Model-based Engineering of Supervisory Controllers

Michel Reniers

M.A.Reniers@tue.nl

IPA Formal Methods Course

2018.06.14

TU/e Technische Universiteit Eindhoven University of Technology

Subjects

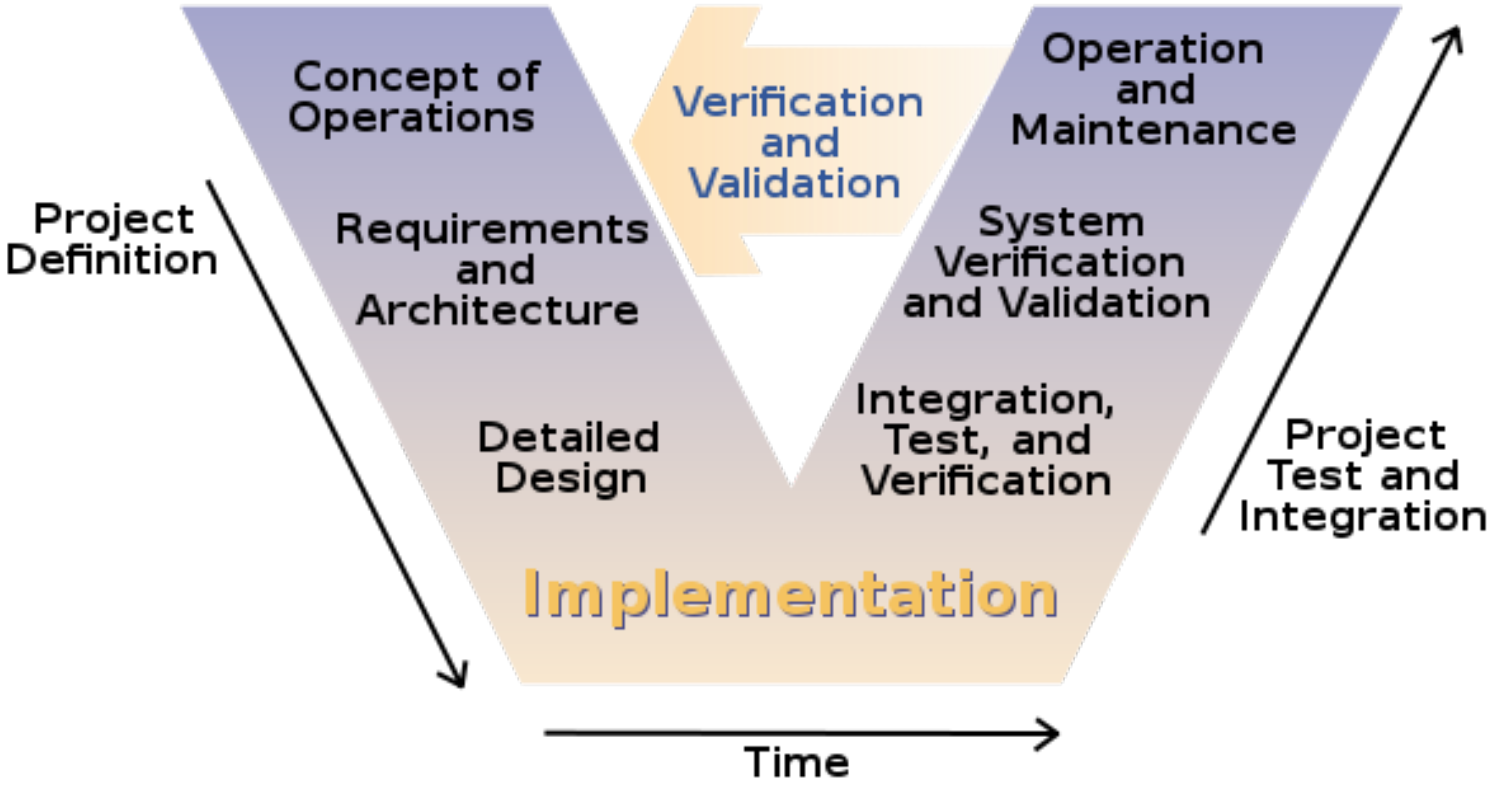
- Model-based engineering (MBE) of supervisory controllers
- Design models
- Modelling and validation (simulation and visualization/animation)
- Discrete-event models
- Supervisor synthesis
  - MBSE for supervisory control synthesis
  - Specification of requirements
  - Basic data-based supervisory control problem
  - CIF support for synthesis

Systems Engineering

- Interdisciplinary process
- To ensure stakeholder needs are satisfied
- In a high quality, trustworthy, cost efficient and schedule compliant manner
- Throughout entire system life cycle

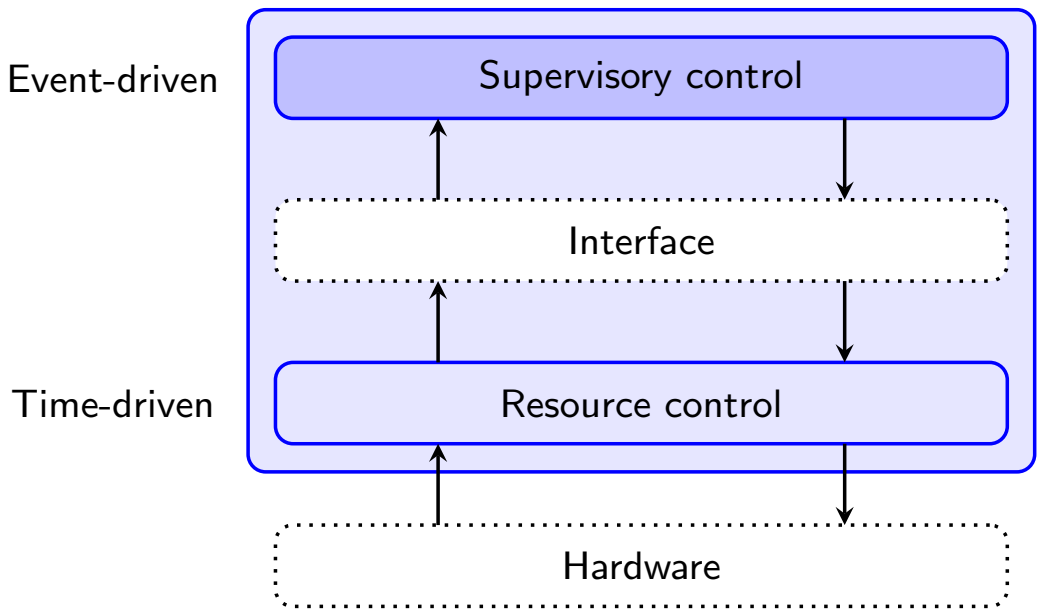
1. State the problem. Requirements should be traceable to this problem statement. What must be done, not how to do it.
2. Investigate alternatives. Performance, schedule, cost and risk figures, compliance with requirements.
3. Model the system. Models help explain the system, can be used in tradeoff studies and risk management.
4. Integrate. Components, modules, systems, etc. must be integrated so that they interact with one another.
5. Launch the system. Preferred alternative is designed in detail; the parts are built, integrated and tested at various levels.
6. Assess performance. Technical performance measures and metrics are used.
7. Re-evaluate, re-evaluate, re-evaluate.

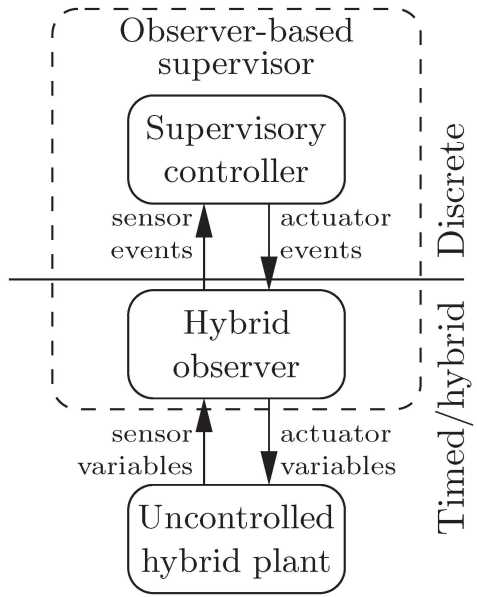
Traditional engineering model: V-model



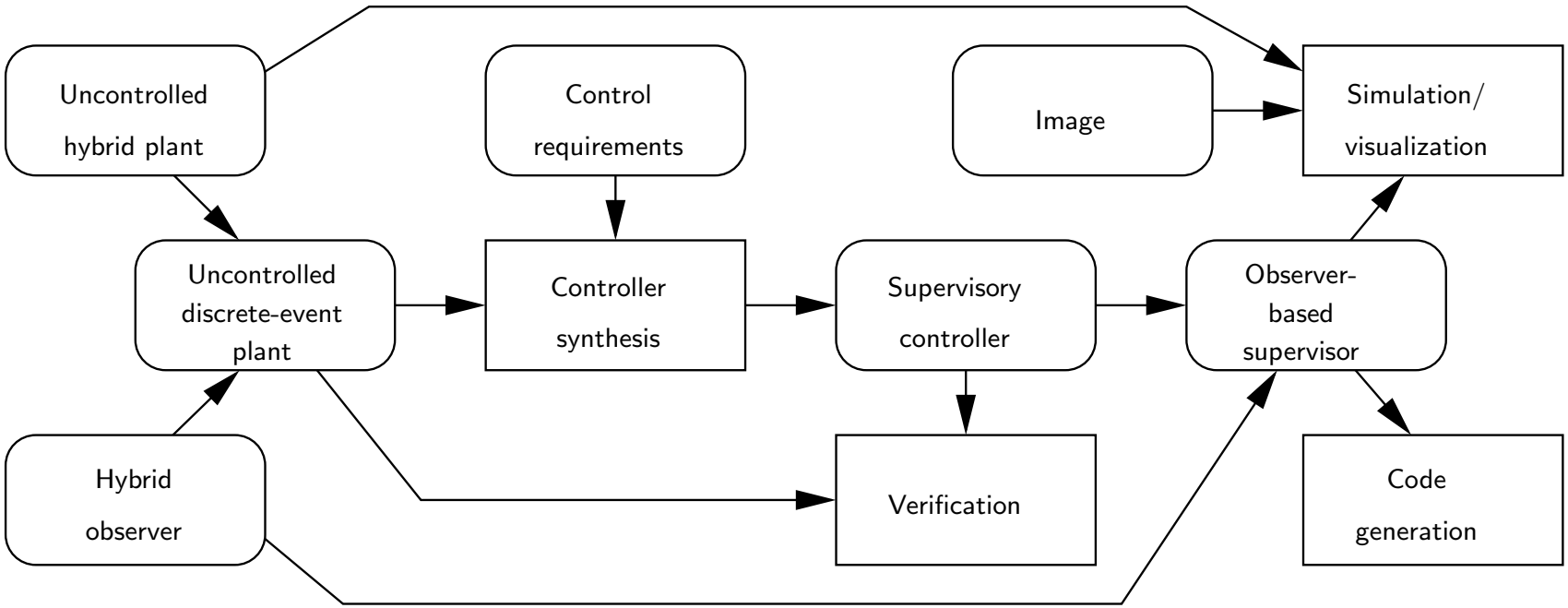
Supervisory control

Coordination of behavior of a (cyber-physical) system from discrete-event observations of its state.
Based on them the supervisory controller decides on the activities that the uncontrolled system can safely perform and that lead to acceptable system behavior.

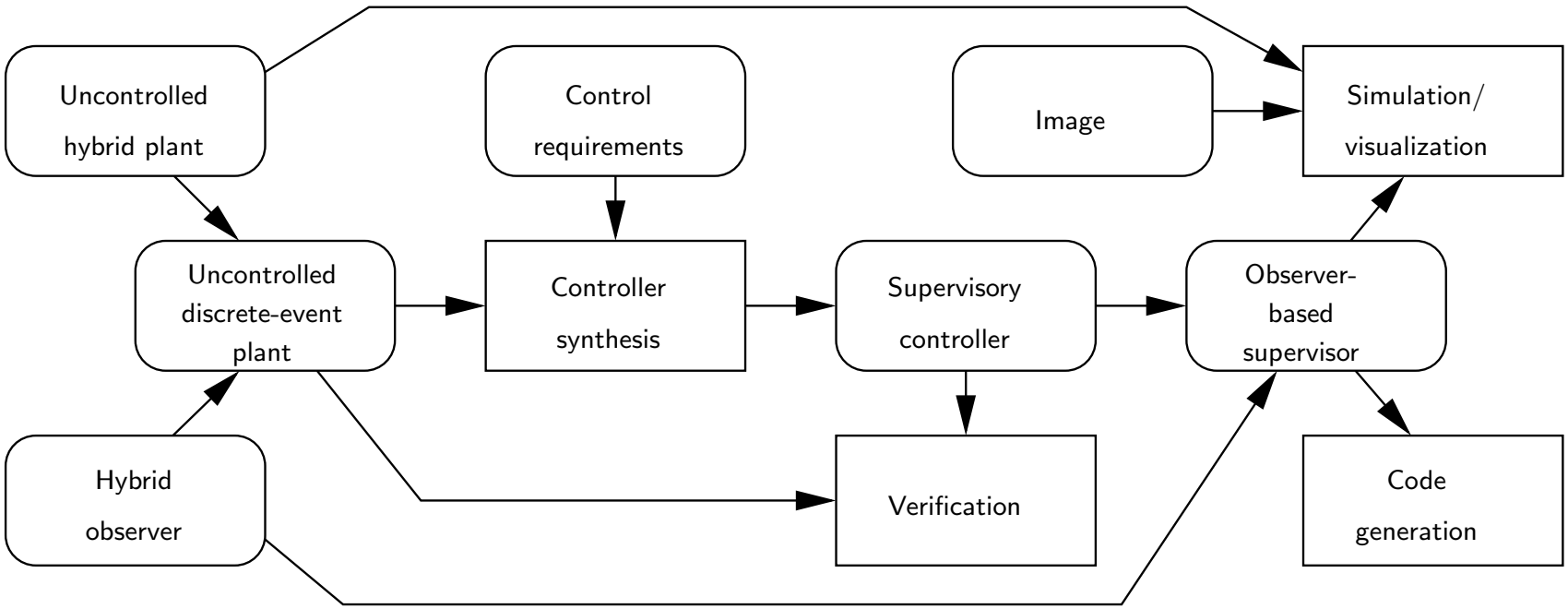




MBSE for supervisory control



Modelling plants and supervisors



- uncontrolled system / plant
- supervisor / controller
- controlled plant
- requirements

Concepts in cyber-physical systems

1. *Physical quantities*, such as temperature, speed, mass, energy, etc.

2. Different *operational modes*, such as equipment being on/off, maintenance modes, error modes, modes associated with different functions of the systems under consideration.

3. *Decomposition* of the system in subsystems, e.g., a robot may consist of subsystems for movement, vision, etc.

4. *Interaction* between the subsystems, e.g., exchange of material and information.

Physical quantities

Time-evolution of such quantities: differential equations!

Acceleration of a vehicle: $\dot{v} = 15$

Mass attached to spring:

$$m \cdot \ddot{x} + k \cdot x = 0$$

where

- x is the extension of the spring
- m is the mass
- k is the spring constant that represents a measure of spring stiffness

Mode switching

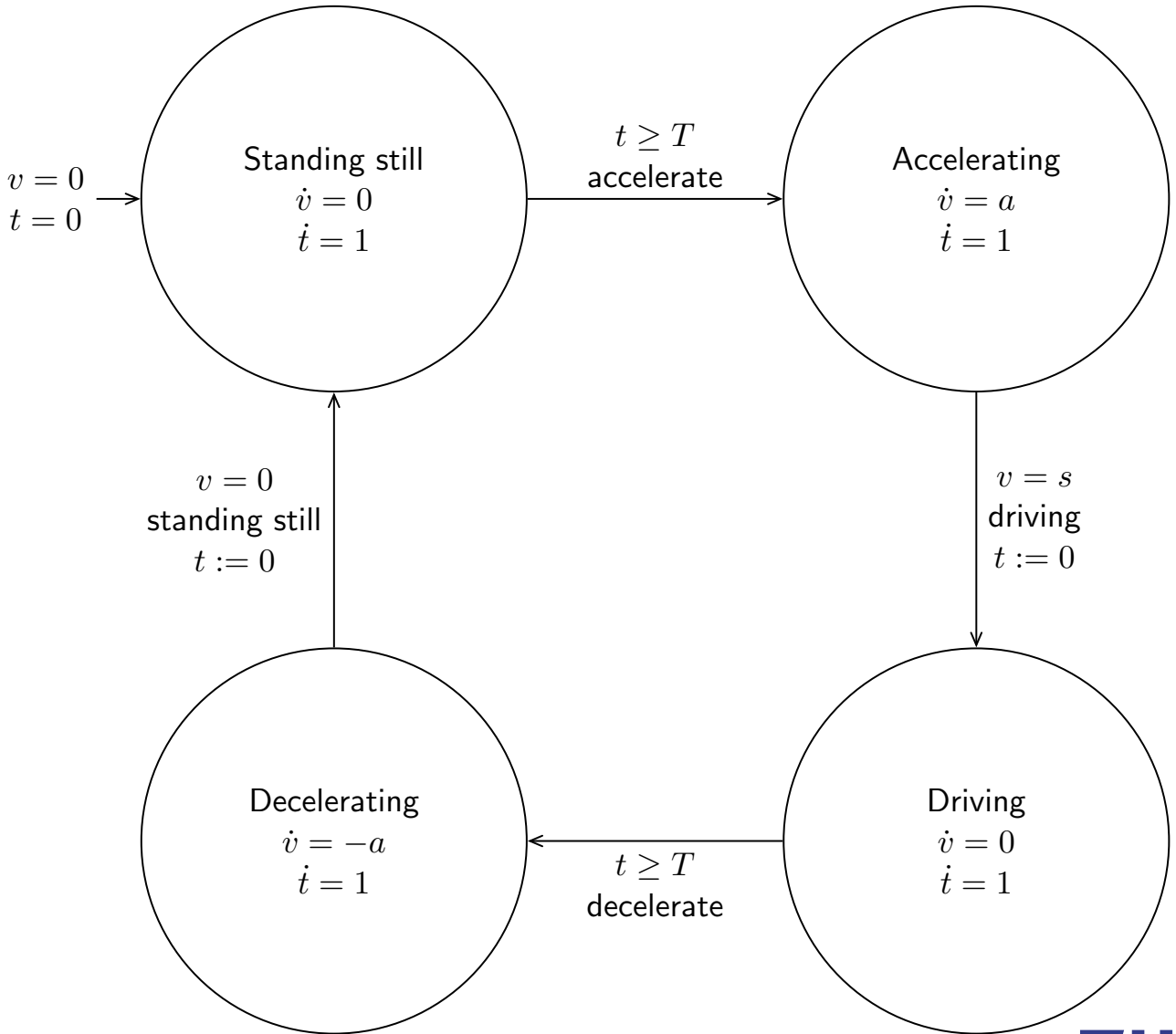
Different operational modes of a vehicle:

- Standing still: $\dot{v} = 0$
- Accelerating: $\dot{v} = a$
- Driving: $\dot{v} = 0$
- Decelerating: $\dot{v} = -a$

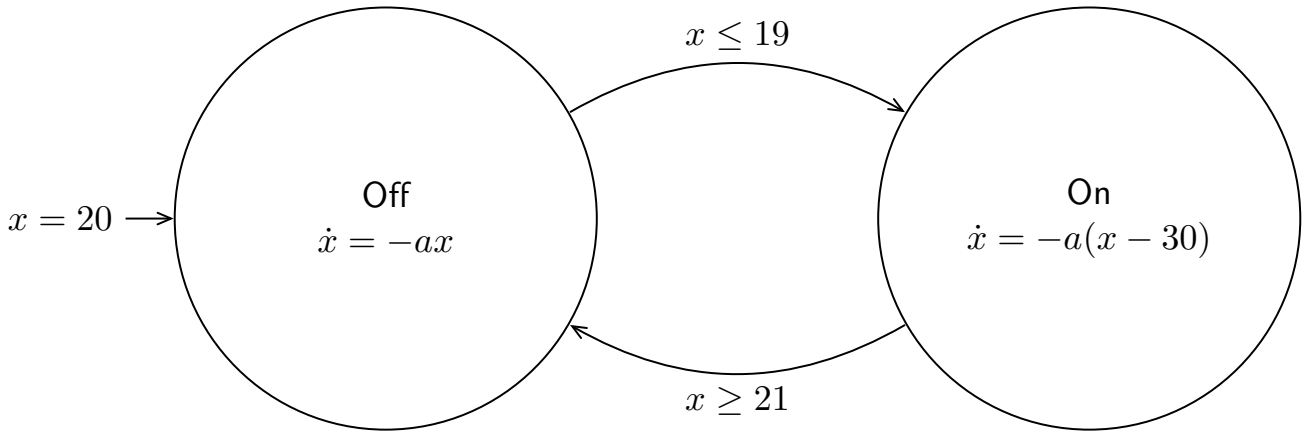
Switching between modes based on values of continuous quantities such as speed.

Exercise: give some examples of systems where you observe phenomena that could be classified as mode switching!

Mode switching in hybrid automata



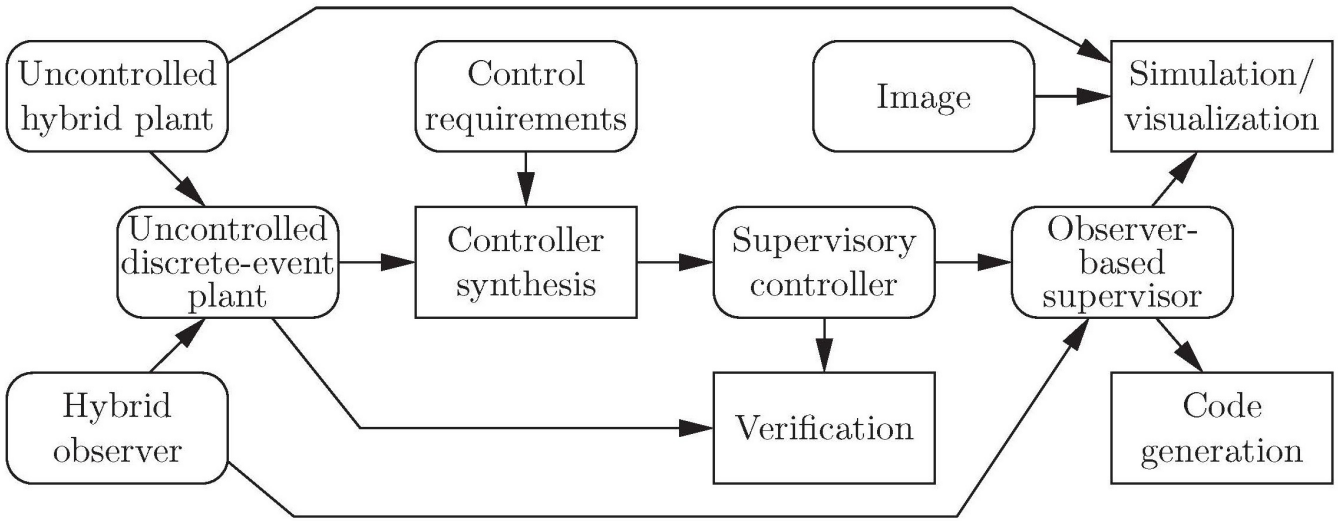
Another example: Thermostat



We will use simulation to convince ourselves (and others) that the model is in correspondence with reality. This called validation.

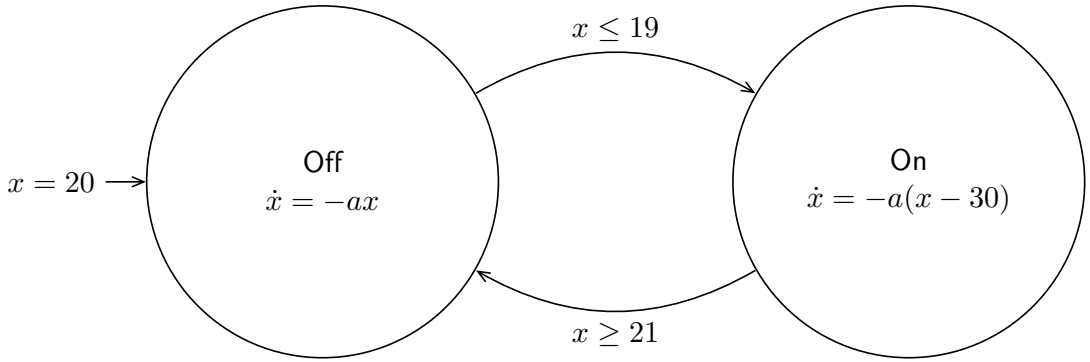
CIF modelling language

- Automata-based modelling language that supports MBE of supervisory controllers.



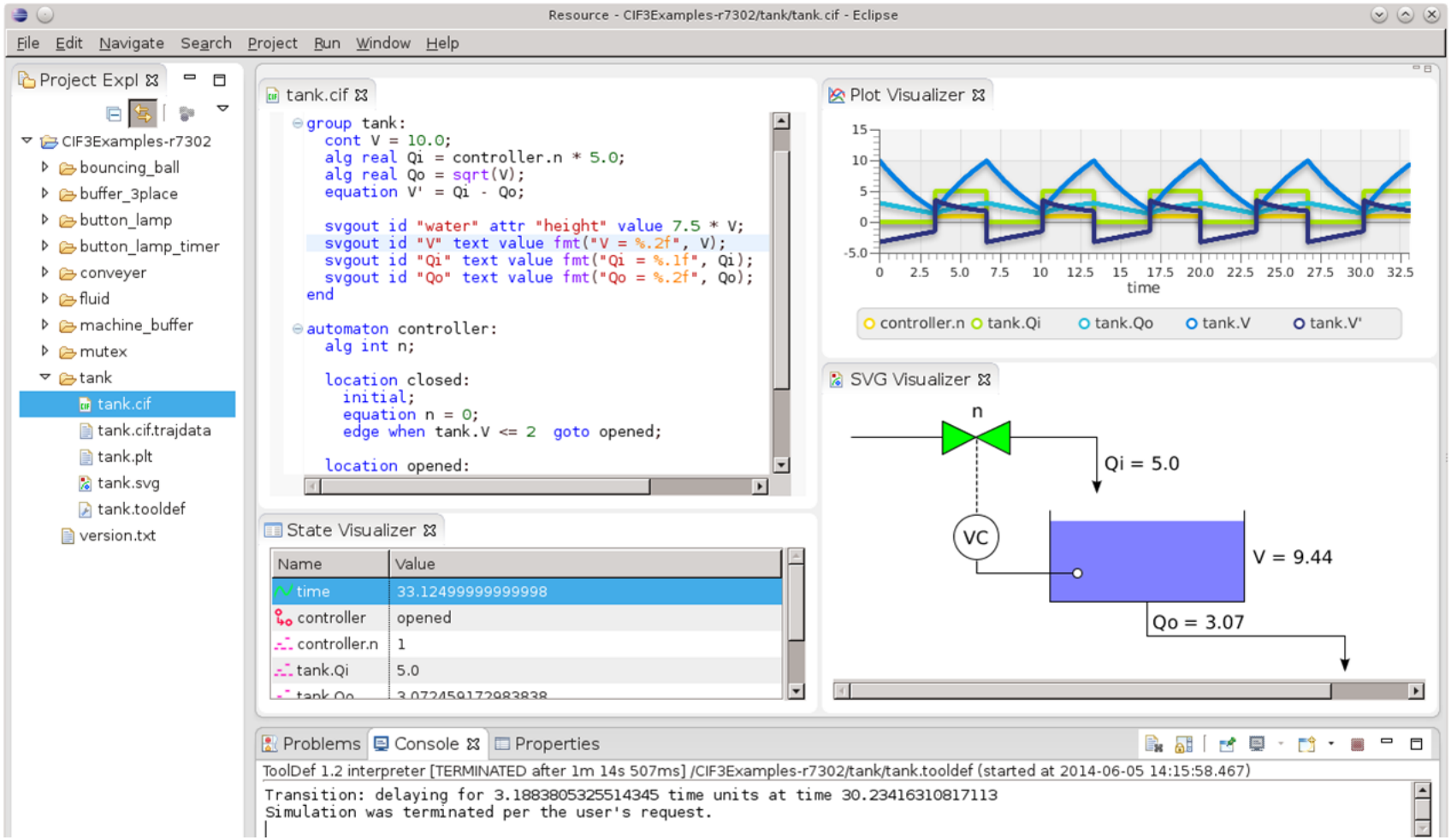
- Developed by Control Systems Technology Group.
- CIF open source (cif.se.wtb.tue.nl): code can be freely downloaded, used and adapted.

CIF model for thermostat

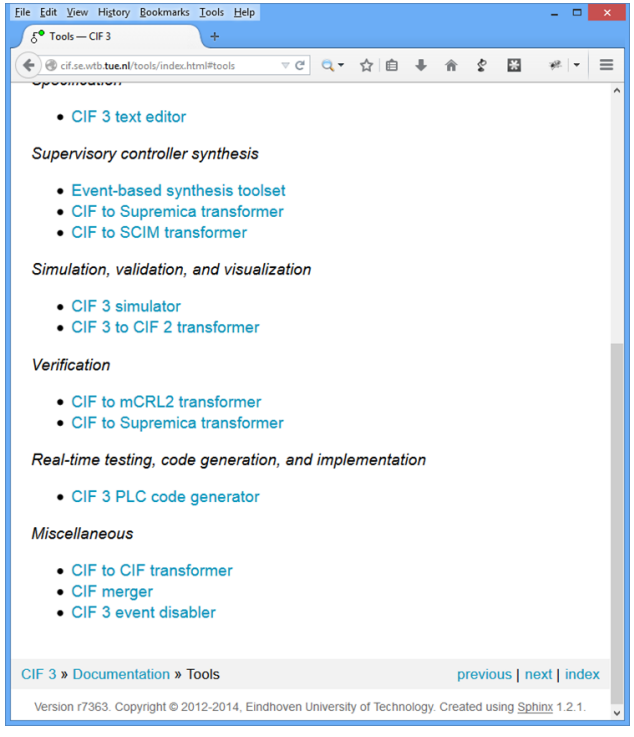



```
1  automaton thermostat:
2    const real a = 0.1;
3    cont x = 20;
4
5    location Off:
6      initial;
7      equation x' = - a * x;
8      edge when x <= 19 goto On;
9
10   location On:
11     equation x' = - a * (x - 30);
12     edge when x >= 21 goto Off;
13 end
```

CIF screenshot



CIF documentation

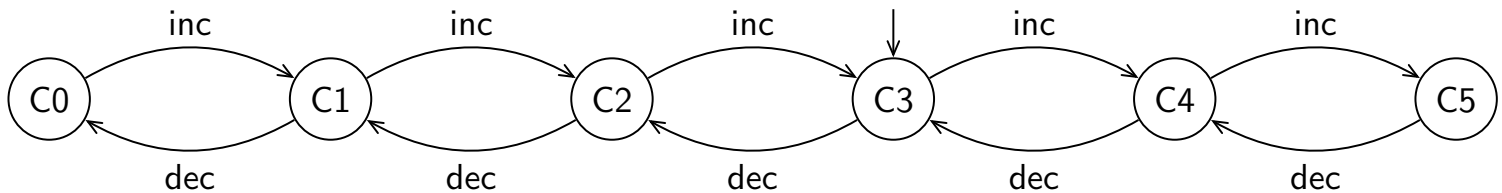


CIF demo

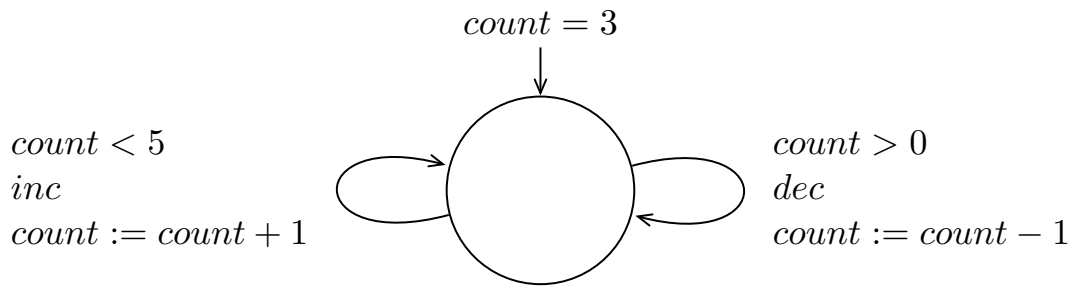
Discrete variables

- How to represent quantities that are discrete?
  - number of vehicles in front of a traffic light
  - number of pages inside a certain part of a printer
  - menu choices in a vending machine or an ATM

- Use additional states!



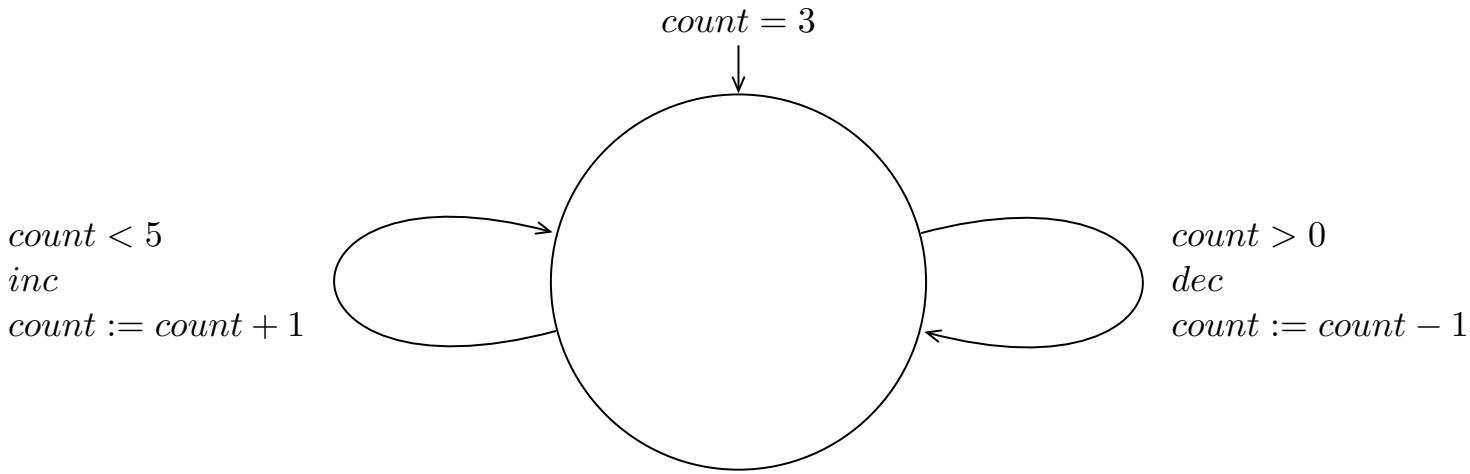
Discrete variables

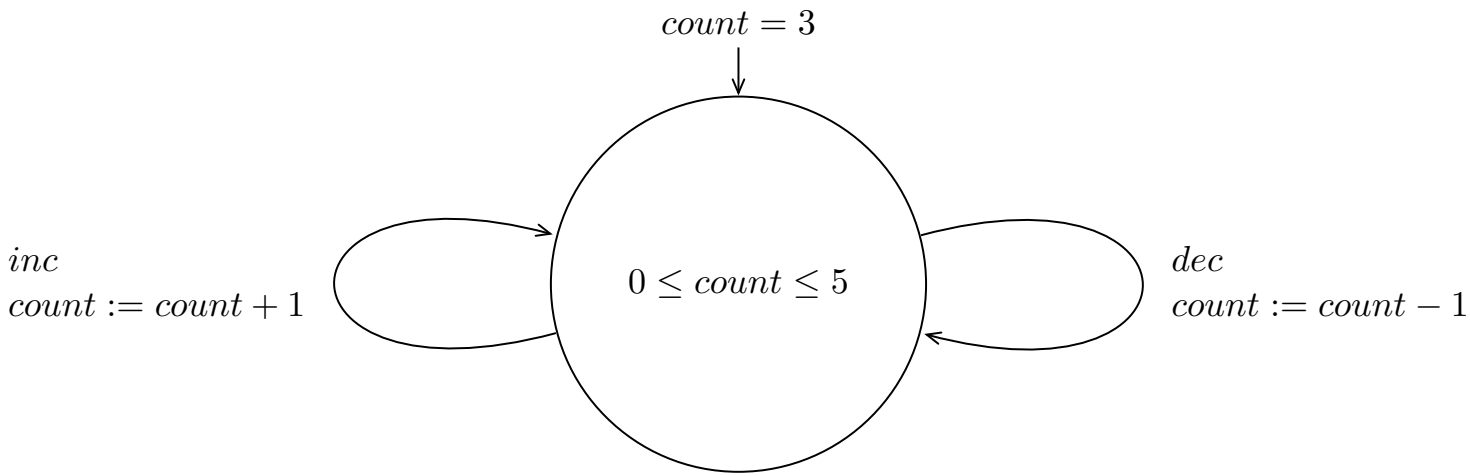


- Remain constant during time passing
- May change value as consequence of a mode switch or transition

```
1 automaton counter:
2   event inc, dec;
3
4   disc int count = 3;
5
6   location:
7     edge dec when count > 0 do count := count - 1;
8     edge inc when count < 5 do count := count + 1;
9  end
```

Invariants


count = 3
count < 5
inc
count := count + 1
count > 0
dec
count := count − 1


count = 3
inc
count := count + 1
0 ≤ count ≤ 5
dec
count := count − 1

Networks of automata

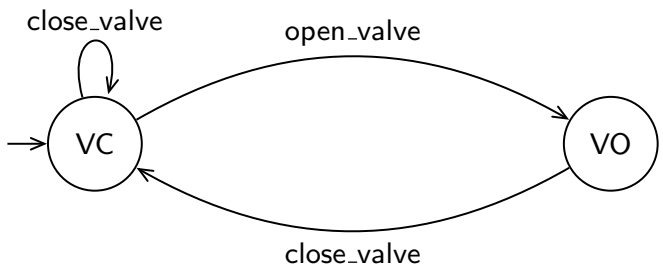
- Systems are composed of several subsystems geographically or logically
- Modelling cyber-physical systems in single automaton not feasible: state space explosion

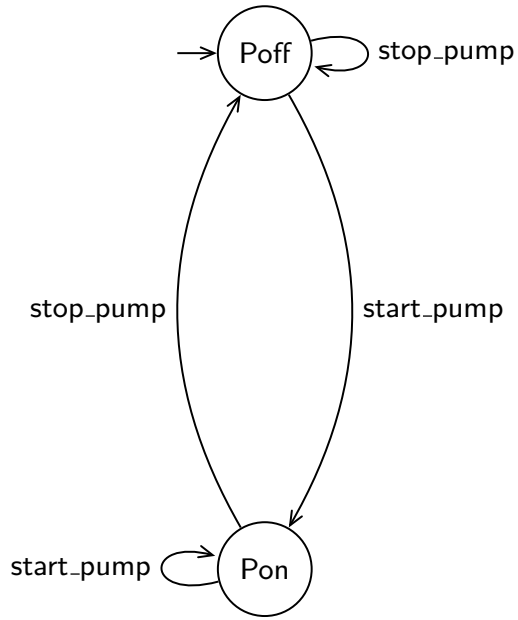
$\Rightarrow$ network of interacting automata

Interaction mechanisms:

- synchronization on time
- synchronization of events
- shared variables

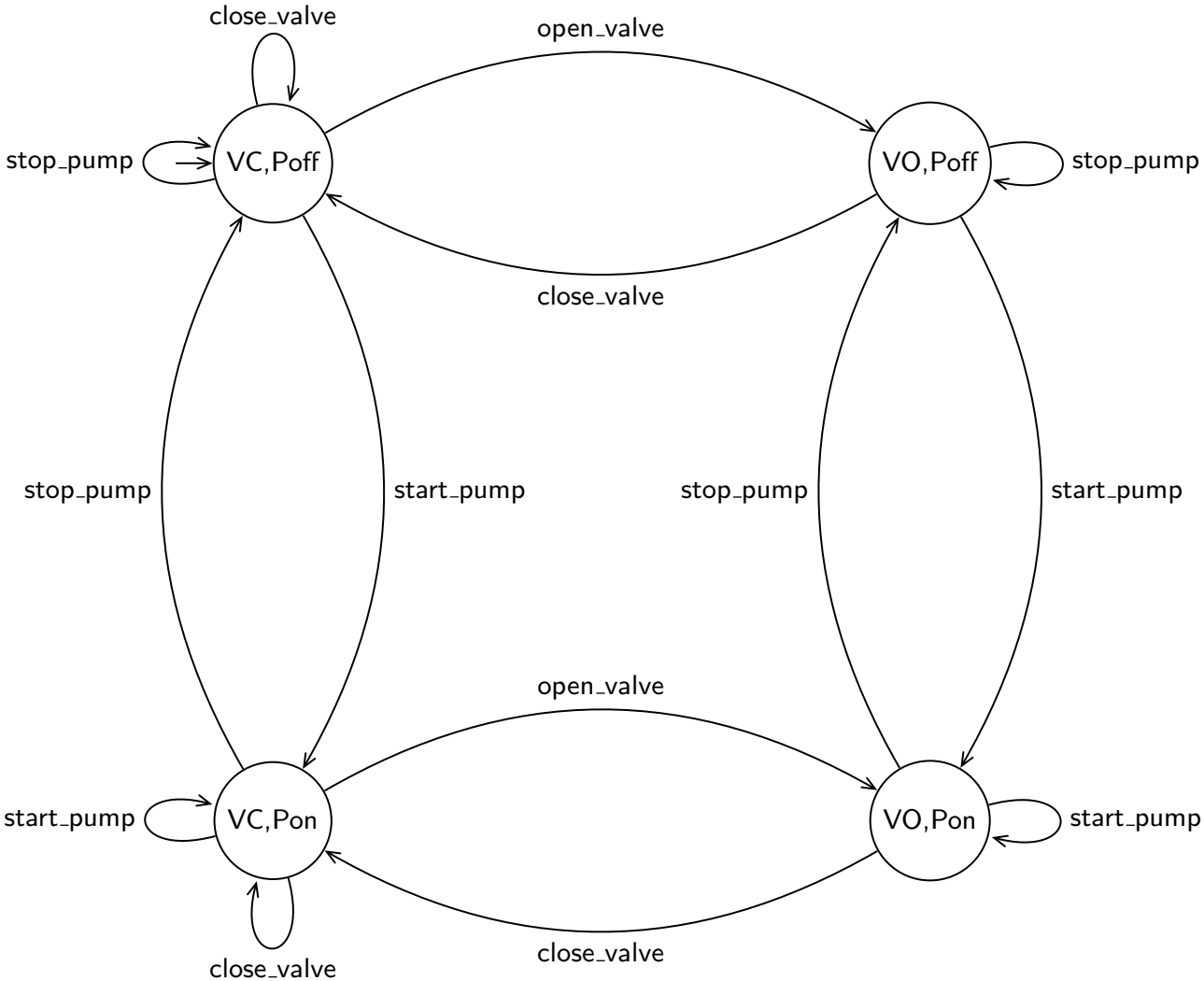
Independent automata





Exercise: Give a single automaton with the same behaviour

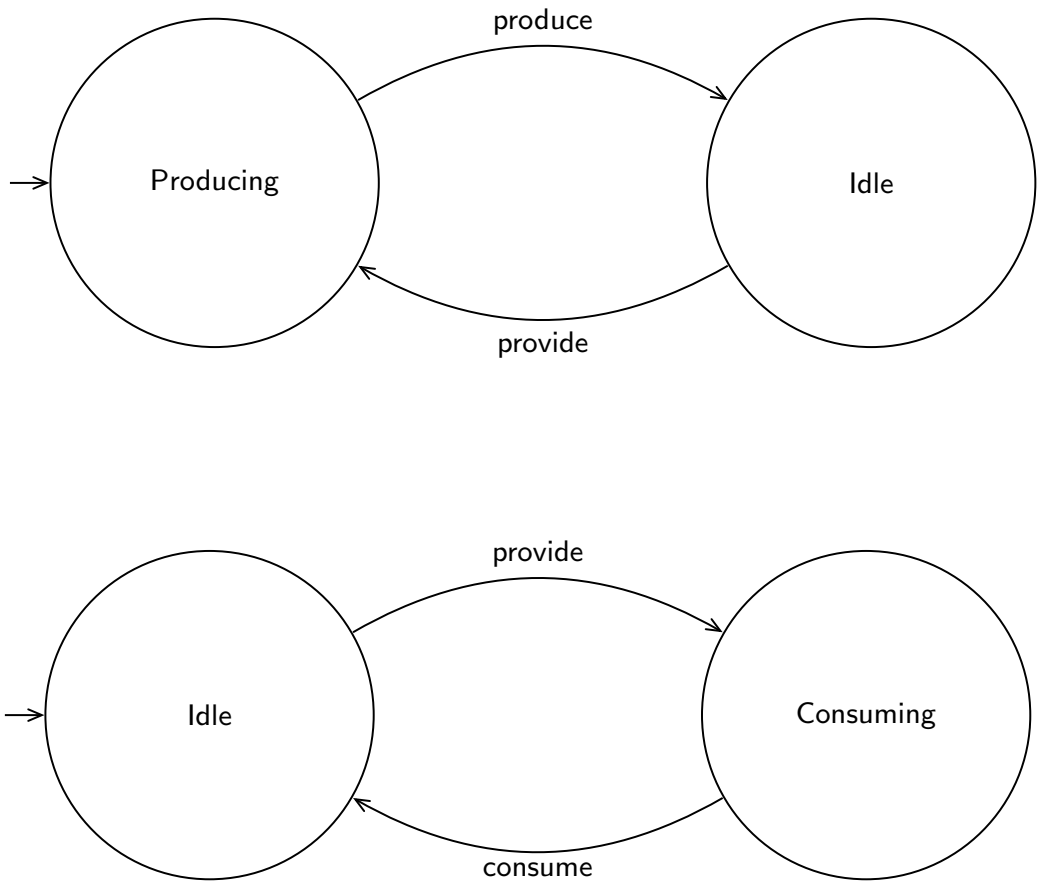
Solution



Synchronization on time


```
1  automaton Var_x:
2    cont x = 0;
3
4    location X:
5      initial;
6      equation x' = 2;
7  end
8
9  automaton Var_y:
10   cont y = 0;
11
12   location Y:
13     initial;
14     equation y' = 1;
15 end
```

Synchronization of events



Exercise: Represent behaviour by single automaton.

Synchronization of events in CIF

```
1                                                  event provide;
2
3  automaton producer:                             automaton producer:
4    event produce, provide;                         event produce;
5    location producing:                             location producing:
6      initial;                                      initial;
7      edge produce goto idle;                         edge produce goto idle;
8    location idle:                                  location idle:
9      edge provide goto producing;                    edge provide goto producing;
10 end                                               end
11
12 automaton consumer:                             automaton consumer:
13   event consume;                                  event consume;
14   location idle:                                  location idle:
15     initial;                                        initial;
16     edge producer.provide goto consuming;           edge provide goto consuming;
17   location consuming:                             location consuming:
18     edge consume goto idle;                         edge consume goto idle;
19 end                                               end
```

Shared variables

Discrete variables:

- Declared in an automaton
- May only be assigned in that automaton
- May be read by other automata (used in guards and invariants)

Continuous variables:

- Declared in automaton or globally
- May only be assigned in the automaton they are declared in (if any)
- May be read by other automata

CIF example

```
automaton queue1:
  event enter, leave;
  disc int count = 0;
  location: initial;
    edge enter when count < 2 do count := count + 1;
    edge leave when count > 0 do count := count - 1;
end

automaton queue2:
  event enter, leave;
  disc int count = 0;
  location: initial;
    edge enter when count < 2 do count := count + 1;
    edge leave when count > 0 do count := count - 1;
end

automaton customer:
  location: initial;
    edge queue1.enter when queue1.count <= queue2.count;
    edge queue2.enter when queue2.count <= queue1.count;
end
```

Recap

- Model-based systems engineering as an approach to the engineering of supervisory controllers
- Modelling formalisms hybrid automata and CIF 3.0 as supporting tool
- Various types of models
  - Continuous state or discrete state
  - Continuous time or discrete time
  - Time driven or event driven
- Various examples
  - Hybrid uncontrolled plants
  - Discrete-event supervisors

Model-based Engineering of Supervisory Controllers

Michel Reniers

M.A.Reniers@tue.nl

IPA Formal Methods Course

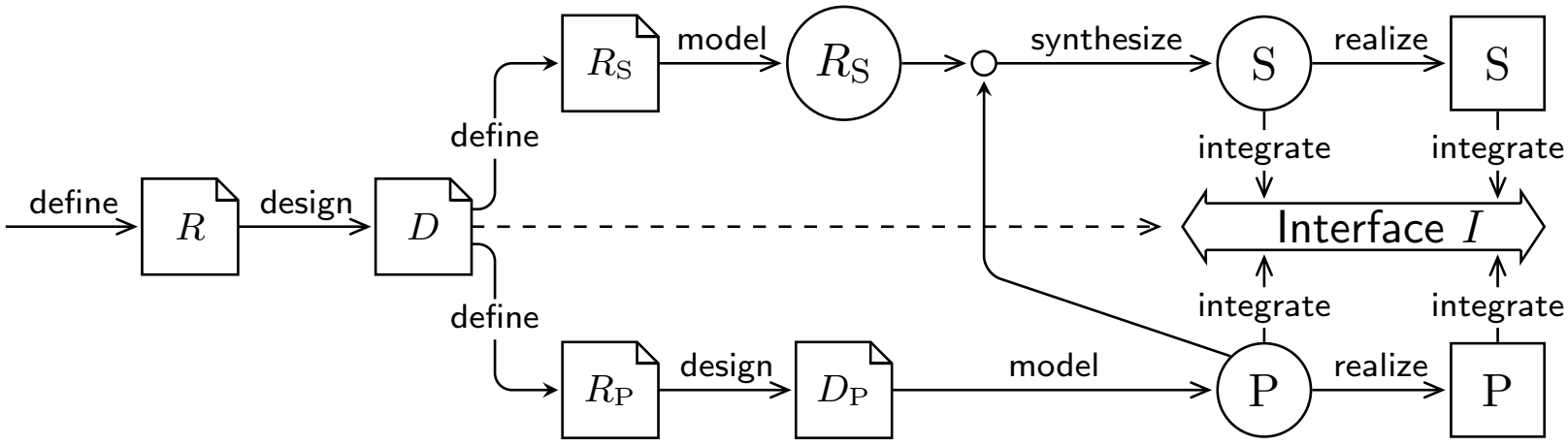
2018.06.14

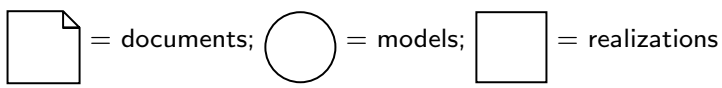
TU/e Technische Universiteit Eindhoven University of Technology

Overview

- MBSE for supervisory control synthesis
- Specification of requirements
- Basic data-based supervisory control problem
- CIF support for synthesis

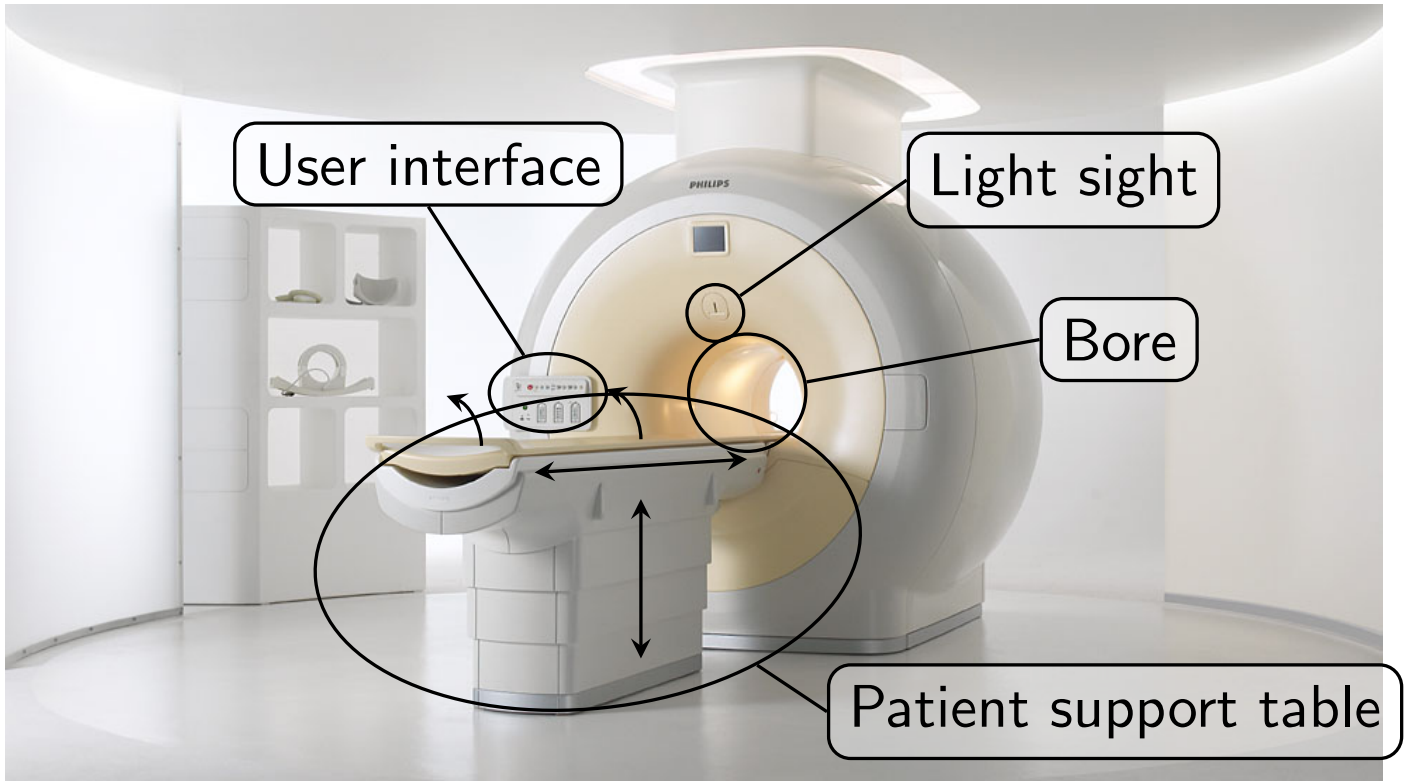
MBSE for supervisory control synthesis





Motivation

Example: safe tabletop handling in an MRI scanner



Motivation

Control requirements:

- Ensure that the tabletop does not move beyond its vertical and horizontal end positions.
- Prevent collisions of the tabletop with the magnet.
- Define the conditions for manual and automatic movements of the tabletop.
- Enable the operator to control the system by means of the manual button and the tumble switch.

Motivation

Two ways of working:

1. Traditional
   - Design (manually) a supervisory controller
   - Verify that the requirements are satisfied (by testing or by model checking)
2. Synthesis-based
   - Define formal plant and requirement models
   - Synthesize (automatically) the supervisory controller
   - (Almost) No need for verification: the specified requirements are satisfied per construction

Synthesis of supervisory controllers

- Model uncontrolled system components (plant)
- Model requirements
- Synthesize supervisory controller such that controlled system
  - satisfies requirements
  - is non-blocking
  - is controllable
  - is maximally permissive
- Simulate/verify/test controlled plant
- Real-time implementations can be generated

Properties of controlled system

1. controlled system is non-blocking

   It is possible to reach a marked state from all reachable states in the controlled system.

2. controlled system is controllable

   For all reachable states in the controlled system, the uncontrollable events that are enabled in the same state in the uncontrolled system are still possible in the controlled system. In other words, uncontrollable events are not restricted.

3. controlled system is maximally permissive

   The controlled system permits all safe, controllable, and non-blocking behaviors.

Model plant

- Automata restricted to discrete elements:
- States: one initial, several marker states.
  - Marker state models task completion.
- Transitions only caused by events: controllable or uncontrollable.
  - Controllable events can be disabled — actuator commands.
  - Uncontrollable events cannot be disabled — changes to sensor readings.

Advantages of MBSE

- Model-based systems engineering contributes to faster product development.
- Supervisor synthesis eliminates manual design of control software and reduces testing effort.
- Event-based distributed framework (e.g. modular) supports reconfigurability.
- Synthesis-based systems engineering is applicable in industry for developing supervisory controllers.
- Formal models and methods are essential for high-tech systems design.

Guidelines for modeling components

- Split the system to be controlled into separate components.
- Define variables that describe the component behavior at the necessary abstraction level.
- Define the component state space.
- Define control and sensor signals that set and read component variables.
- Define corresponding events.
- Typically, setting control signals are controllable events, whereas reading sensor signals are uncontrollable events.

Guidelines for modeling requirements

- Make the description of the admissible behavior as precise as possible.
- Define a few small requirements as opposed to defining a single large requirement.
- Check that requirement models correspond with their descriptions.
- If the supervisor derived is too large, the requirements are probably too relaxed.
- If the supervisor derived is too small or empty, the requirements are probably too strict or conflicting.
- Iterative development: analyze the supervisor derived for redesign of component or requirement models.

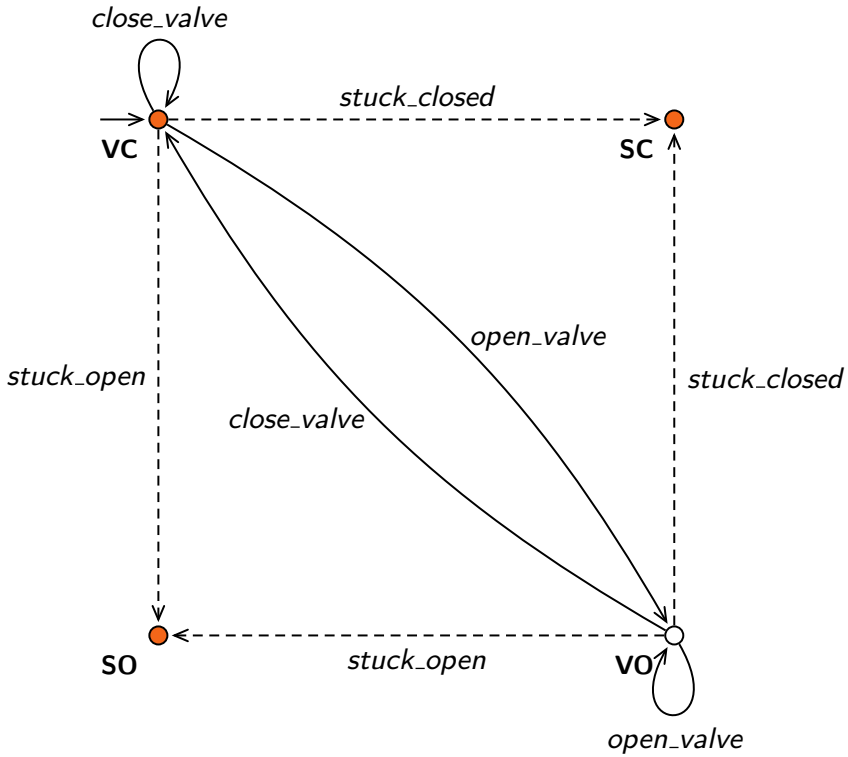
Types of system properties

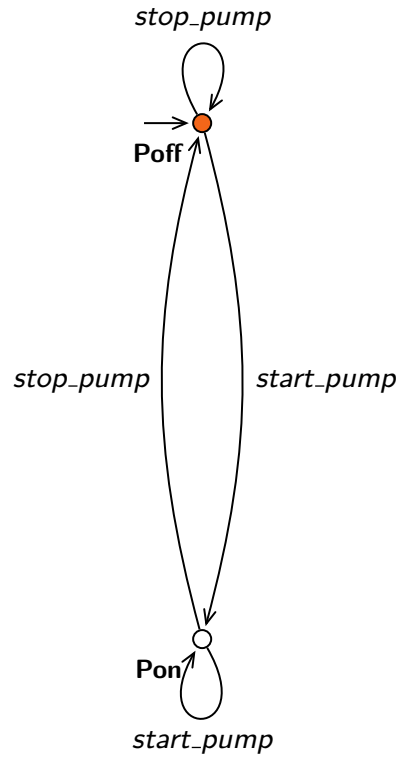
- Safety properties: "nothing bad happens in the system".
- Liveness properties: "something good eventually happens in the system".

In models:

- Safety: no error/blocking state is reachable.
- Liveness = progress: an action is eventually executed.

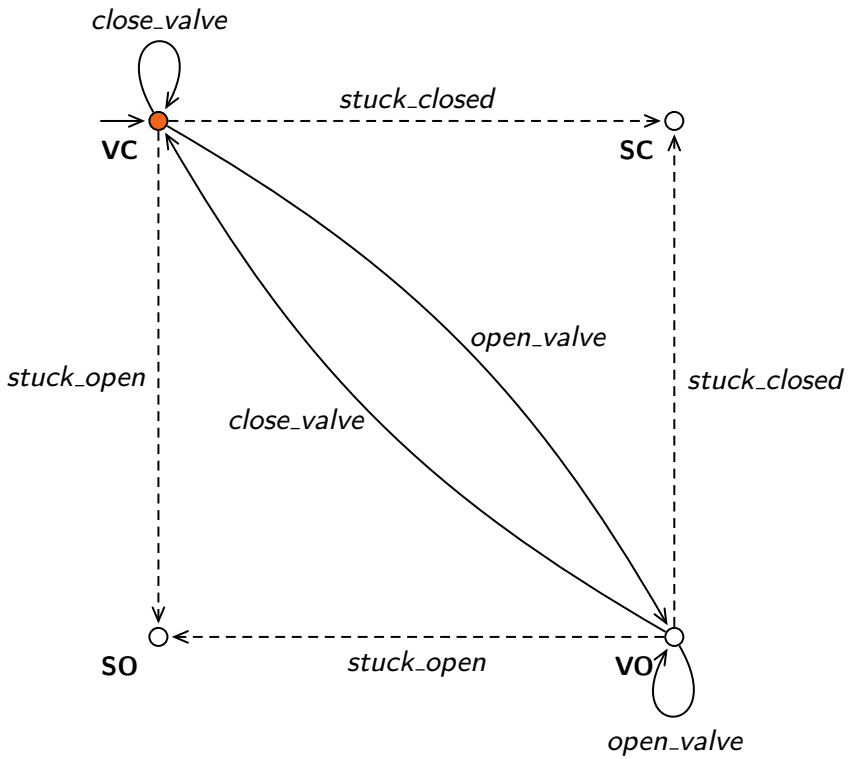
Pump-valve example

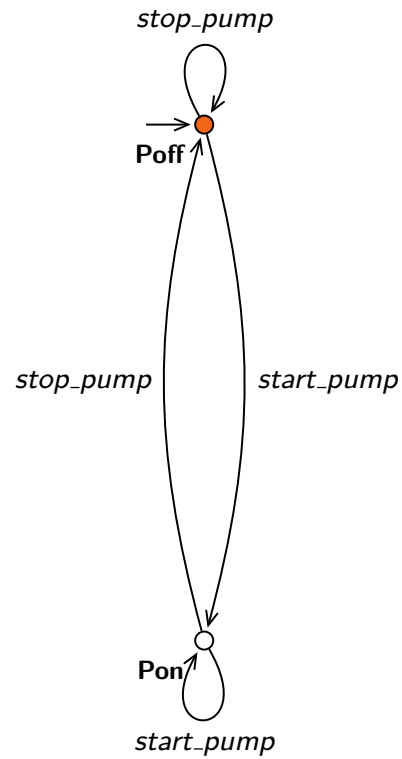




Requirement: prevent that the pump is operating while the valve is closed.

Pump-valve example: progress





Pump-valve example: progress

- For this system, independent of the requirement, no proper supervisor can be synthesized or manually designed.
- The reason is the design choice for marker states: only the initial state is marked.
- As additionally, $stuck\_open$ and $stuck\_close$ are uncontrollable events, it is not possible to prevent the system from getting to **SC** or **SO**.
- We need to reconsider marker states choice.

Syntax of state-based expressions

- System requirements are often expressed in terms of conditions over states.
- State-based specifications often follow naturally from informal, intuitive requirements.
- However, not every requirement can be specified as such a state-based expression.
- Requirements can also be formulated as automata.

Syntax of state-based expressions

- *true* and *false* stand for truth values of propositional logic.
- Negation $\neg$, conjunction $\wedge$, disjunction $\vee$ and implication $\Rightarrow$ stand for standard logical operators.
- State predicate **C.S**$\downarrow$ expresses that component **C** is in state **S**.
- Event predicate $\rightarrow E$ expresses that an event from set E is enabled by the supervisor.
- Most often used state-based expressions are:
  - $\rightarrow E \Rightarrow MS$
  - MS

  where $MS ::= true \mid false \mid$ **C.S**$\downarrow$ $\mid \neg MS \mid MS\ op\ MS$ and $op \in \{\wedge, \vee, \Rightarrow\}$.

State-based expressions in CIF

- $\rightarrow E \Rightarrow MS$ is captured in CIF as

```
1  requirement name:
2     location:
3       initial;
4       marked;
5       edge e1, e2, ..., en when MS;
6  end
```

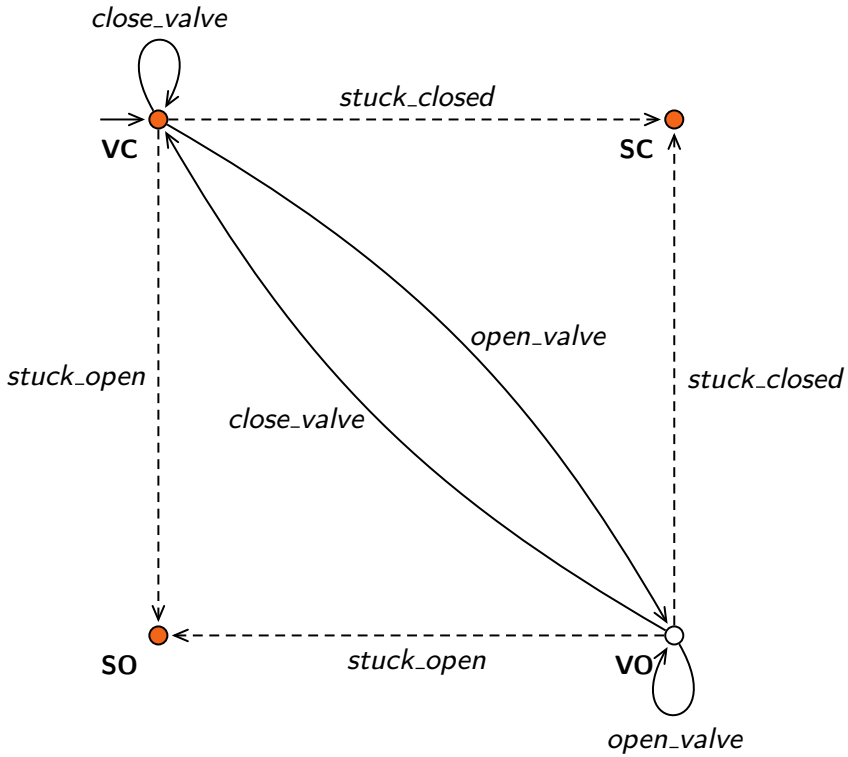
where e1, e2, ..., en are the events that make up set E, and MS is the CIF encoding of expression MS

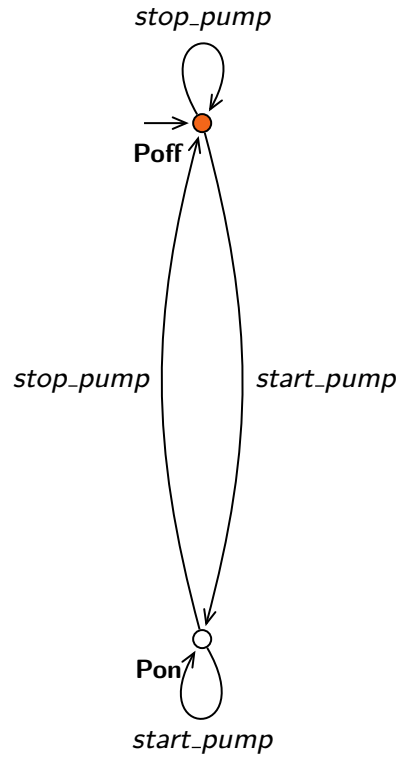
- MS is captured in CIF as

```
1  requirement invariant MS;
```

where MS is the CIF encoding of expression MS

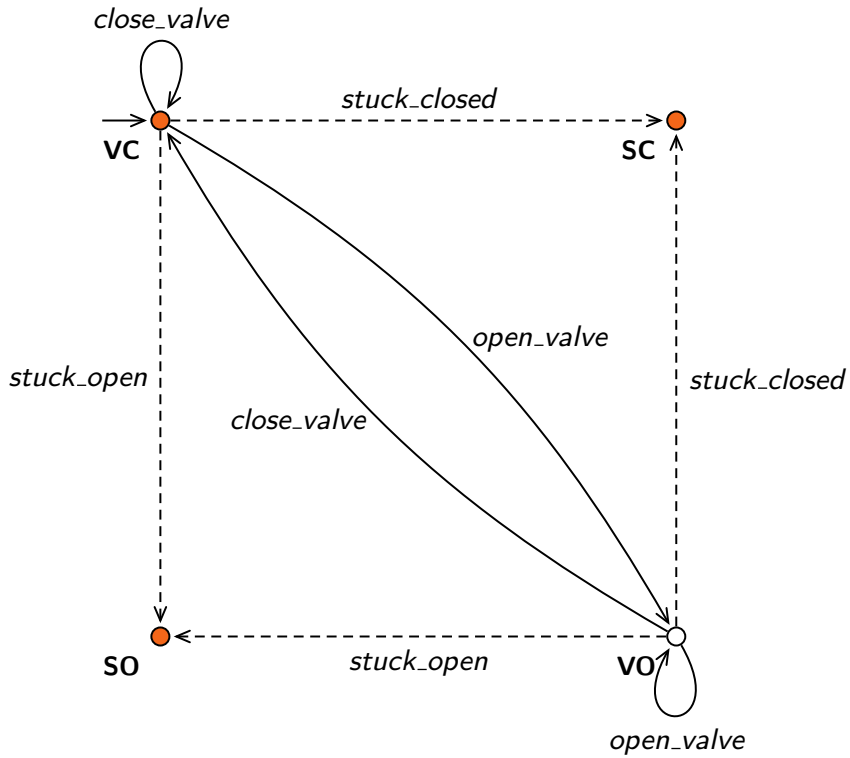
Exercise: Pump-valve example

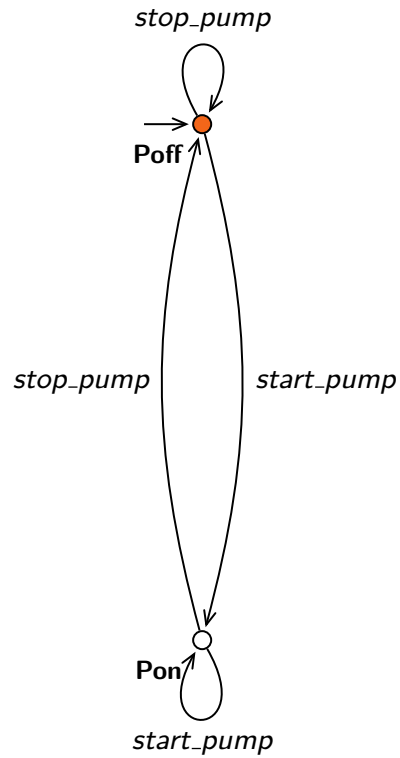




Express the requirement: prevent that the pump is operating while the valve is closed.

Answer: Pump-valve example





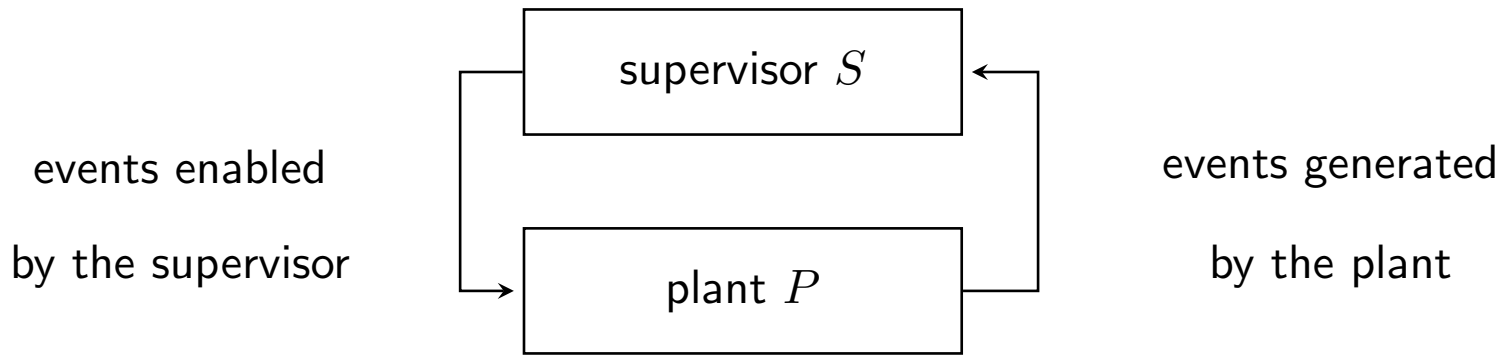
Prevent that the pump is operating while the valve is closed:

$\neg((\textbf{Valve.VC}\downarrow \vee \text{Valve.SC}) \wedge \textbf{Pump.Pon}\downarrow)$
$\rightarrow \{\ \textit{start\_pump}\ \} \Rightarrow (\textbf{Valve.VO}\downarrow \vee \textbf{Valve.SO}\downarrow)$

Basic data-based supervisory control problem

Prevent the system from getting to undesired states

- By disabling controllable events
- Based on the events generated by the plant



Basic data-based synthesis

- allow plant automata with discrete variables with finite domain:
  - Boolean
  - ranged integer type `int[0..11]`
  - enumeration type

  More restrictions apply; see

  `http://cif.se.wtb.tue.nl/tools/datasynth.html`
- network of plant automata first needs to be transformed into single plant automaton

Basic data-based synthesis problem

For a plant a supervisor is required such that:

1. controlled system is non-blocking

   It is possible to reach a marked state from all reachable states in the controlled system.

2. controlled system is controllable

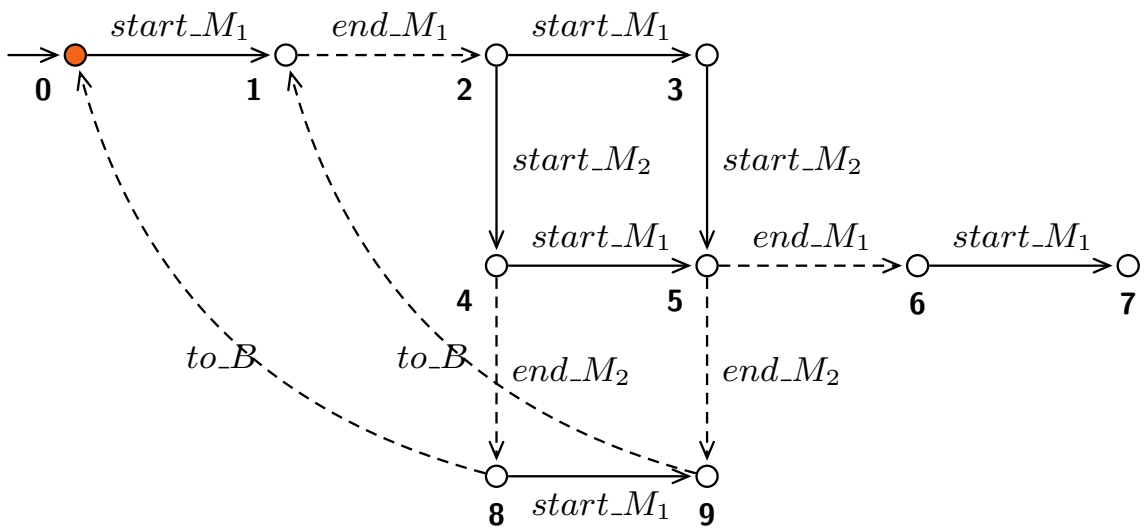
   For all reachable states in the controlled system, the uncontrollable events that are enabled in the same state in the uncontrolled system are still possible in the controlled system. In other words, uncontrollable events are not restricted.

3. controlled system is maximally permissive

   The controlled system permits all safe, controllable, and non-blocking behaviors.

High-level approach basic synthesis

Question: How does basic synthesis algorithm work?

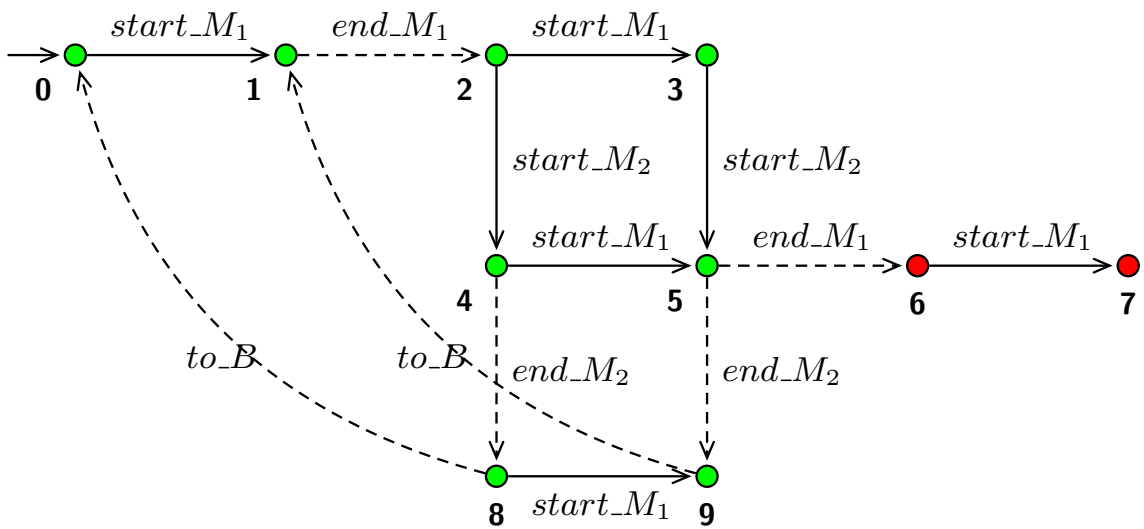


Repeat until resulting system stabilizes:

- Identify blocking states (green is non-blocking, red is blocking)
- Identify bad states
- Remove controllable events leading into bad states

Basic synthesis - non-blocking states

Question: How does basic synthesis algorithm work?

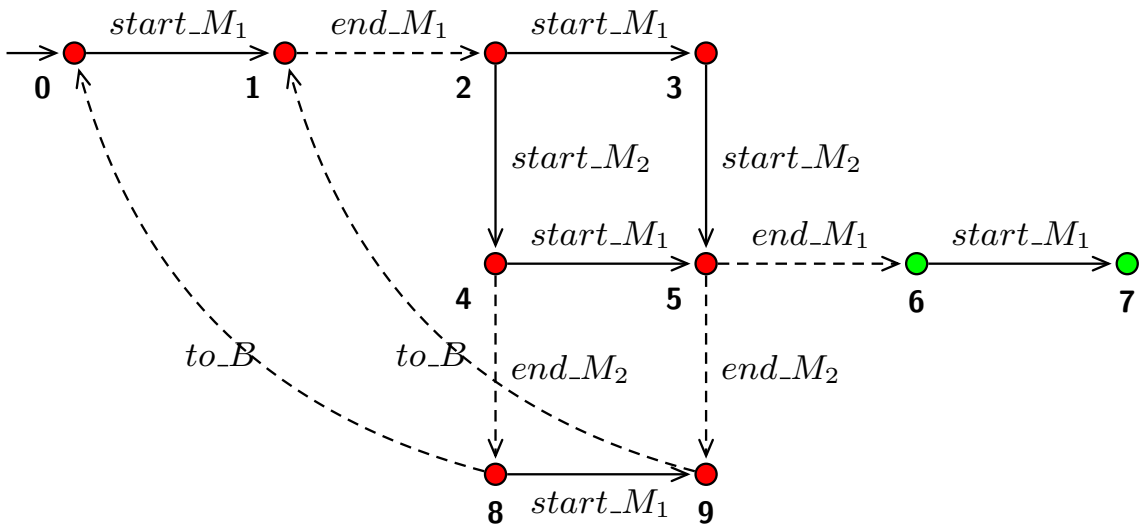


Repeat until resulting system stabilizes:

- Identify blocking states (green is non-blocking, red is blocking)
- Identify bad states
- Remove controllable events leading into bad states

Basic synthesis – bad states

Question: How does basic synthesis algorithm work?

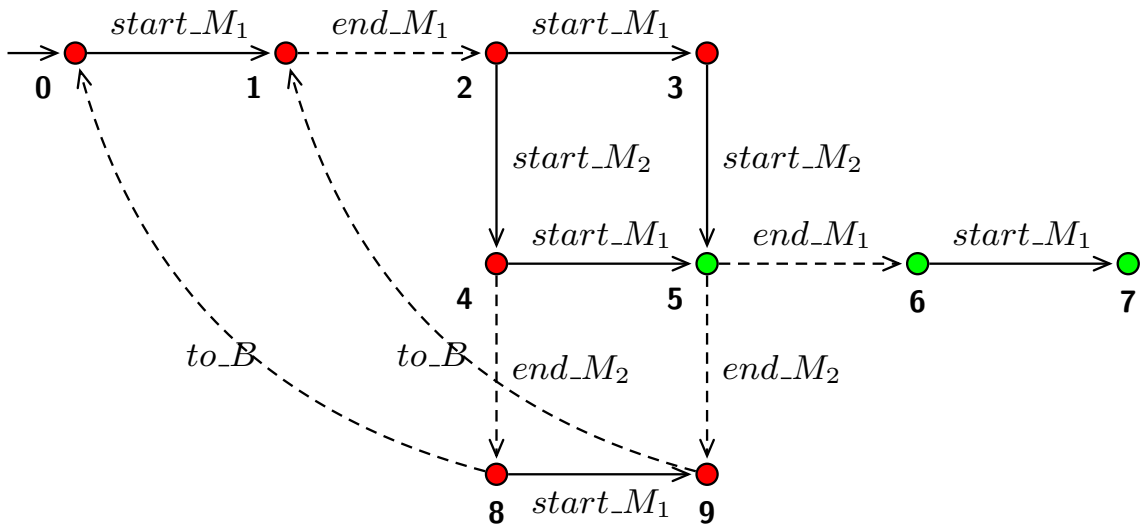


Repeat until resulting system stabilizes:

- Identify blocking states
- Identify bad states (green is bad state, red is safe state)
- Remove controllable events leading into bad states

Basic synthesis – bad states

Question: How does basic synthesis algorithm work?

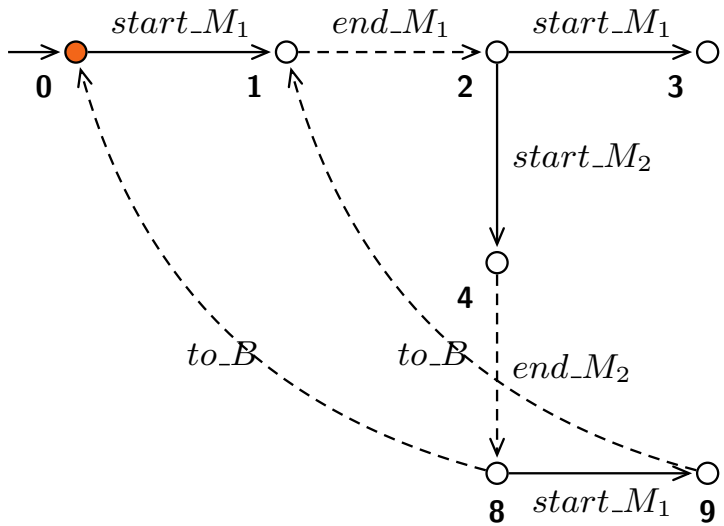


Repeat until resulting system stabilizes:

- Identify blocking states
- Identify bad states (green is bad state, red is safe state)
- Remove controllable events leading into bad states

Basic synthesis – remove controllable events

Question: How does basic synthesis algorithm work?

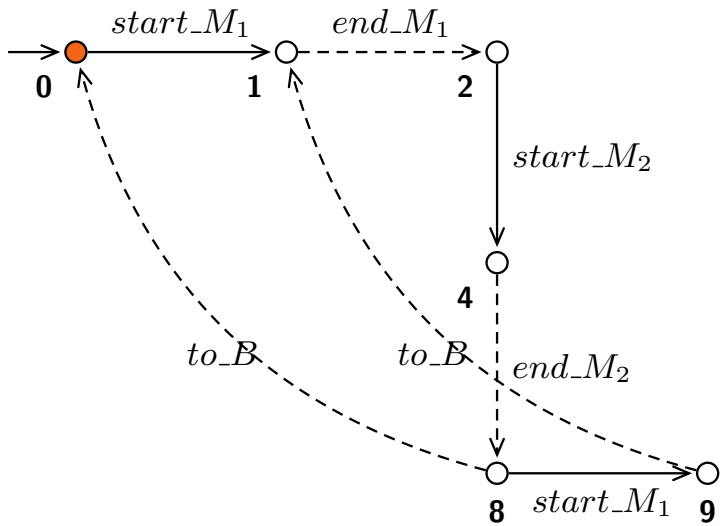


Repeat until resulting system stabilizes:

- Identify blocking states
- Identify bad states
- Remove controllable events leading into bad states

Basic synthesis – repetition

Question: How does basic synthesis algorithm work?



Repeat until resulting system stabilizes:

- Identify blocking states
- Identify bad states
- Remove controllable events leading into bad states

Basic data-based synthesis

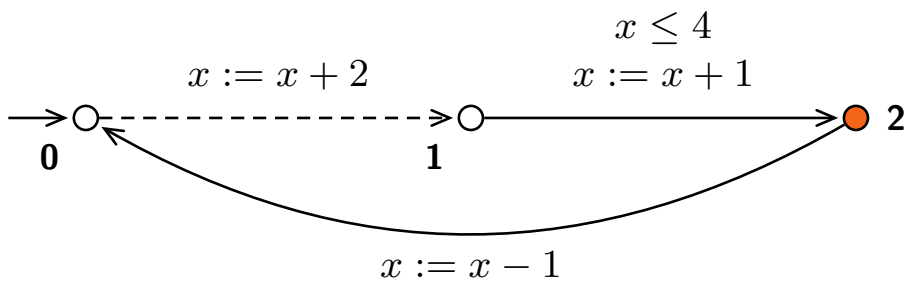
- Adapt the guards of the plant automaton repeatedly until the situation arises that no more changes occur.

- Each of these iterations consists of three phases:
  1. Compute nonblocking conditions for all locations of the plant.
  2. Compute bad state conditions for all locations.
  3. Adapt guards of transitions with controllable events to obtain an adapted automaton.

Nonblocking for automata with variables

- Nonblocking depends on the actual value of the variables in the location.
- Associate with each location an expression that states for which combination sof values of variables the location is nonblocking.

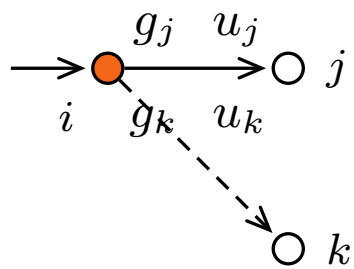
Exercise: Give nonblocking condition for each location



- In general, not easy to "guess"

Computing nonblocking conditions

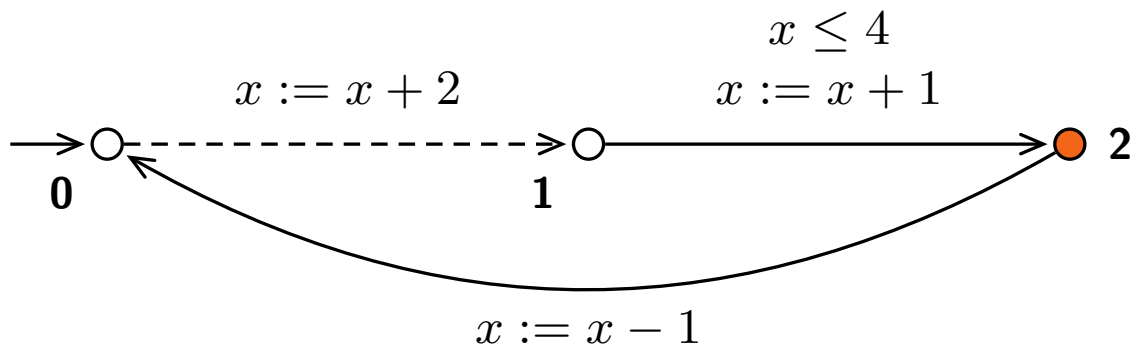
- Determining the nonblocking conditions is done iteratively
- Label all marker states with nonblocking condition $true$ and all non-marker states with nonblocking condition $false$.
- In each next iteration: (N_i is nonblocking condition of location i)



$$N_i := N_i \vee (g_j \wedge N_j[u_j]) \vee (g_k \wedge N_k[u_k]) \vee \cdots$$

- $N_j[u_j]$ is N_j where all occurrences of the variables are replaced by the expressions they are assigned in the update u_j
- Always simplify the resulting predicates!

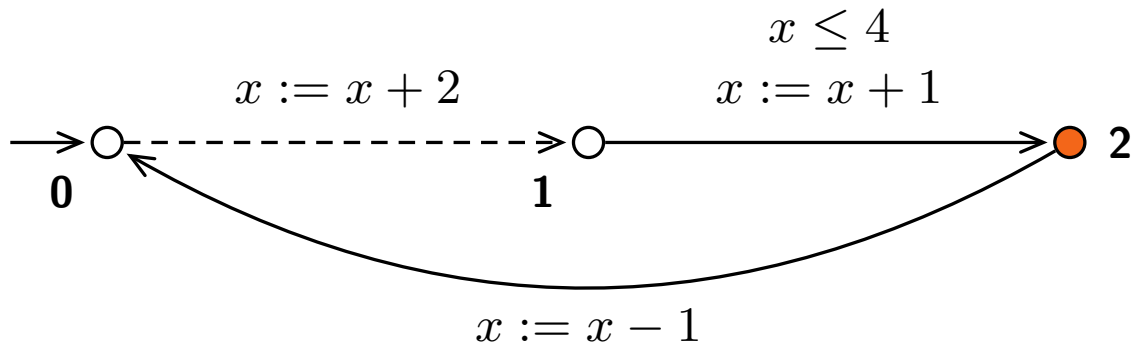
Exercise, computed solution



Location 0	Location 1	Location 2
false	*false*	*true*

$$N_i := N_i \vee (g_j \wedge N_j[u_j]) \vee (g_k \wedge N_k[u_k]) \vee \cdots$$

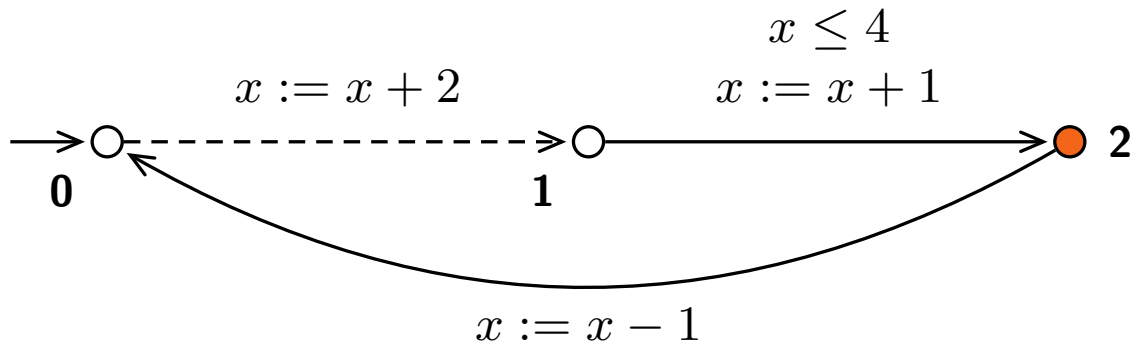
Computed solution, iteration 2



Location 0	Location 1	Location 2
$false$	$false$	$true$
$false \vee (true \wedge false[x := x+2])$ $= false$	$false \vee (x \leq 4 \wedge true[x := x+1])$ $= x \leq 4$	$true$

$$N_i := N_i \vee (g_j \wedge N_j[u_j]) \vee (g_k \wedge N_k[u_k]) \vee \cdots$$

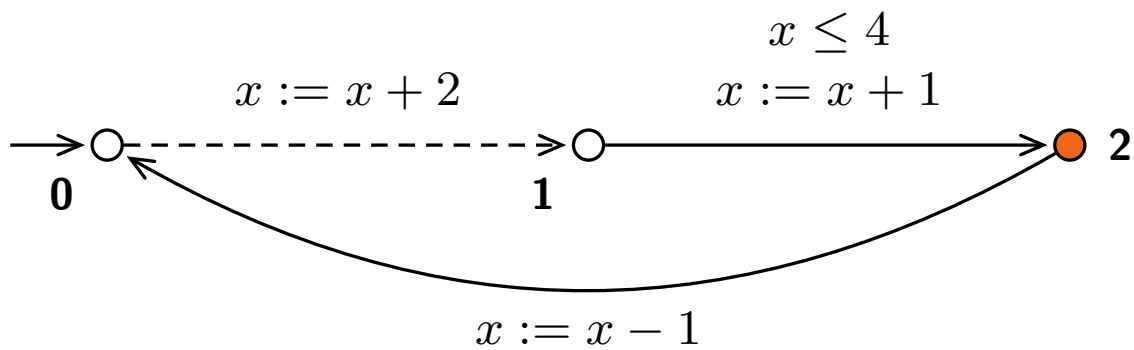
Computed solution, iteration 3



Location 0	Location 1	Location 2
$false$	$false$	$true$
$false$	$x \leq 4$	$true$
$false \vee (true \wedge x \leq 4[x := x+2])$ $= x \leq 2$	$x \leq 4 \vee (x \leq 4 \wedge true[x := x+1])$ $= x \leq 4$	$true$

$$N_i := N_i \vee (g_j \wedge N_j[u_j]) \vee (g_k \wedge N_k[u_k]) \vee \cdots$$

Computed solution, iteration 4

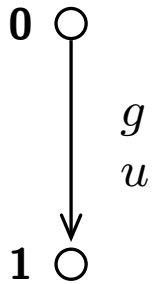


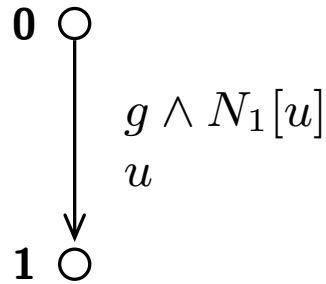
Location 0	Location 1	Location 2
$false$	$false$	$true$
$false$	$x \leq 4$	$true$
$x \leq 2$	$x \leq 4$	$true$
$x \leq 2$	$x \leq 4$	$true$

$$N_i := N_i \vee (g_j \wedge N_j[u_j]) \vee (g_k \wedge N_k[u_k]) \vee \cdots$$

Bad state conditions

- purpose of nonblocking conditions





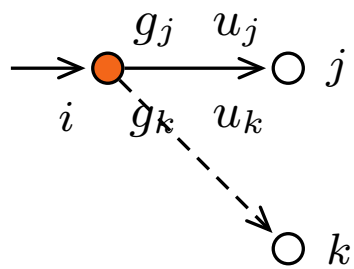
- not allowed for uncontrollable events to enforce the required restrictions by adapting the guard
- such restrictions are propagated backwards until an edge with a controllable event is encountered (for which the guard can be restricted)
- achieved by iterative bad state condition computation

Computing bad state conditions

- start from plant automaton and computed nonblocking conditions
- initial bad state condition for each location is the logical negation of the nonblocking condition of that location:

$$B_i := \neg N_i$$

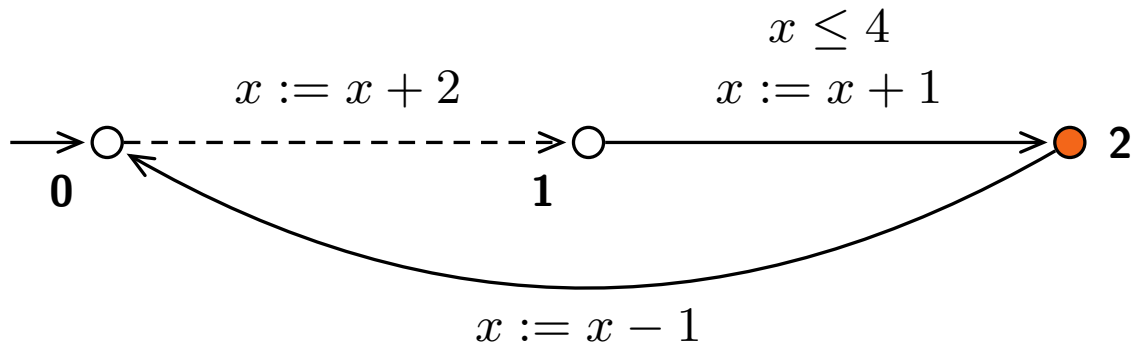
- In each next iteration:



$$B_i := B_i \vee (g_k \wedge B_k[u_k]) \vee \cdots$$

- Only for uncontrollable events!

Computed bad state conditions, solution



Location 0	Location 1	Location 2
$x > 2$	$x > 4$	$false$
$x > 2$	$x > 4$	$false$

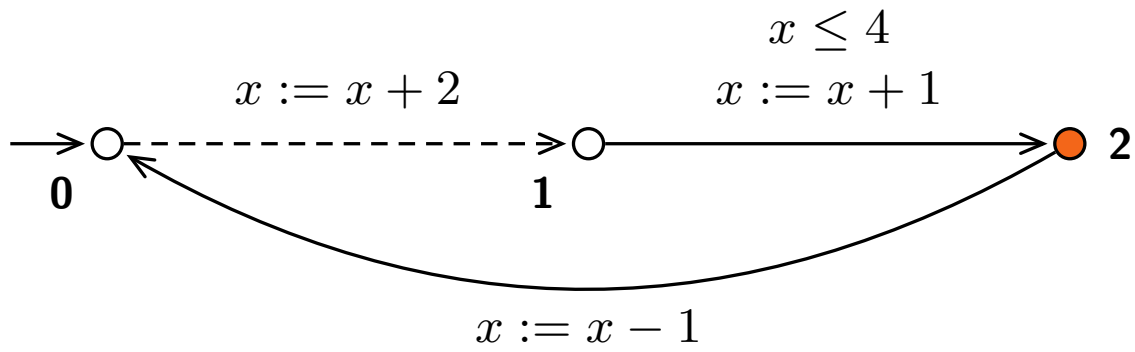
$$B_i := B_i \vee (g_k \wedge B_k[u_k]) \vee \cdots$$

Adapts guards

- use computed bas state conditions; these express which combinations of values of variables should be avoided in a specific location, taking into account the limitation that guards of uncontrollable events may not be altered.
- guards of the edges with a controllable event are adapted:

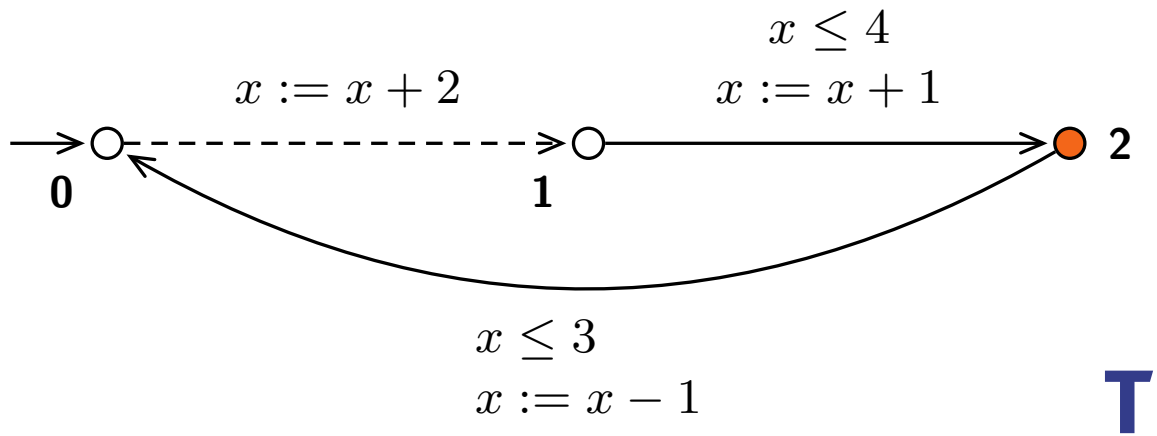
$$g_j := g_j \wedge \neg B_j[u_j] \wedge \cdots$$

Adapted guards, solution

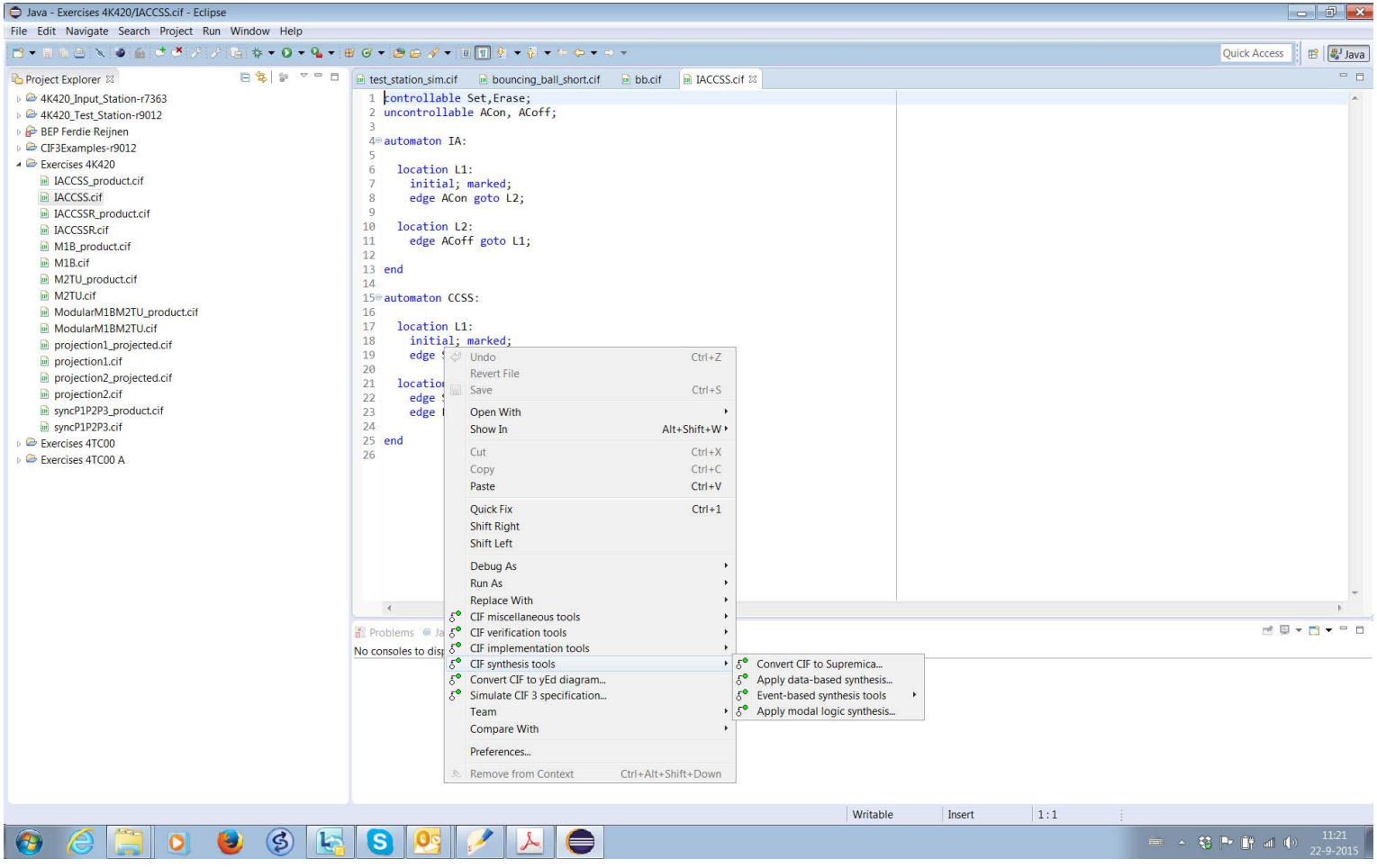


Location 0	Location 1	Location 2
$x > 2$	$x > 4$	$false$

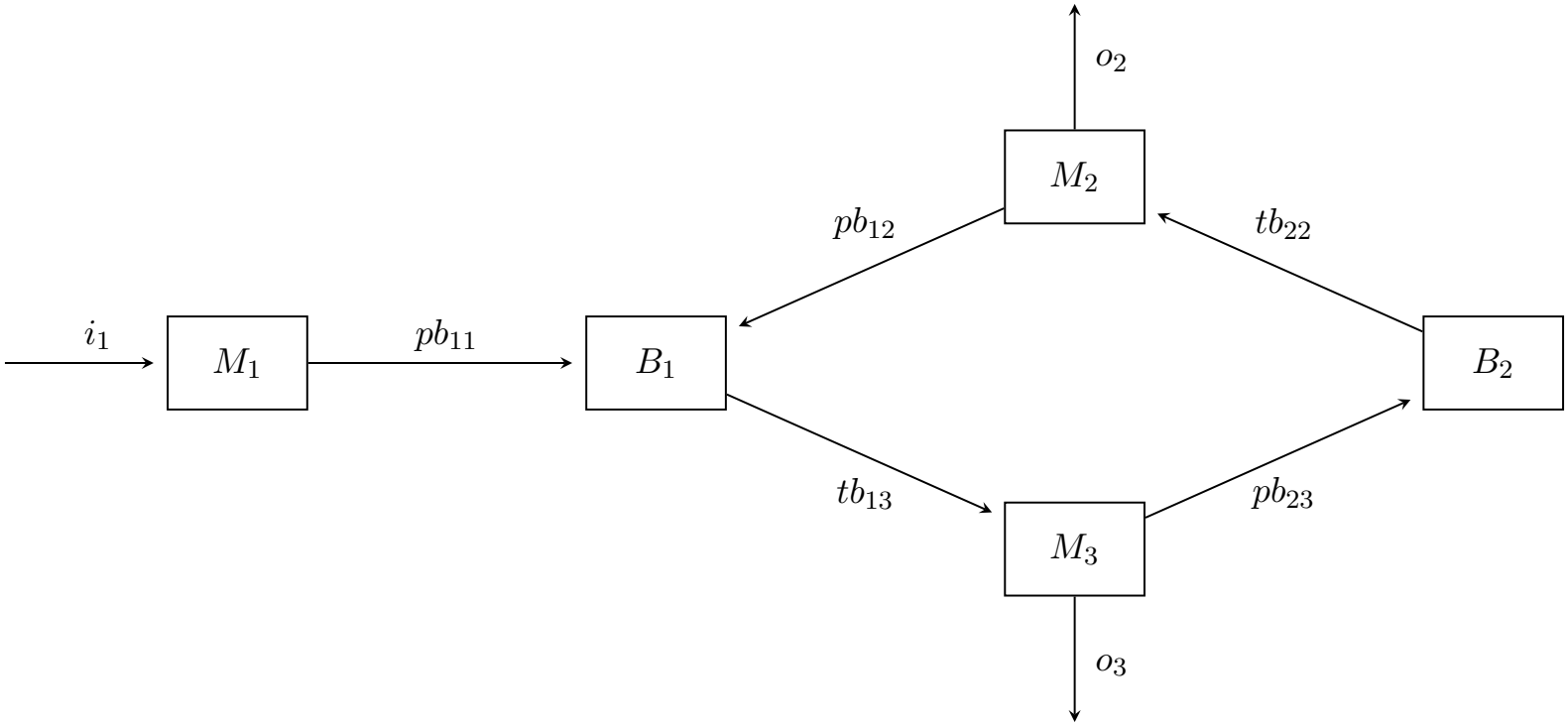
$$g_j := g_j \wedge \neg B_j[u_j] \wedge \cdots$$



CIF support


Java - Exercises 4K420/IACCSS.cif - Eclipse
File Edit Navigate Search Project Run Window Help
Project Explorer
4K420_Input_Station-r7363
4K420_Test_Station-r9012
BEP Ferdie Reijnen
CIF3Examples-r9012
Exercises 4K420
IACCSS_product.cif
IACCSS.cif
IACCSSR_product.cif
IACCSSR.cif
M1B_product.cif
M1B.cif
M2TU_product.cif
M2TU.cif
ModularM1BM2TU_product.cif
ModularM1BM2TU.cif
projection1_projected.cif
projection1.cif
projection2_projected.cif
projection2.cif
syncP1P2P3_product.cif
syncP1P2P3.cif
Exercises 4TC00
Exercises 4TC00 A
test_station_sim.cif
bouncing_ball_short.cif
bb.cif
IACCSS.cif
controllable Set,Erase;
uncontrollable ACon, ACoff;
automaton IA:
location L1:
initial; marked;
edge ACon goto L2;
location L2:
edge ACoff goto L1;
end
automaton CCSS:
location L1:
initial; marked;
end
Undo Ctrl+Z
Revert File
Save Ctrl+S
Open With
Show In Alt+Shift+W
Cut Ctrl+X
Copy Ctrl+C
Paste Ctrl+V
Quick Fix Ctrl+1
Shift Right
Shift Left
Debug As
Run As
Replace With
CIF miscellaneous tools
CIF verification tools
CIF implementation tools
CIF synthesis tools
Convert CIF to yEd diagram...
Simulate CIF 3 specification...
Team
Compare With
Preferences...
Remove from Context Ctrl+Alt+Shift+Down
Convert CIF to Supremica...
Apply data-based synthesis...
Event-based synthesis tools
Apply modal logic synthesis...
Problems
No consoles to disp
Writable
Insert
1 : 1
11:21
22-9-2015

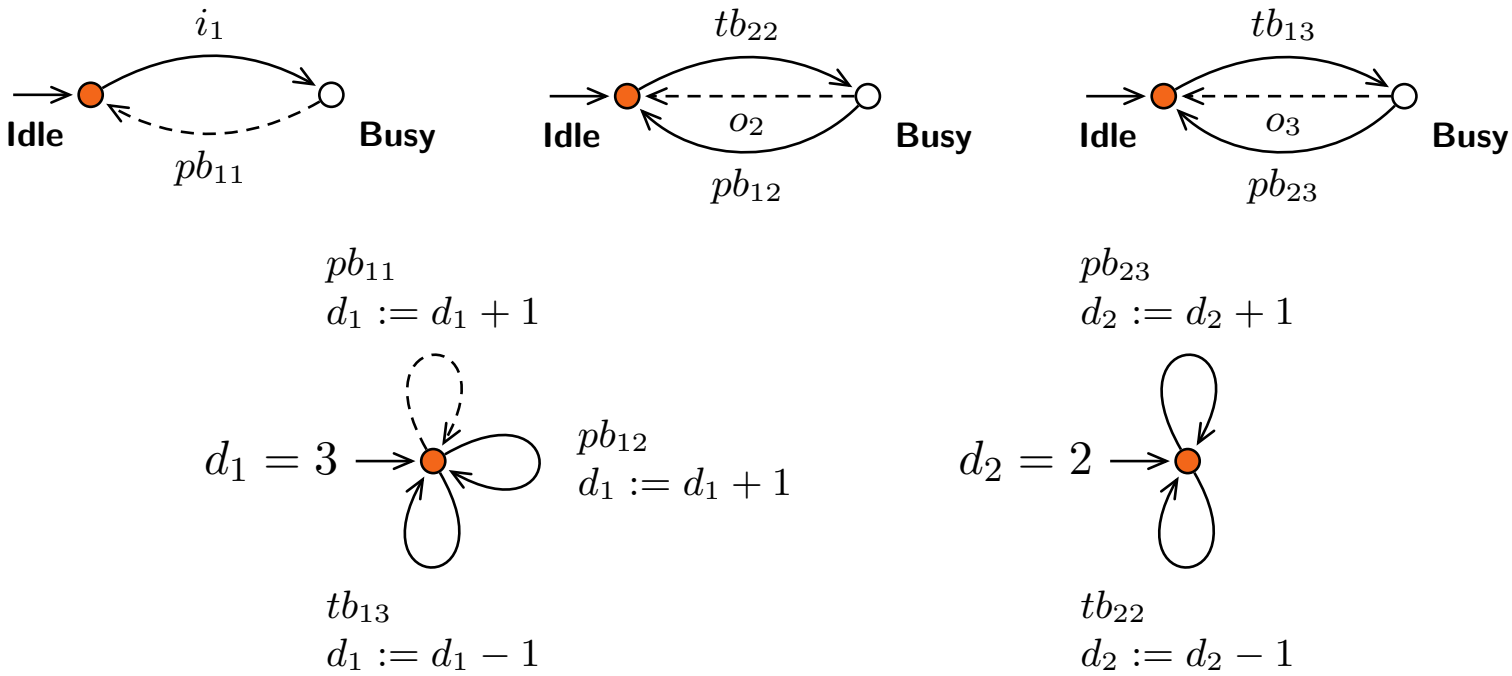
Example



Uncontrollable: pb_{11}, o_2 and o_3

Requirements: No overflow and no underflow: $3 \leq d_1 \leq 16$ and $2 \leq d_2 \leq 8$.

Models



Requirements:

```
1    requirement invariant 3 <= B1.d and B1.d <= 16;
2    requirement invariant 2 <= B2.d and B2.d <= 8;
```

Resulting supervisor

S

```
1   alphabet i1, tb22, pb12, tb13, pb23;
2   location:
3     initial;
4     marked;
5     edge i1 when (B1.d = 8 or B1.d = 10) and (B2.d = 8 and M1.Idle) or ((B1.d =
       8 or B1.d = 10)
6                   and (B2.d = 4 and M1.Idle) or (B1.d = 8 or B1.d = 10) and ((B2
       .d = 2 or
7                   B2.d = 6) and ... ;
8     edge pb12 when B1.d = 8 and (B2.d = 8 and M2.Busy) or B1.d = 8 and (B2.d = 4
        and ...;
9     edge pb23 when B1.d = 16 and B2.d = 4 and (M1.Idle and M3.Busy) or (B1.d =
       16 and ...;
10    edge tb13 when B1.d = 16 and (B2.d = 8 and M3.Idle) or (B1.d = 16 and (B2.d
       = 4 and ...;
11    edge tb22 when B1.d = 16 and B2.d = 8 and (M1.Idle and M2.Idle) or (B1.d =
       16 and ...;
12  end
```

Model-based Engineering of Supervisory Controllers

Michel Reniers

M.A.Reniers@tue.nl

IPA Formal Methods Course

2018.06.14

TU/e Technische Universiteit Eindhoven University of Technology