# Initiation à la Vérification
# Basics of Verification

Stefan Schwoon
credits for slides: Paul Gastin, Javier Esparza, Keijo Heljanko

M1 MPRI, 2022/2023

# Organisation

## Timetable

- Course: Friday 10:45 – 12:45 (Stefan Schwoon)
- Exercises: Friday 8:30 – 10:30 (Stéphane Le Roux)

## Controls (to be confirmed)

- Homework 1 (1/6)
- Midterm exam (1/3)
- Homework 2 (1/6)
- Final exam (1/3)

Second session: Homeworks + Replacement exam

# Need for formal verifications methods

### Critical systems

- Transport
- Energy
- Medicine
- Communication
- Finance
- Embedded systems
- ...

# Disastrous software bugs

## Mariner 1 probe, 1962

See http://en.wikipedia.org/wiki/Mariner_1

- Destroyed 293 seconds after launch
- Missing hyphen in the data or program? No!
- Overbar missing in the mathematical specification:
  $\overline{\dot{R}_n}$: $n$th smoothed value of the time derivative of a radius.
  Without the smoothing function indicated by the bar, the program treated normal minor variations of velocity as if they were serious, causing spurious corrections that sent the rocket off course.

# Disastrous software bugs

## Ariane 5 flight 501, 1996

See `http://en.wikipedia.org/wiki/Ariane_5_Flight_501`

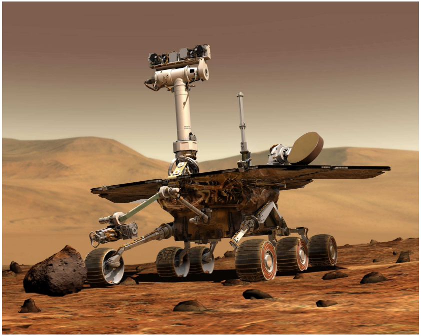
- Destroyed 37 seconds after launch (cost: 370 millions dollars).
- data conversion from a 64-bit floating point to 16-bit signed integer value caused a hardware exception (arithmetic overflow).
- Efficiency considerations had led to the disabling of the software handler (in Ada code) for this error trap.
- The fault occured in the inertial reference system of Ariane 5. The software from Ariane 4 was re-used for Ariane 5 without re-testing.
- On the basis of those calculations the main computer commanded the booster nozzles, and somewhat later the main engine nozzle also, to make a large correction for an attitude deviation that had not occurred.
- The error occurred in a realignment function which was not useful for Ariane 5.

# Disastrous software bugs

## Spirit Rover (Mars Exploration), 2004

See http://en.wikipedia.org/wiki/Spirit_rover

- Landed on January 4, 2004.
- Ceased communicating on January 21.
- Flash memory management anomay:
  too many files on the file system
- Resumed to working condition on February 6.



# Disastrous software bugs

## Other well-known bugs

- Therac-25, at least 3 death by massive overdoses of radiation.
  Race condition in accessing shared resources.
  See `http://en.wikipedia.org/wiki/Therac-25`
- Electricity blackout, USA and Canada, 2003, 55 millions people.
  Race condition in accessing shared resources.
  See `http://en.wikipedia.org/wiki/Northeast_Blackout_of_2003`
- Pentium FDIV bug, 1994.
  Flaw in the division algorithm, discovered by Thomas Nicely.
  See `http://en.wikipedia.org/wiki/Pentium_FDIV_bug`
- Needham-Schroeder, authentication protocol based on symmetric encryption.
  Published in 1978 by Needham and Schroeder
  Proved correct by Burrows, Abadi and Needham in 1989
  Flaw found by Lowe in 1995 (man in the middle)
  Automatically proved incorrect in 1996.
  See `http://en.wikipedia.org/wiki/Needham-Schroeder_protocol`

# Formal verifications methods

**Complementary approaches**

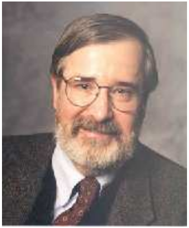
- Theorem prover
- Model checking
- Static analysis
- Test

# What does “Model-Checking” mean?

# What does “Model-Checking” mean?

# Model Checking

- Purpose 1: automatically finding software or hardware bugs.
- Purpose 2: prove correctness of abstract models.
- Should be applied during design.
- Real systems can be analysed with abstractions.



E.M. Clarke

E.A. Emerson

J. Sifakis

Prix Turing 2007.

# Model Checking

## 3 steps

- Constructing the model $M$ (transition systems)
- Formalizing the specification $\varphi$ (temporal logics)
- Checking whether $M \models \varphi$ (algorithmics)

## Main difficulties

- Size of models (combinatorial explosion)
- Expressivity of models or logics
- Decidability and complexity of the model-checking problem
- Efficiency of tools

## Challenges

- Extend models and algorithms to cope with more systems.
  Infinite systems, parameterized systems, probabilistic systems, concurrent systems, timed systems, hybrid systems, ...
- Scale current tools to cope with real-size systems.
  Needs for modularity, abstractions, symmetries, ...

# References

## Bibliography


[1] Christel Baier and Joost-Pieter Katoen.
*Principles of Model Checking*.
MIT Press, 2008.

[2] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, Ph. Schnoebelen.
*Systems and Software Verification. Model-Checking Techniques and Tools*.
Springer, 2001.

[3] E.M. Clarke, O. Grumberg, D.A. Peled.
*Model Checking*.
MIT Press, 1999.

[4] Z. Manna and A. Pnueli.
*The Temporal Logic of Reactive and Concurrent Systems: Specification*.
Springer, 1991.

[5] Z. Manna and A. Pnueli.
*Temporal Verification of Reactive Systems: Safety*.
Springer, 1995.

# Outline

Introduction

2. Models
   - Transition systems
   - . . . with variables
   - Concurrent systems
   - Synchronization and communication

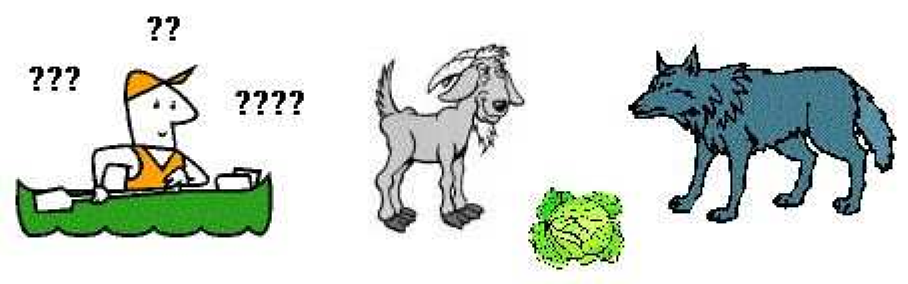
Specifications

Linear Time Specifications

Branching Time Specifications

# Constructing the model

**Example: Men, Wolf, Goat, Cabbage**




**Model = Transition system**

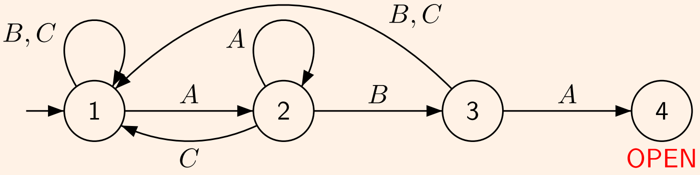
- State = who is on which side of the river
- Transition = crossing the river
- Specification
  Safety: Never leave WG or GC alone
  Liveness: Take everyone to the other side of the river.

# Transition system or Kripke structure

**Definition: TS** $M = (S, \Sigma, T, I, \mathrm{AP}, \ell)$

- $S$: set of states (finite or infinite)
- $\Sigma$: set of actions
- $T \subseteq S \times \Sigma \times S$: set of transitions
- $I \subseteq S$: set of initial states
- $\mathrm{AP}$: set of atomic propositions
- $\ell : S \to 2^{\mathrm{AP}}$: labelling function.

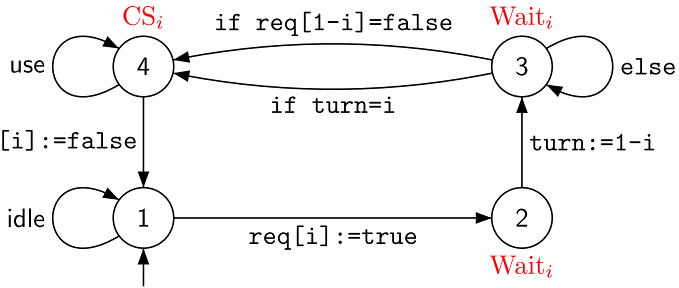
**Example: Digicode ABA**




Every discrete system may be described with a TS.

# Peterson's algorithm (1981)

```
Process i:
   loop forever
      req[i] := true; turn := 1-i
      wait until (turn = i or req[1-i] = false)
      Critical section
      req[i] := false
```




**Exercise:**

- Draw the concrete TS assuming the first two assignments are atomic.
- Is the algorithm still correct if we swape the first two assignments?

# Description Languages

**Pb: How can we easily describe big systems?**

**Description Languages (high level)**

- Programming languages
- Boolean circuits
- Modular description, e.g., parallel compositions
  problems: concurrency, synchronization, communication, atomicity, fairness, ...
- Petri nets (intermediate level)
- Transition systems (intermediate level)
  with variables, stacks, channels, ...
  synchronized products
- Logical formulae (low level)

**Operational semantics**

High level descriptions are translated (compiled) to low level (infinite) TS.

# Transition systems with variables

**Definition: TSV** $M = (S, \Sigma, \mathcal{V}, (D_v)_{v\in\mathcal{V}}, T, I, \mathrm{AP}, \ell)$

- $\mathcal{V}$: set of (typed) variables, e.g., boolean, [0..4], ...
- Each variable $v \in \mathcal{V}$ has a domain $D_v$ (finite or infinite)
- Guard or Condition: unary predicate over $D = \prod_{v\in\mathcal{V}} D_v$
  Symbolic descriptions: $x < 5$, $x + y = 10$, ...
- Instruction or Update: map $f : D \to D$
  Symbolic descriptions: $x := 0$, $x := (y+1)^2$, ...
- $T \subseteq S \times (2^D \times \Sigma \times D^D) \times S$
  Symbolic descriptions: $s \xrightarrow{x<50,?\mathrm{coin},x:=x+\mathrm{coin}} s'$
- $I \subseteq S \times 2^D$
  Symbolic descriptions: $(s_0, x = 0)$

**Example: Vending machine**

- coffee: 50 cents, orange juice: 1 euro, ...
- possible coins: 10, 20, 50 cents
- we may shuffle coin insertions and drink selection

# Transition systems with variables

## Semantics: low level TS

- $S' = S \times D$
- $I' = \{(s, \nu) \mid \exists (s, g) \in I \text{ with } \nu \models g\}$
- Transitions: $T' \subseteq (S \times D) \times \Sigma \times (S \times D)$

$$\frac{s \xrightarrow{g,a,f} s' \wedge \nu \models g}{(s, \nu) \xrightarrow{a} (s', f(\nu))}$$

SOS: Structural Operational Semantics

- $\mathrm{AP}'$: we may use atomic propositions in $\mathrm{AP}$ or guards in $2^D$ such as $x > 0$.

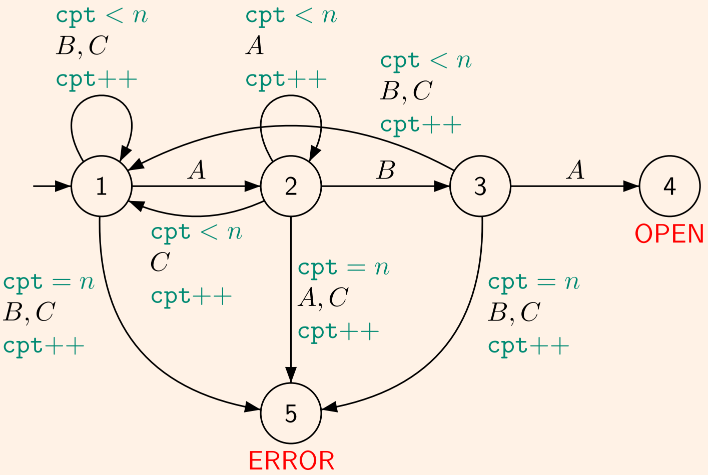
## Programs = Kripke structures with variables

- Program counter = states
- Instructions = transitions
- Variables = variables

## Example: GCD

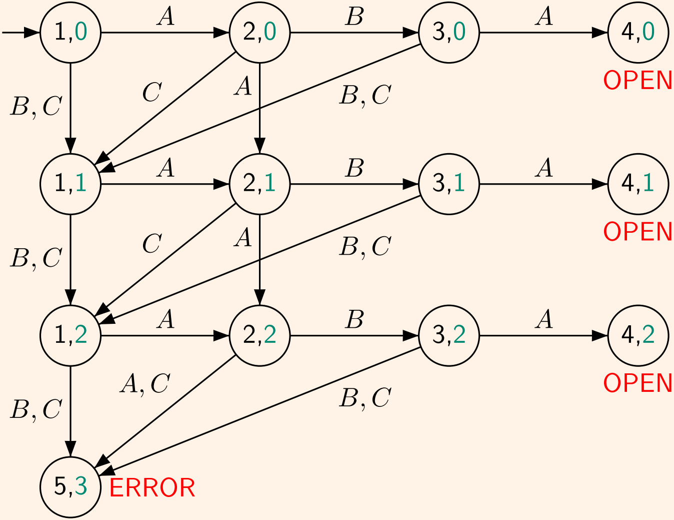
# TS with variables . . .

Example: Digicode



# . . . and its semantics ($n = 2$)

Example: Digicode



# Modular description of concurrent systems

$$M = M_1 \parallel M_2 \parallel \cdots \parallel M_n$$

**Semantics**

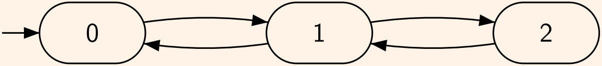
- Various semantics for the parallel composition $\parallel$
- Various communication mechanisms between components:
  Shared variables, FIFO channels, Rendez-vous, ...
- Various synchronization mechanisms

**Example: Elevator with 1 cabin, 3 doors, 3 calling devices**

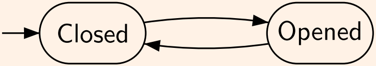
# Modular description of concurrent systems

## Example: Elevator

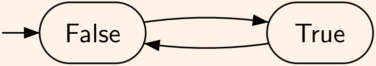
- Cabin:




- Door for level $i$:




- Call for level $i$:




The actual system is a synchronized product of all these automata.
It consists of (at most) $3 \times 2^3 \times 2^3 = 192$ states.

# Synchronized products

## Definition: General product

- Components: $M_i = (S_i, \Sigma_i, T_i, I_i, \mathrm{AP}_i, \ell_i)$
- Product: $M = (S, \Sigma, T, I, \mathrm{AP}, \ell)$ with

  $S = \prod_i S_i, \quad \Sigma = \prod_i (\Sigma_i \cup \{\varepsilon\}), \quad$ and $\quad I = \prod_i I_i$

  $T = \{(p_1, \ldots, p_n) \xrightarrow{(a_1,\ldots,a_n)} (q_1, \ldots, q_n) \mid$ for all $i$, $(p_i, a_i, q_i) \in T_i$ or $p_i = q_i$ and $a_i = \varepsilon\}$

  $\mathrm{AP} = \biguplus_i \mathrm{AP}_i$ and $\ell(p_1, \ldots, p_n) = \bigcup_i \ell(p_i)$

## Synchronized products: restrictions of the general product.

Parallel compositions

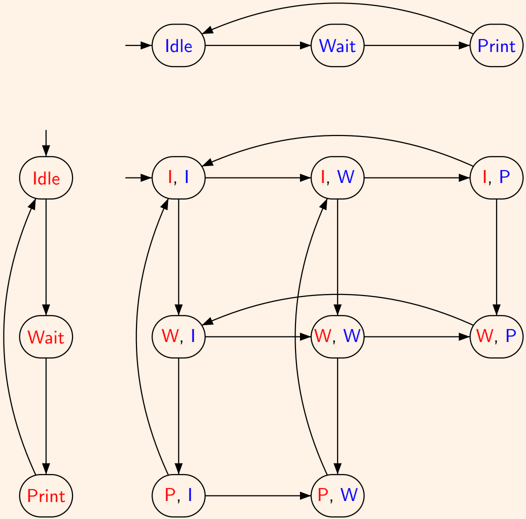
- Synchronous: $\Sigma_{\mathrm{sync}} = \prod_i \Sigma_i$
- Asynchronous: $\Sigma_{\mathrm{sync}} = \biguplus_i \Sigma'_i \qquad$ with $\Sigma'_i = \{\varepsilon\}^{i-1} \times \Sigma_i \times \{\varepsilon\}^{n-i}$

Synchronizations

- By states: $S_{\mathrm{sync}} \subseteq S$
- By labels: $\Sigma_{\mathrm{sync}} \subseteq \Sigma$
- By transitions: $T_{\mathrm{sync}} \subseteq T$

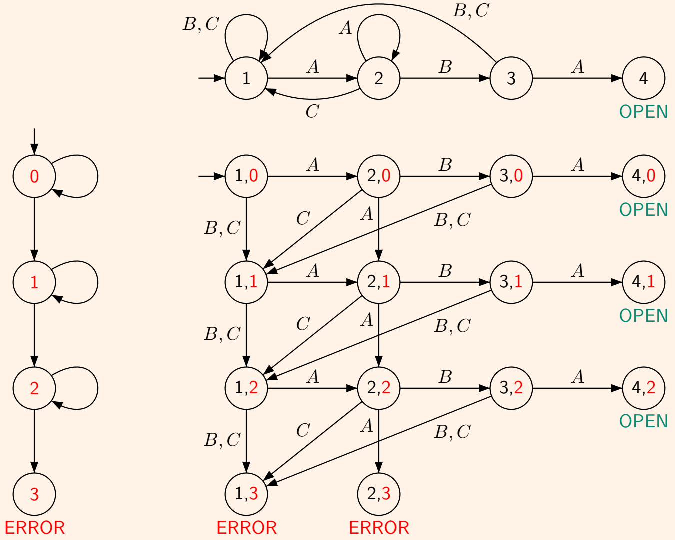
# Example: Printer manager

Example: Asynchronous product
Synchronization by states: $(P, P)$ is forbidden



# Example: digicode

Example: Synchronous product
Synchronization by transitions



# Synchronization by Rendez-vous

Synchronization by transitions is universal but too low-level.

## Definition: Rendez-vous

- $!m$ sending message $m$
- $?m$ receiving message $m$
- SOS: Structural Operational Semantics

Local actions
$$\frac{s_1 \xrightarrow{a_1}_1 s_1'}{(s_1, s_2) \xrightarrow{a_1} (s_1', s_2)} \qquad \frac{s_2 \xrightarrow{a_2}_1 s_2'}{(s_1, s_2) \xrightarrow{a_2} (s_1, s_2')}$$

Rendez-vous
$$\frac{s_1 \xrightarrow{!m}_1 s_1' \wedge s_2 \xrightarrow{?m}_2 s_2'}{(s_1, s_2) \xrightarrow{m} (s_1', s_2')} \qquad \frac{s_1 \xrightarrow{?m}_1 s_1' \wedge s_2 \xrightarrow{!m}_2 s_2'}{(s_1, s_2) \xrightarrow{m} (s_1', s_2')}$$

- It is a kind of synchronization by actions.
- Essential feature of process algebra.

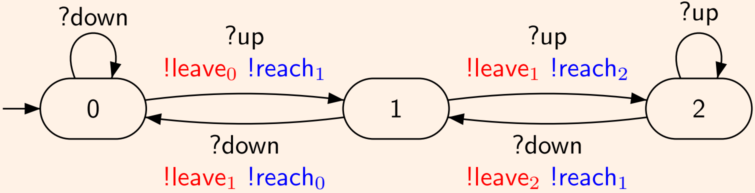
## Example: Elevator with 1 cabin, 3 doors, 3 calling devices

- $?\mathrm{up}$ is uncontrollable for the cabin
- $?\mathrm{leave}_i$ is uncontrollable for door $i$
- $?\mathrm{call}_0$ is uncontrollable for the system

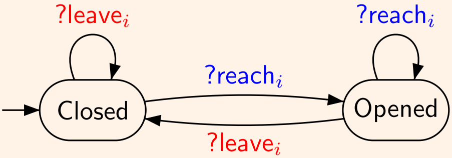
# Example: Elevator

## Example: Synchronization by Rendez-vous

Cabin:




Door for level $i$:




We should design the controller

# Shared variables

## Definition: Asynchronous product + shared variables

$\bar{s} = (s_1, \ldots, s_n)$ denotes a tuple of states
$\nu \in D = \prod_{v \in \mathcal{V}} D_v$ is a valuation of variables.

Semantics (SOS)

$$\frac{\nu \models g \wedge s_i \xrightarrow{g,a,f} s'_i \wedge s'_j = s_j \text{ for } j \neq i}{(\bar{s}, \nu) \xrightarrow{a} (\bar{s}', f(\nu))}$$

## Example: Mutual exclusion for 2 processes satisfying

- Safety: never simultaneously in critical section (CS).
- Liveness: if a process wants to enter its CS, it eventually does.
- Fairness: if process 1 wants to enter its CS, then process 2 will enter its CS at most once before process 1 does.

using shared variables but no synchronization mechanisms: the atomicity is

- testing or reading or writing a single variable at a time
- no test-and-set: $\{x = 0; x := 1\}$