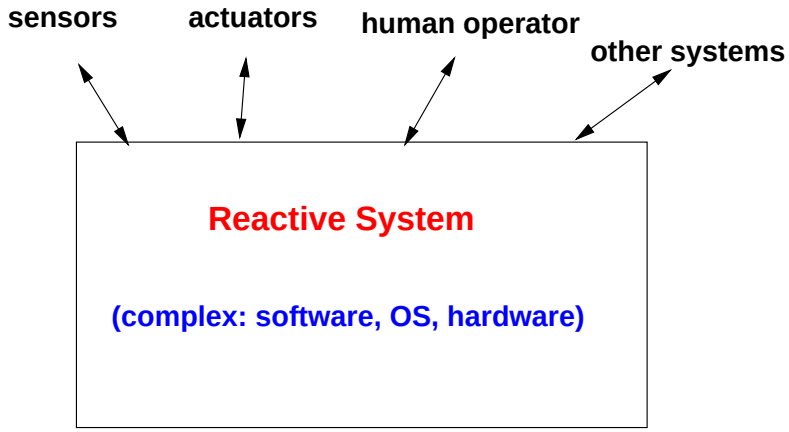
Modelling and validation of Real-time embedded systems: the asynchronous approach

Susanne Graf

Verimag/CNRS -Grenoble - France

Problem: Modelling and validation of Real-time embedded systems

Reactive systems - external view:



Reactive systems - internal view: a concurrent system

- concurrent activities with internal communication
- distributed and/or scheduled execution
- conflicts for accessing resources

Verification Problem

Given a Model S (software/hardware system) and given a Property φ, prove that S satisfies φ

$$S \models \varphi$$

A model-checking framework is given by

1. Model: A formalism for the description of the Semantics of systems
2. Property: A language for describing Properties
3. Verification: A precise definition of the satisfaction relation $\models$ and an algorithm allowing to prove that $S \models \varphi$ (or alternatively $S \subseteq \varphi$, $S \preceq \varphi$, $\models S \Rightarrow \varphi$) or to give evidence that $S \not\models \varphi$ (counter-example)

Overview on verification frameworks

1. Model = set of all *behaviours* of a system

   - Linear model: a behaviour is a sequence of states, actions, properties
     $\Rightarrow$ Model = a language
   - branching model: a behaviour is a tree $\Rightarrow$ Model = a tree language

Both kind of models can be represented by a labelled *graph*

2. "Atomic" properties

   - assertions control locations, values of variables, messages in queues, ...): decorate states with "atomic propositions" (*Kripke structures*)
   - names (and parameters) of actions: decorate transitions with "action names" (*Labelled transition systems*)

There exist transformations between both types of models

3. Properties

- Logic specifications: Properties are given by formulas of a temporal logic interpreted as sets of behaviours
- Operational specifications: Properties are given by an *abstract model*

4. Verification frameworks

- Structural approach (restricted interpretation of model-checking): compute the set of states satisfying each sub-formula
- Automata based approach (automata on infinite words or trees): transform Specifiations into a model and use autamata based methods for deciding “language” containment
- axiomatic approach: transform the satisfaction problem into a validity check and provide an appropriate axiomatisation

Comparison of models

- In the above classification we consider non structured models: distinction between execution sequences or trees
- here we are interested in distinguishing different kinds of *structured* models : different parallel composition, different communication timed versus untimed,....

Focus with respect to verification methods

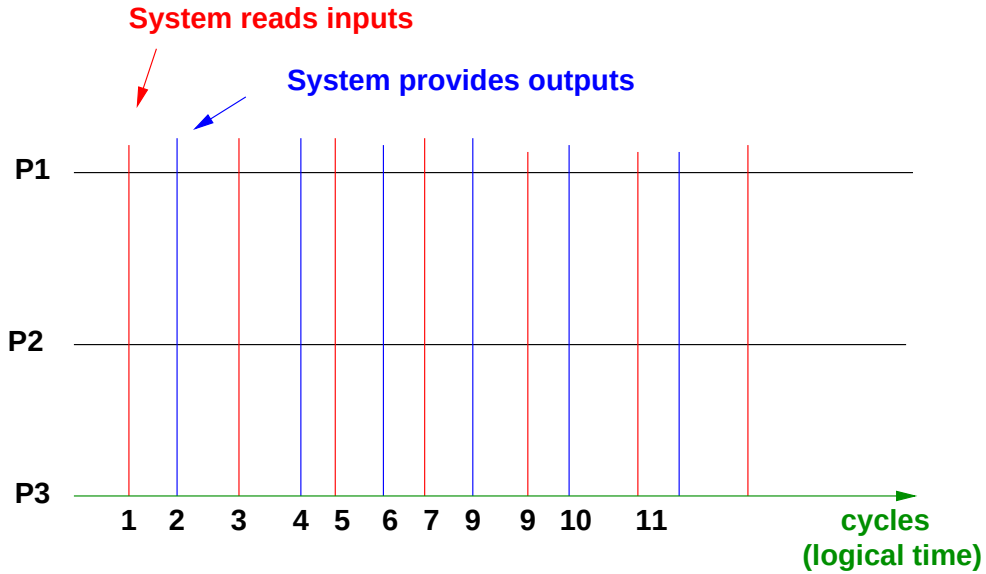
- We focus on enumerative automata-based verification technology (reachability problem)

Overview

1. Part I: Asynchronous versus synchronous system modelling
   - characteristics of synchronous and asynchronous models
   - some asynchronous frameworks
   - mastering the complexity of model-checking
2. Part II: IF - language & toolset for verifying asynchronous systems
   - IF language concepts: functional, non functional, property expression
   - A toolbox for the verification of timed, asynchronous systems
3. Part III: A Case study : Arinae 5 flight software
   - Combined synchronous/asynchronous modelling
   - verification of timing properties under resource constraints

Synchronous Approach: abstract functional semantics

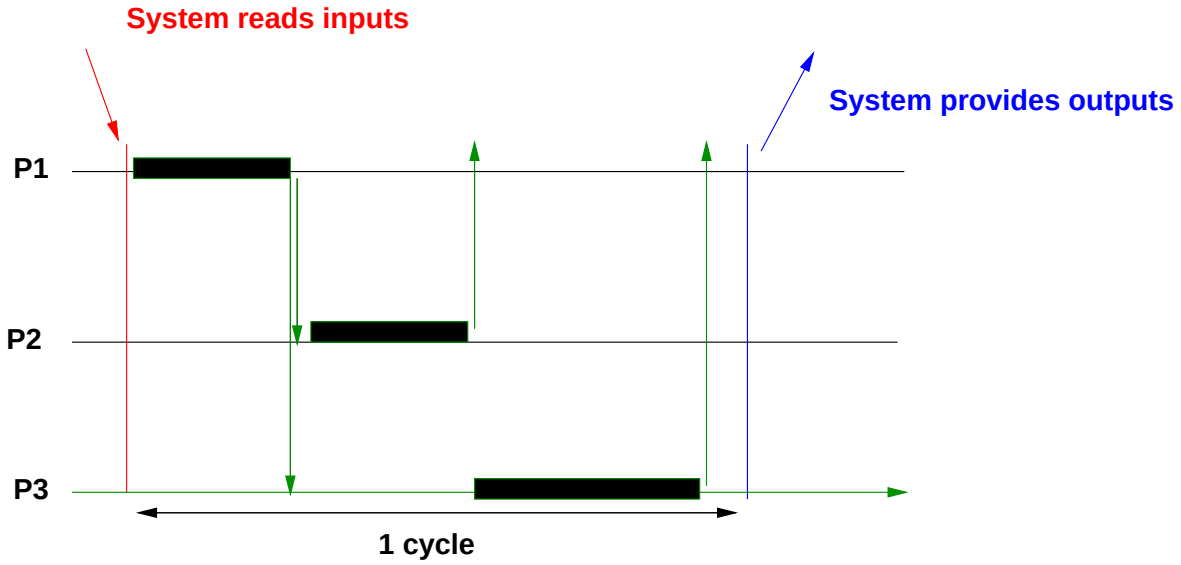
Cyclic system progress: read input at cycle start - output at cycle end



All “processes” in the system progress at the same speed.

- Functionalities needed less often “idle” in some (many, most) cycles
- Functionalies requiring more than one cycle to be computed, must be cut in smaller pieces (e.g. synchronous circuit design)
- Execution order “within a cycle” not exposed
- Can be extended to multicyclic systems (avoids idling)

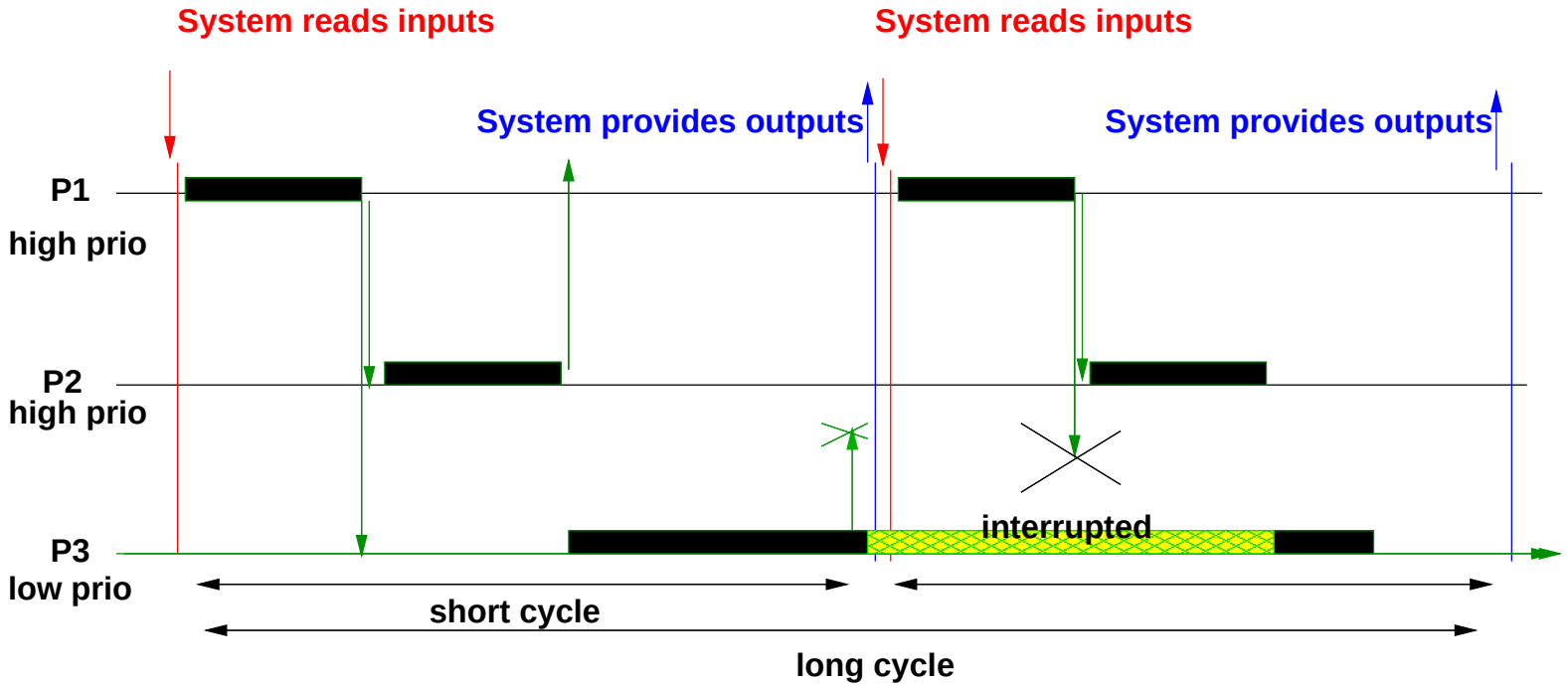
Synchronous Approach: implementation and correctness by construction for centralized implementation



- when maximal cycle execution time is known, execute the cycle every "tick" of some system clock (real-time view)
- execution order of system internal activities:
  - synchronous languages (Lustre, Esterel): all orders are functionally equivalent
  - state charts (UML): correct functionality may depend on order

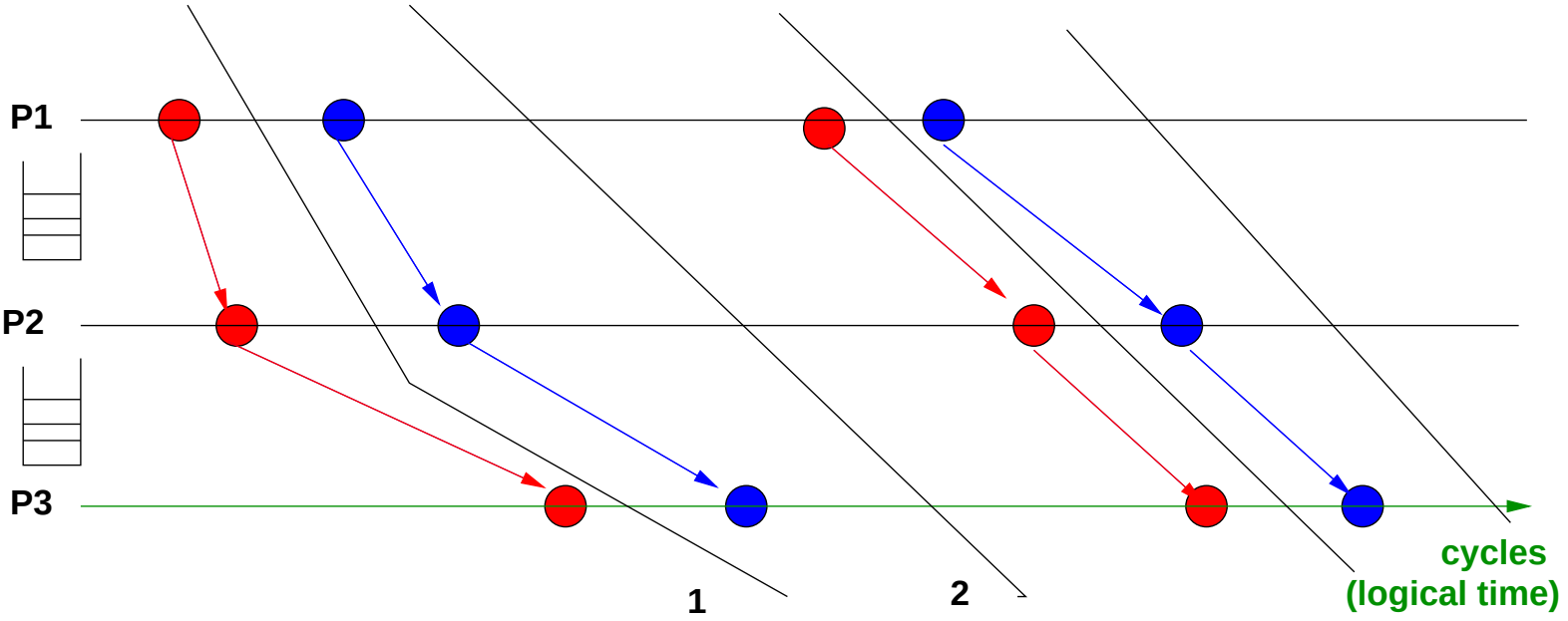
Extensions of the synchronous Approach

1. RMS scheduling can be used when long cycles are low priority



2. Distributed implementation: more complicated (discussed later)

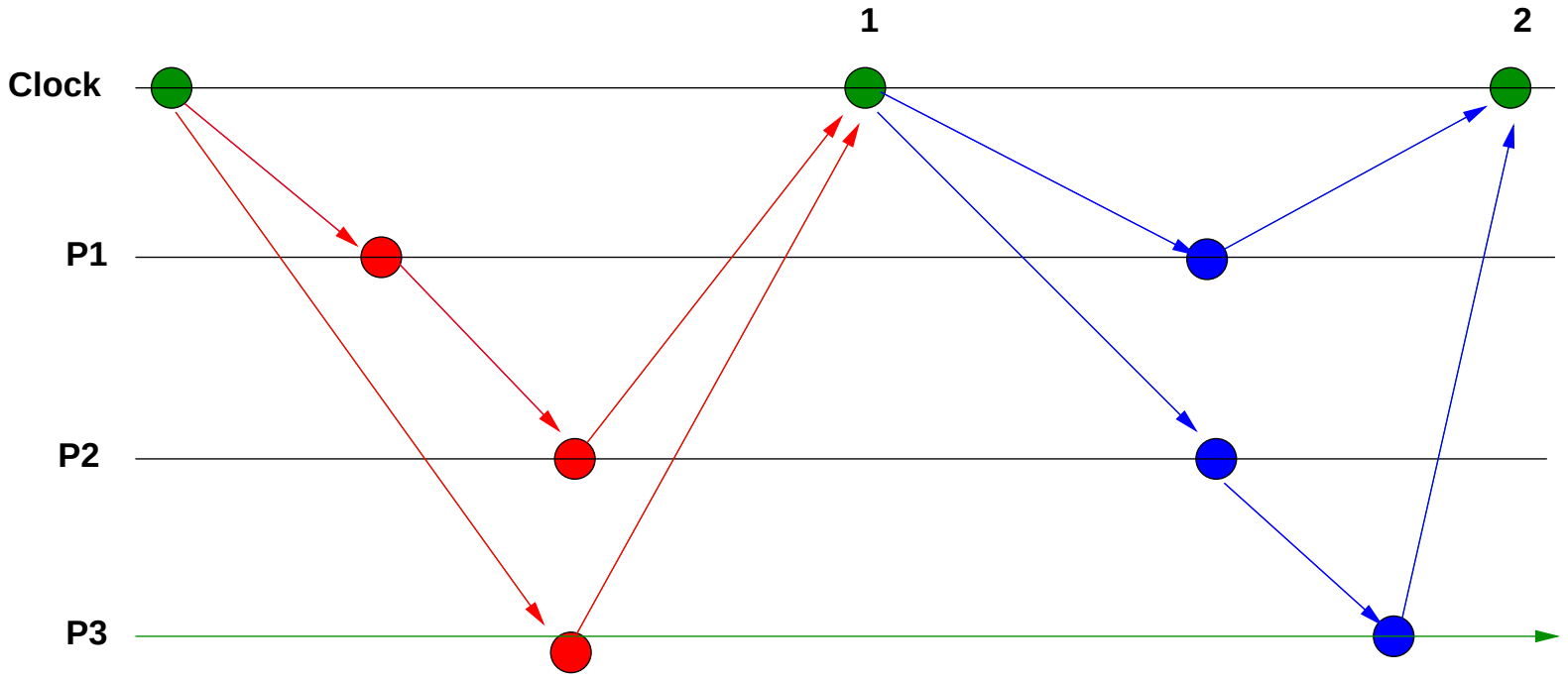
3. Kahn networks: acyclic communication through non lossy (unbounded) fifo buffers



functionality can be analysed assuming synchrony

Semantic view

The event structure associated with a synchronous model contains strong synchronisation points



Summary: Main characteristics of the synchronous approach

- Stimuli of the environment taken into account at fixed pace: events of a faster environment are lost
- Reaction of the system must follow the same pace
- Deterministic (abstract cycle based view)
- Functional verification is easy:
  - system = transfer function defined by a cycle (no concurrency)
  - no need to consider timing issues

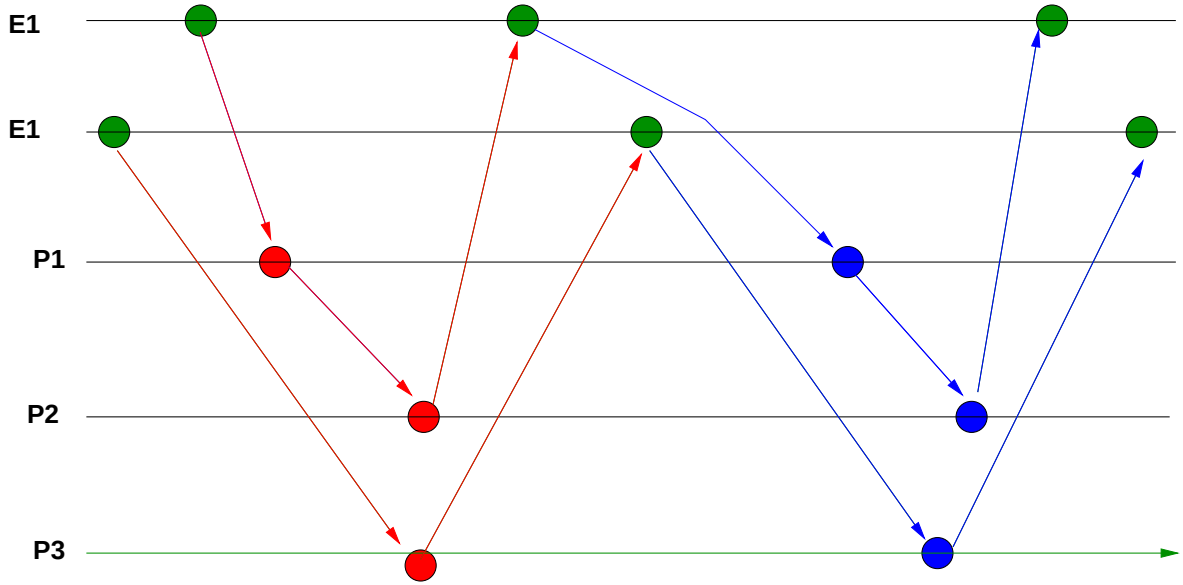
Asynchronous Approach

Main characteristics

- Computations triggered by on need (occurrence of a stimulus), stimuly can be irregular
- Non deterministic specifications (environment and system)
- Buffered and/or lossy communication
- No global synchronisation points required, each component progresses “at it’s own pace”
- More freedom to provide a “correct” implementation (scheduling)

Semantic view

The event structure associated with an asynchronous model may contain no strong synchronisation points



When external stimuli are buffered, $P1, P2$ may progress infinitly faster than $P3$. Validation is harder

- Proof of progress based on fairness assumptions
- Proof of bounded response time based on timing assumptions

Some framworks for asynchronous modelling

- Petrinets
- Process algebras such as CCS, CSP, Lotos
- Finite state machines communicating through (buffered) messages
- Modelling formalisms such as Estelle, SDL, UML
- Timed automata, ...

When is asynchronous modelling useful?

- System with a non physical environment (continuous paradigm is inappropriate)
- Systems where QoS properties are dominant (variable communication delays, variable request flow)
- When energy is an issue
- For the analysis of non functional properties of a single cycle of a synchronous system

Asynchronous models

The relevant structuring features

- variables and assignments

  $\leadsto$ structured states instead of simple control states (e.g. timed automata)

- parallel compositions

  $\leadsto$ complex control structure and non determinism

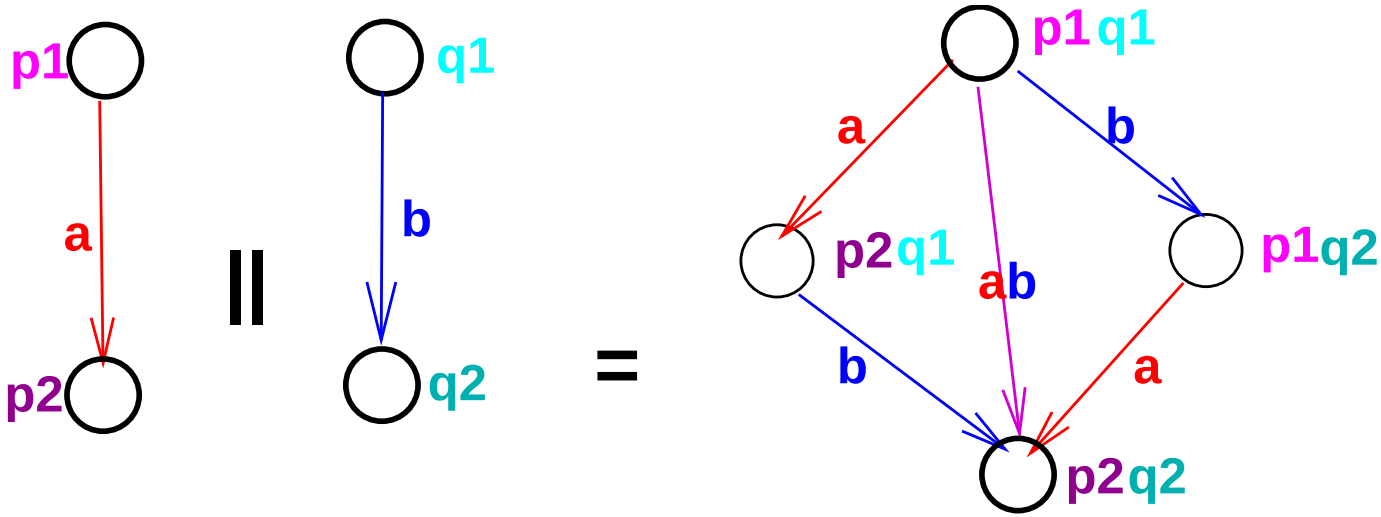
- communication mechanisms

  - synchronisation (rendez-vous, abstract)

  - buffered communication

Parallel Composition of Labelled Transition systems

- asynchronous parallel composition
- interleaving parallel composition
- synchronous parallel composition
- synchronising parallel composition
- buffered communication by message passing



Asynchronous Product

$$(Q_1, q_{10}, \mathcal{A}_1, \rightarrow_1) || (Q_2, q_{20}, \mathcal{A}_2, \rightarrow_2) =$$
$$(Q_1 \times Q_2, (q_{10}, q'_{20}), \mathcal{A}_1 \cup \mathcal{A}_2 \cup (\mathcal{A}_1 \times \mathcal{A}_2), \rightarrow)$$

where

$$\frac{q_1 \stackrel{a}{\rightarrow}_1 q'_1}{(q_1, q_2) \stackrel{a}{\rightarrow} (q'_1, q_2)} \quad , \quad \frac{q_2 \stackrel{b}{\rightarrow}_2 q'_2}{(q_1, q_2) \stackrel{b}{\rightarrow} (q_1, q'_2)}$$

Notice that the asynchronous product is only meaningful in combination with some form of *hide* operator which forbids some of the transitions. A typical example is communication by shared variables.

Synchronous Product

$$(Q_1, q_{10}, \mathcal{A}_1, \rightarrow_1) \times (Q_2, q_{20}, \mathcal{A}_2, \rightarrow_2) =$$
$$(Q_1 \times Q_2, (q_{10}, q'_{20}), (\mathcal{A}_1 \times \mathcal{A}_2), \rightarrow)$$

where

$$\frac{q_1 \overset{a}{\rightarrow}_1 q'_1 \ \& \ q_2 \overset{b}{\rightarrow}_2 q'_2}{(q_1,q_2) \overset{(a,b)}{\rightarrow} (q'_1,q'_2)}$$

Synchronised Product

Let be $\mathcal{A}_i$ of the form $\mathcal{A}_i^{nc} \cup \mathcal{A}_i^{c}$ where the first kind of actions are internal or non communication actions and the latter are potential communication actions

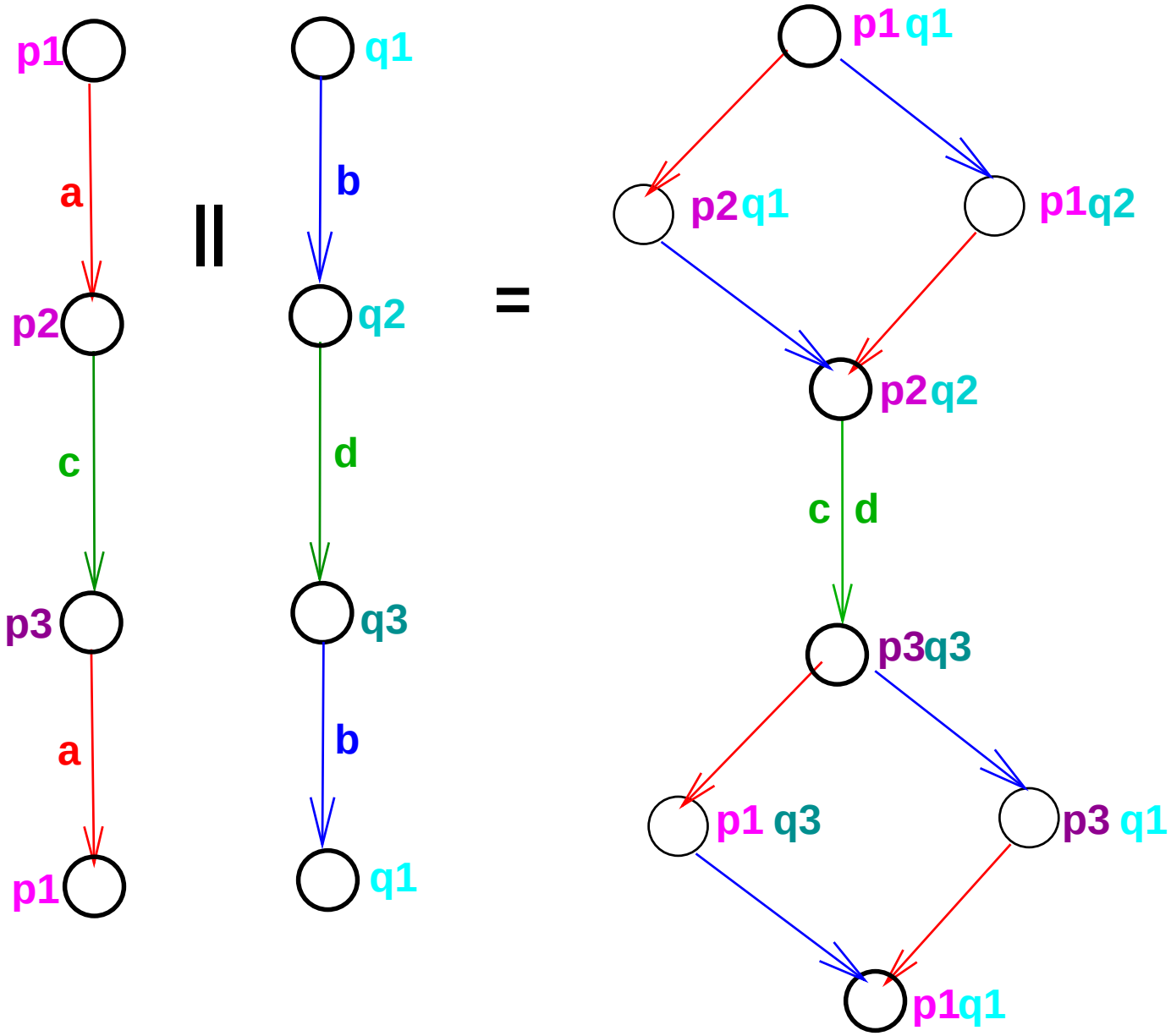
Let be $L \subseteq \mathcal{A}_1^c \times \mathcal{A}_2^c$

$$(Q_1, q_{10}, \mathcal{A}_1, \rightarrow_1) || _L (Q_2, q_{20}, \mathcal{A}_2, \rightarrow_2) =$$
$$(Q_1 \times Q_2, (q_{10}, q_{20}), \mathcal{A}_1^{nc} \cup \mathcal{A}_2^{nc} \cup L, \rightarrow)$$

where

$$\frac{q_1 \stackrel{a}{\rightarrow}_1 q_1',\ a \in \mathcal{A}_1^{nc}}{(q_1,q_2) \stackrel{a}{\rightarrow} (q_1',q_2)} \quad , \quad \frac{q_2 \stackrel{b}{\rightarrow}_2 q_2',\ b \in \mathcal{A}_2^{nc}}{(q_1,q_2) \stackrel{b}{\rightarrow} (q_1,q_2')} \quad , \quad \frac{q_1 \stackrel{a}{\rightarrow}_1 q_1' \ \&\ q_2 \stackrel{b}{\rightarrow}_2 q_2' \ \&\ (a,b) \in L}{(q_1,q_2) \stackrel{(a,b)}{\rightarrow} (q_1',q_2')}$$

Illustration: Synchronised product



Finite State machines communication through (unbounded) Fifo buffers

A Finte state machine is defined as $FSM = (Q, M, \rightarrow, B, q_0)$ where

- Q is a finite set of states and q_0 an initial state
- M is a set of "messages" sent or received
- B is a set of "buffers" through which messages are sent or received
- $\rightarrow \subseteq Q \times B \times \{!, ?\} \times Q$ is a transition relation that combines a message reception (input) or message emission (output) with a state change. We write $q \stackrel{B!m}{\rightarrow} q$ or $q \stackrel{B?m}{\rightarrow} q$

Semantics of systems of FSM communicating through buffers

The semantics of a system S of n finite state machines FSM_i on a a common set of messages M and k buffers B_i is defined by a global transition system $STE = (Q_1 \times ... \times Q_n \times (M^*)^k, \rightarrow, q_0)$ where

- initially all buffers are empty (not important)
- the transition of STE corresponds to a transition of exactly 1 FSM_i and
  - if it is a reception the consumption of the *first* message in the envolved Buffer B_i which must match with the consumed message . A reception is not possible if the buffer is empty
  - if it is an emission, the *append* of emmitted message to the involved buffer

  The notion of *first* and *append* depends on the communication policy

Exercice: formalize this informal definition for unbounded FIFO buffers

Results

- STE may have infinitly many states
- The questions if *buffer B_i has bounded length* or *system S has the finite buffer property* is undecidable for strict FIFO buffers
- Reachability of a state configuration is undecidable for unbounded FIFO buffers and decidable for lossy unbounded FIFO buffers

Systems with timing constraints: timed automata

A timed automaton is a tuple $TA = (Q, q_0, \mathcal{A}, X, \rightarrow)$, where

- Q is a set of control states with initial state q_0
- $\mathcal{A}$ is a set of transition labels
- $X = (x_1, ..., x_n)$ is a set of clocks
- $\rightarrow \subseteq Q \times (\mathcal{A} \times \gamma \times R) \times Q$ is a transition relation, where
  - $R \subseteq X$ is the set of *resetted* clocks
  - γ is the set of clock formulas of the form

$$\gamma ::= x \star c \mid x + c \star y + d \mid \neg\gamma \mid \gamma \wedge \gamma$$

where $\star \in \{<, \leq, =, >\geq\}$ and $c, d \in \mathcal{R}_{\geq 0}$

Semantics of timed automata

Interpret timed automata as labelled transition systems on the set of states $(Q \times \mathcal{E})$

Clock valuations $\mathcal{E}: \ X \rightarrow \mathcal{R}_{\geq 0}$

Transition relation is the smallest relation obtained by the following 2 rules:
Discrete transitions:

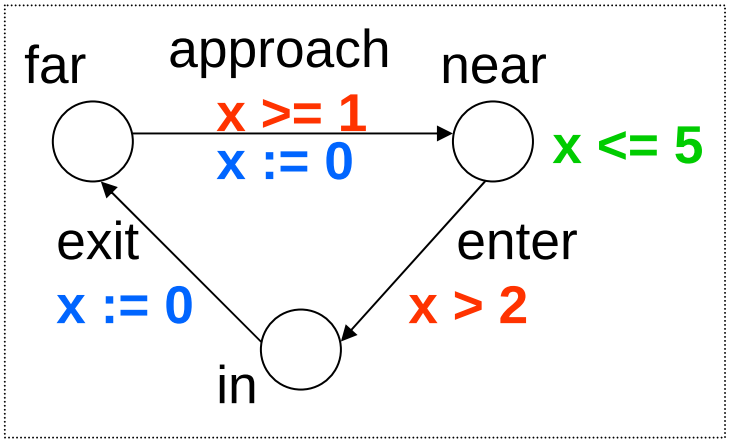
$$\frac{q \overset{a,\gamma,R}{\rightarrow} q' \ \wedge \ v \models \gamma \ \wedge \ v' = v[R := 0]}{(q, v) \overset{e}{\rightarrow} (q, v')}$$

Time passing transitions:

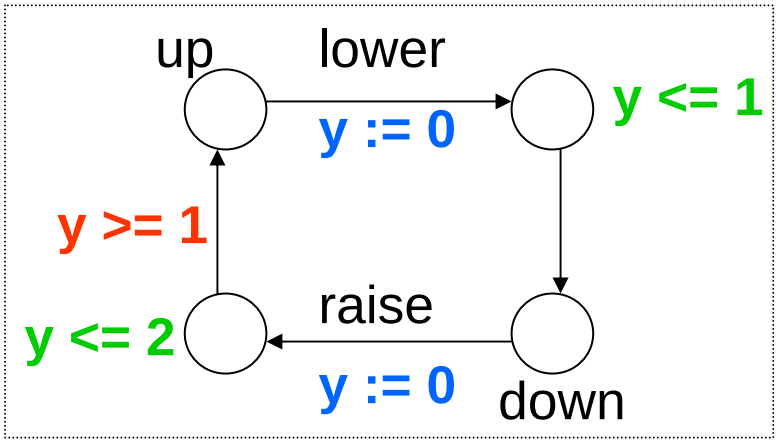
$$\frac{\exists \delta . \delta > 0 \wedge v' = v + \delta}{(q, v) \overset{time}{\rightarrow} (q, v')}$$

Notice that the set of states is uncountable

Timed Automata

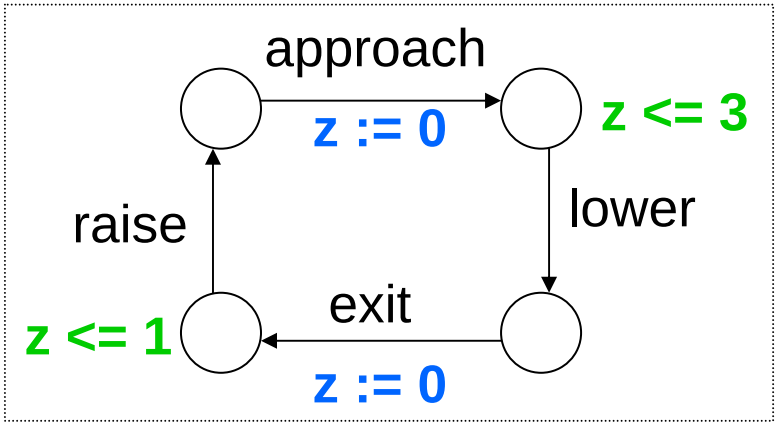


Train



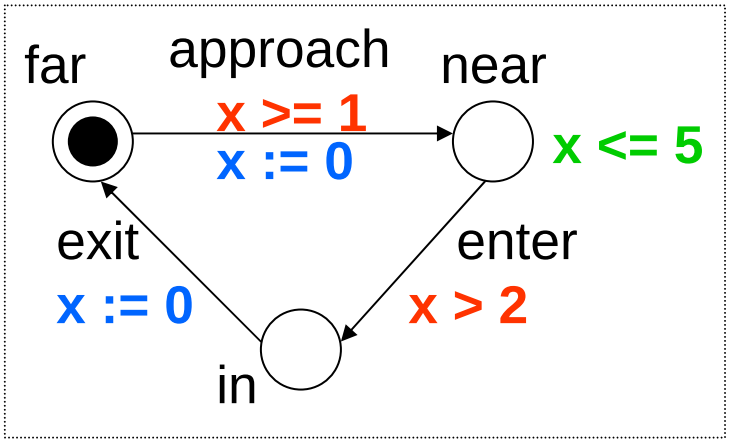
Gate

Controller

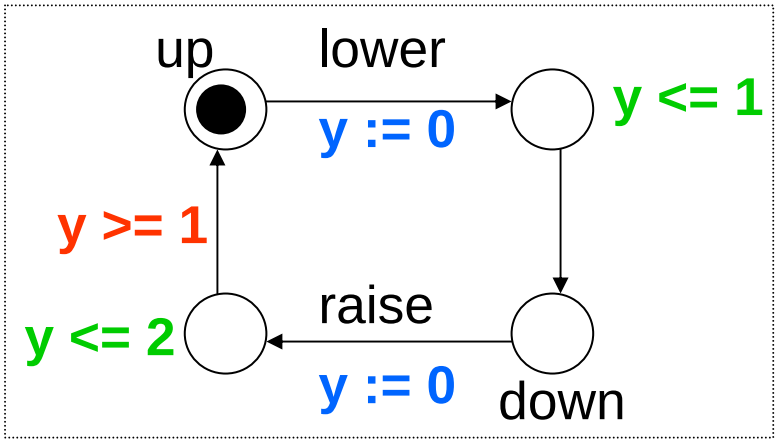


Slides of timed automata example provided by Stavros Tripakis

Timed Automata

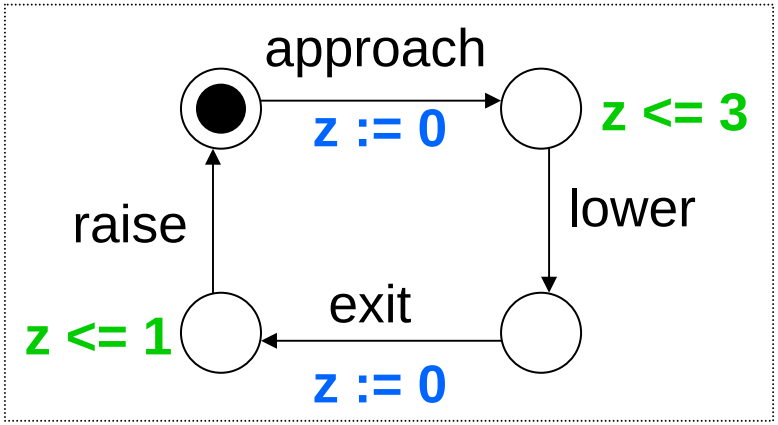


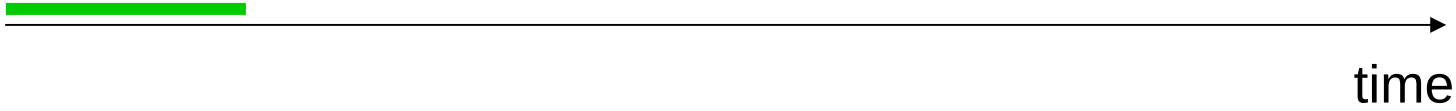
Train



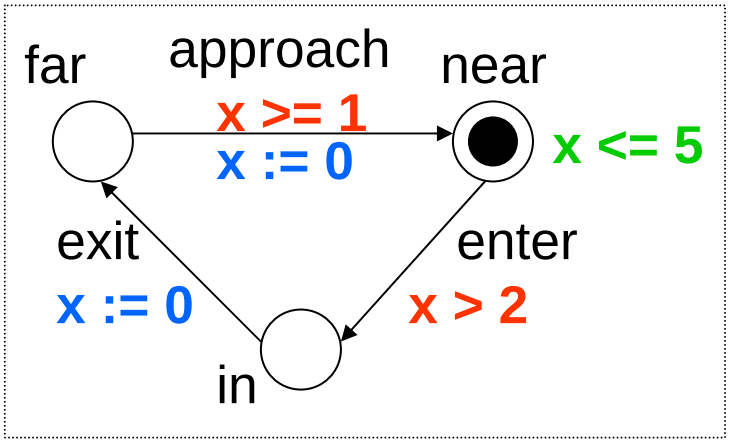
Gate

Controller

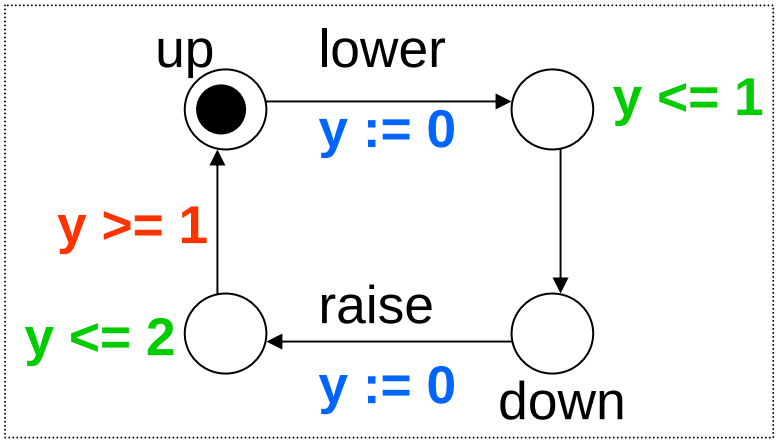




Timed Automata

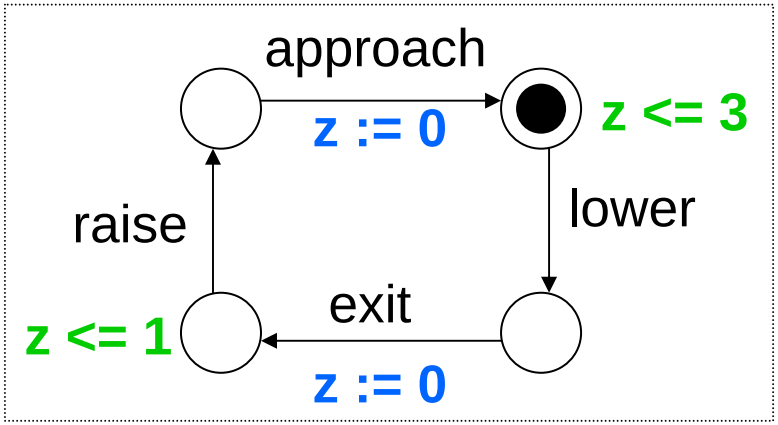


Train

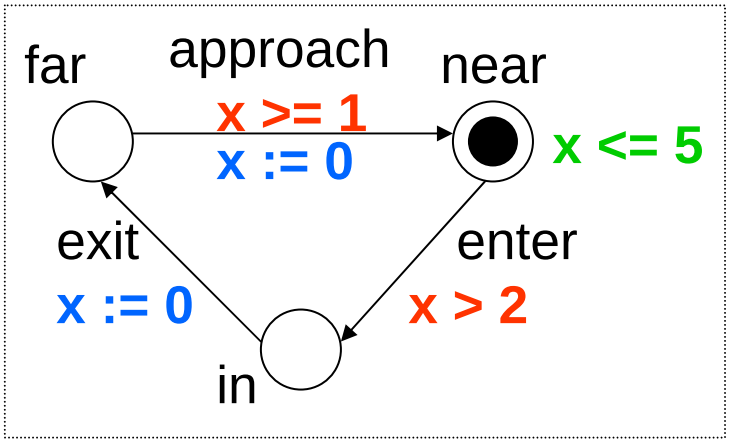


Gate

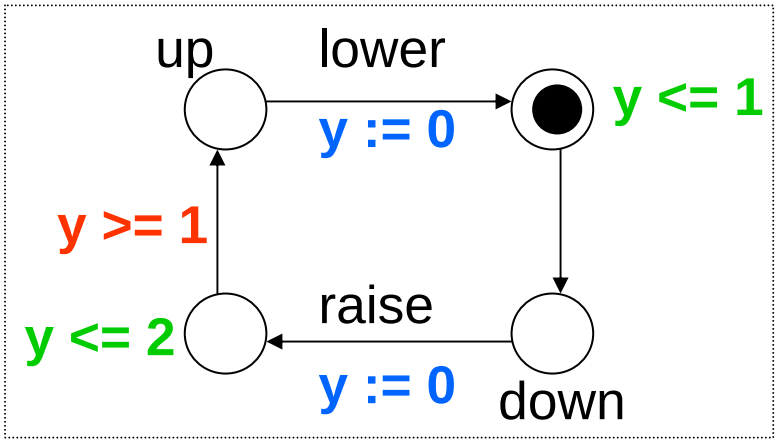
Controller



Timed Automata

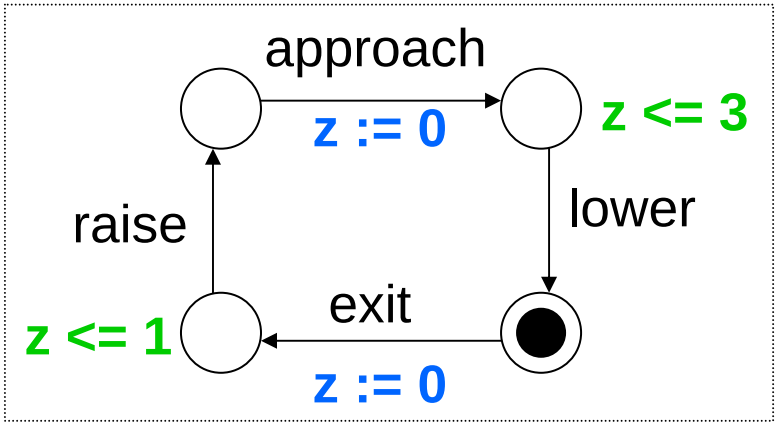


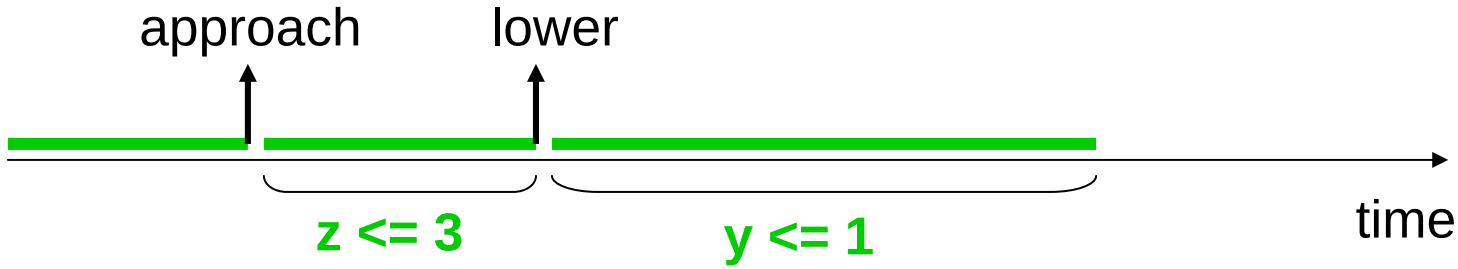
Train



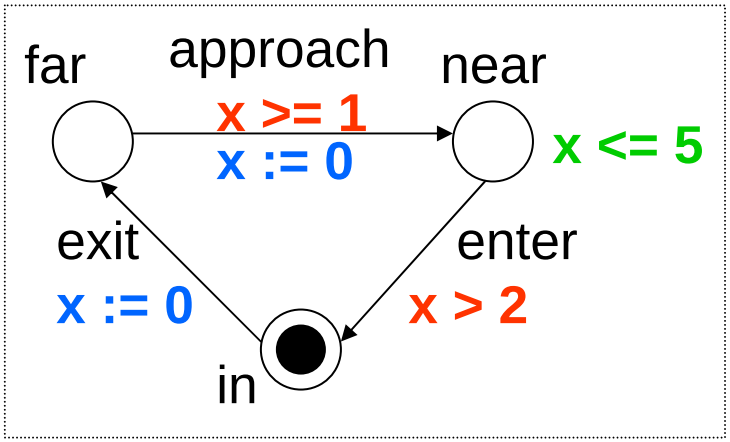
Gate

Controller

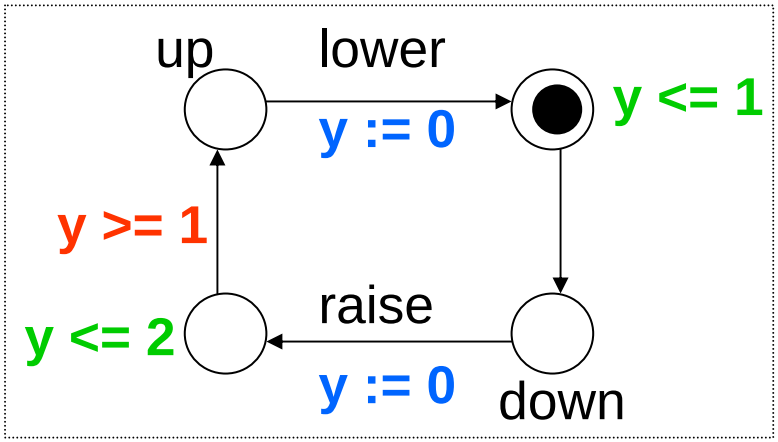




Timed Automata

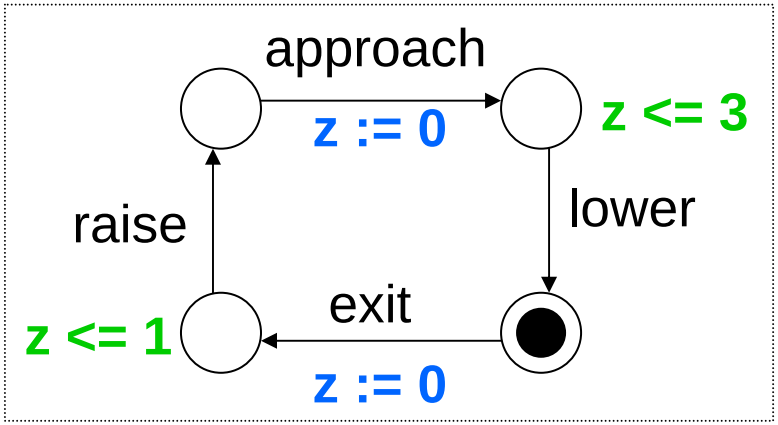


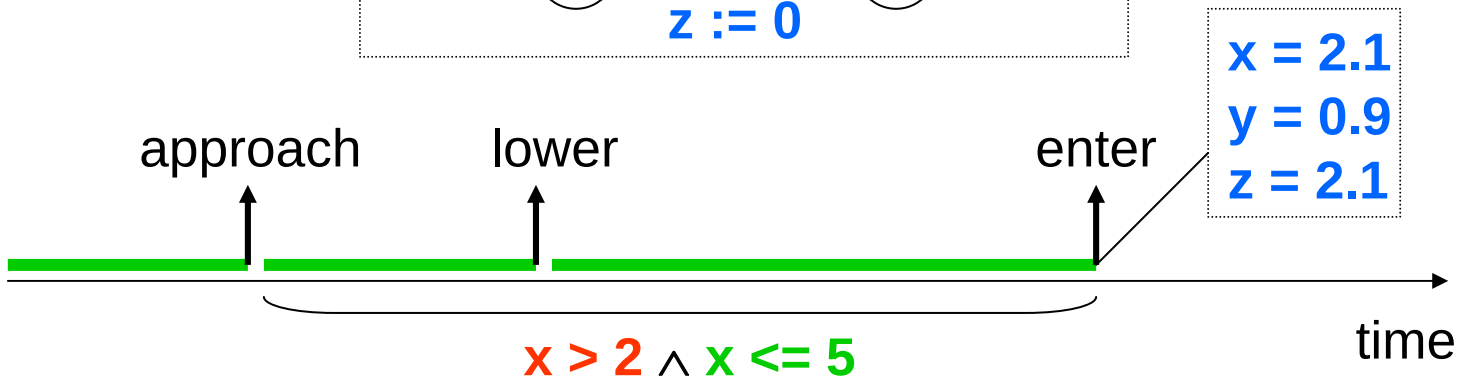
Train



Gate

Controller





Decidability results (for a single (finite state) timed automaton)

Theorem: the reachability problem is decidable

Consider the logic TCTL (standing for timed CTL), with terms obtained by the following grammar:

$$\varphi ::= p \mid \neg\phi \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \; \exists\mathcal{U} \; \varphi_2$$

Theorem: the verification problem for TCTL on timed automata is decidable

Theorem: the problem of bisimulation checking on timed automata is decidable

Making enumerative model-checking efficient: some pists

On-the-fly verification (enumerative, forward verification)

aim: reachability of a set of states in $\mathcal{K} = (Q, q_0, \rightarrow, \mathcal{P}, \mathcal{I})$. All we need is:

- a compact implicit representation of the global successor relation allowing to generate from a given global state all its successors
- start exploring in the initial state and keep in memory all so far reached states and some bit telling if it has already been explored or not

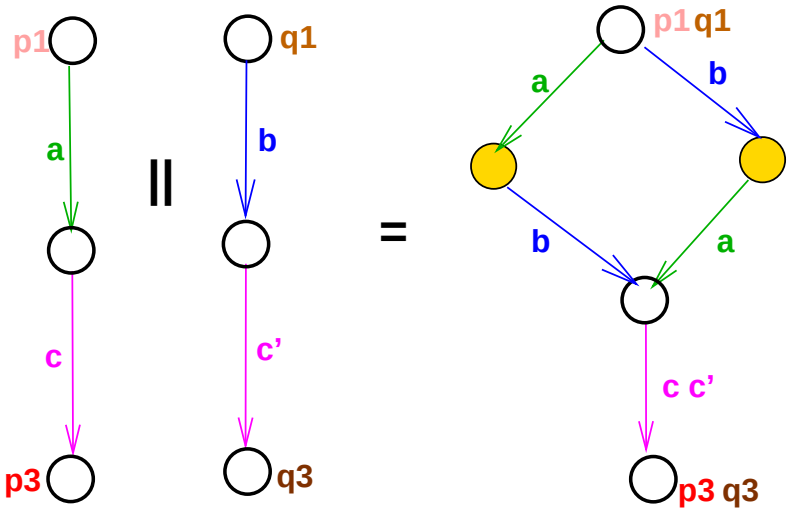
  Thus we only need to store the set of *reachable* states of Q, but not the transition relation $\rightarrow$
- the algorithm stops if either a unsafe has been reached or all states have been explored. In the case of a *depth first* traversal, one gets a *counter example* when an unsafe state is reached.

On-the-fly verification of general (linear safety) properties

- the algorithm can easily be extended to general safety properties by expressing the property as a safety automaton which is composed with the system model on-the-fly
- the number of states to be stored is $O(\|Q\| \times \|Q_\varphi\|)$ and the number of transitions to be traversed $O(\|\rightarrow\| \times \|\rightarrow_\varphi\|)$ but the number may much smaller when the property is not verified

Partial order reductions (exploit *true concurrency*)

Observation: The presence of *independent* transitions which are enabled at the same time blow up the model unnecessarily



If actions a and b are independent and their target states "not observed by the property", not all interleavings need to be explored.

So called true concurrency models (*Petri Nets, Event structures,..*) avoid this redundancy by not making interleavings explicit. Nevertheless, there exists not many verification methods working directly such models.

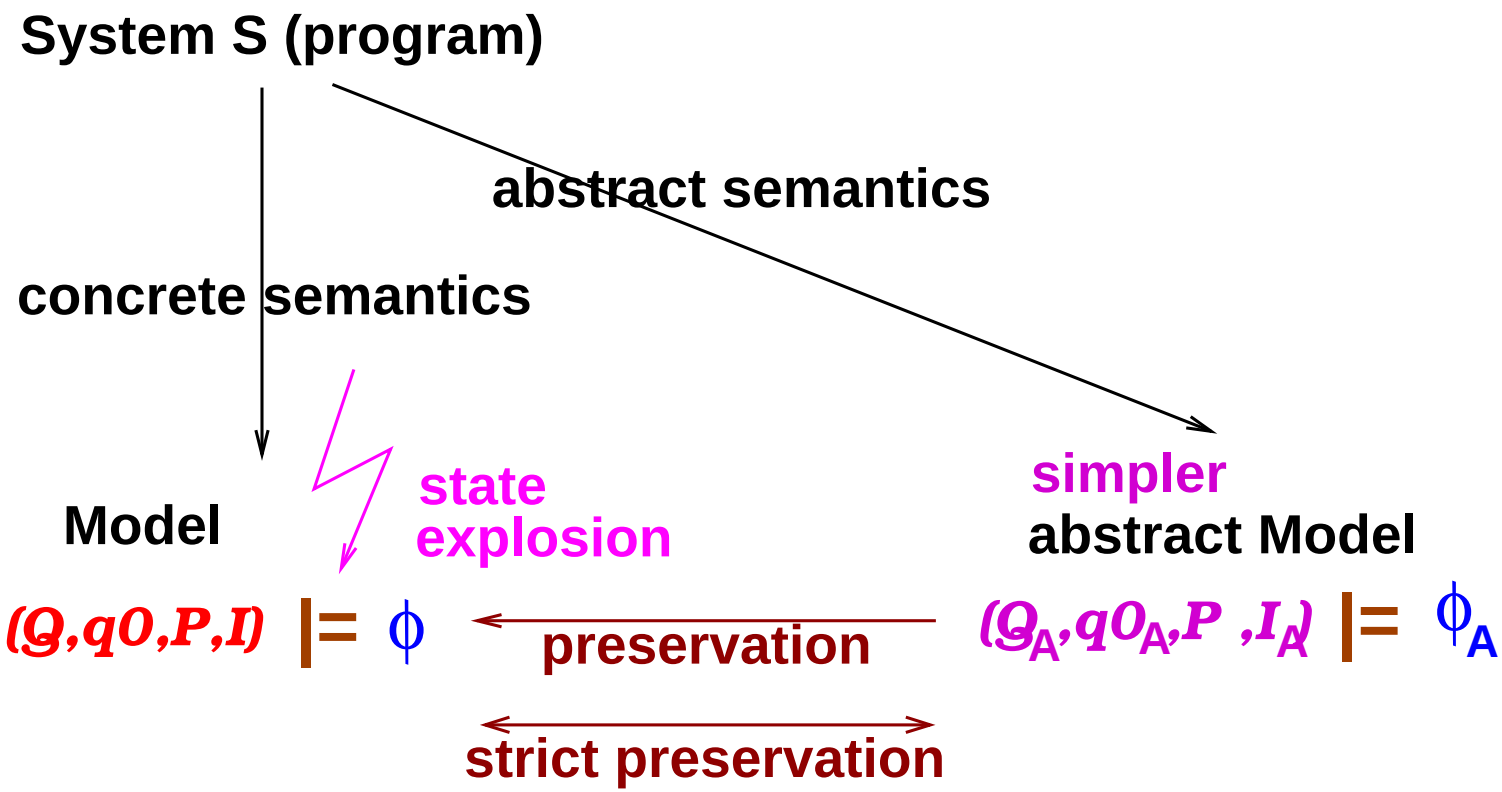
Partial order reductions, principles and applications

The principles: (Godefroid'90, Peled'92, Valmari'91,...)

1. determine independence of actions (not observable by the properties to be verified)
   - either dynamically during the state space generation by maintaining particular data structures
   - or exploiting static independence (internal actions of parallel components, ..)
2. whenever more several independent action are enabled, take them in any order (nevertheless making sure that loops contain all of them)

Fact: When using a static dependence relation, the overhead in the exploration algorithm is negligible, whereas the effect can be tremendous, especially in loosely coupled systems with many components

Abstraction: the motivation



Important: construct the abstract model directly without building the concrete one