

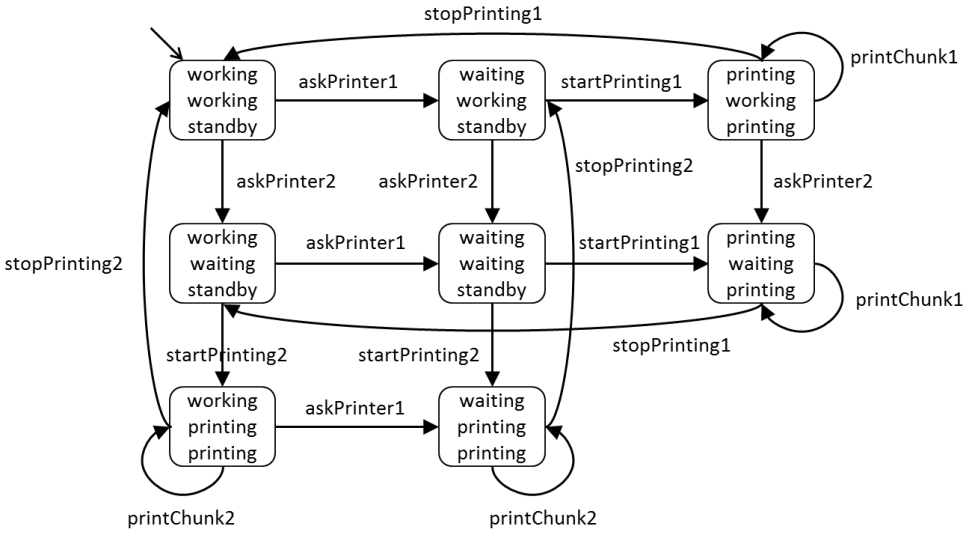
	computer 1	computer 2	printer
askPrinter1	askPrinter		
askPrinter2		askPrinter	
startPrinting1	startPrinting		startJob
startPrinting2		startPrinting	startJob
stopPrinting1	stopPrinting		stopJob
stopPrinting2		stopPrinting	stopJob
printChunk1	printChunk		printChunk
printChunk2		printChunk	printChunk

```
product Network
   Computer C1
   Computer C2
   Printer P
   event askPrinter1 (C1.askPrinter)
   event askPrinter2 (C2.askPrinter)
   event startPrinting1 (C1.startPrinting, P.startJob)
   event startPrinting2 (C2.startPrinting, P.startJob)
   event stopPrinting1 (C1.stopPrinting, P.stopJob)
   event stopPrinting2 (C2.stopPrinting, P.stopJob)
   event printChunk1 (C1.printChunk, P.printChunk)
   event printChunk2 (C2.printChunk, P.printChunk)
end
```

A Textual Format for Models

Model ::= “model” Identifier (Automaton | Product)* “end”

```
model PrintingSystem
    automaton Computer
        …
    end
    automaton Printer
        …
    end
    product Network
        …
    end
end
```



CentraleSupélec

checky.py

The Python program "checky.py" reads a model in a text file and builds the synchronized product.

As most, if not all, programs designed for this course, it is structured as follows.

1. Definition of data structures. In this lecture case, classes automata and products.
2. Classes that makes it possible to read (Reader) and write (Writer) data structures into text files as well as results of calculations.
3. Classes that implement the calculation(s) of interest (Calculator).
4. Main part of the program where the classes are instanced and the calculations performed.

You can run the program either in a command shell or using IDLE.

CentraleSupélec

LECTURE 5. PART 6.
WRAP-UP & ASSIGNMENT

CentraleSupélec

Wrap-Up

- **State automata** (finite or not) are an essential tool for the behavior modeling of systems.
- State automata are used in many other contexts, like to parse string of characters in a broad sense (programs, web pages, DNA…).
- **Behavioral properties** of systems are checked via **state and path properties** of graphs. This technique is called **model-checking**.
- It is also possible to specify the expected behavior of IT systems (programs) via state automata. The concrete system can then be generated directly from the automata.

CentraleSupélec

Assignment

We consider a production system with two production units working in parallel. Each production unit alternates between operation and maintenance phases. Both units are required to be in operation for the system as a whole to be in operation. The maintenance of units is performed by a maintenance crew. The maintenance crew can only maintain one unit at a time. Units may fail while in operation. If a unit is failed and put it maintenance, then it will be repaired at the end of the maintenance.

1. Describe this production system as a synchronized product.
2. Calculate the synchronized product (by means of the checky.py script).
3. Check whether your system can have unexpected behaviors by describing and verifying properties of the synchronized product.

CentraleSupélec

Recommend Readings

Reference book on Automata and Model Checking:

There exists a vast scientific and technical literature on different types automata. E.g.

E.M. Clarke, O. Grumberg , and D.A. Peled. *Model Checking*. MIT Press , Feb. 2000. ISBN-10: 0262032708, ISBN-13: 978-0262032704.

David Harel and Michal Polti. *Modeling Reactive Systems With Statecharts: The Statemate Approach*. McGraw-Hill Inc.,US (1 août 1998). ISBN-10: 0070262055. ISBN-13: 978-0070262058

Tadao Murata. Petri Nets: Properties, Analysis and Applications. *Proceedings of the IEEE*, 77(4):541--580, April 1989.

CentraleSupélec

Louis Charles Joseph Blériot (1872 -1936) is an airplane designer and one of the pioneer pilot of French aviation. He has been the first to cross the channel on July the 25th onboard of the Blériot XI.
He graduated from Ecole Centrale de Paris

Henri Marie Léonce Fabre (1882 -1984) is a French engineer and pilot. He invented the seaplane in 1910.
He graduated from Supélec.