University of Antwerp

Specification and Verification

Lecture 2: Transition systems

Guillermo A. Pérez

September 30, 2024

TL;DR: This lecture in short

What is a TS? Why study them?

A simple model of systems and how they evolve, we will focus on them as the main model on which we apply model checking.

Main references

- Christel Baier, Joost-Pieter Katoen: **Principles of Model Checking.** MIT Press 2018.
- Mickael Randour: Verification course @ UMONS.

Required and target competences

What tools do we need?

Modelling, Automata theory, and computational models

What skills will we obtain?

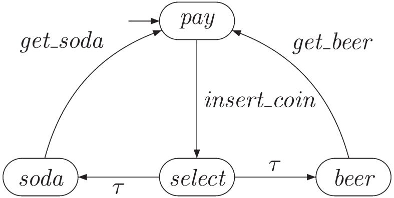
- theory: the main computational model we will be using (transition systems) and its properties
- practice: TSs as models of real systems, the state-explosion problem

How will these skills be useful?

We cannot do verification (model checking, to be precise) if we do not have a model!

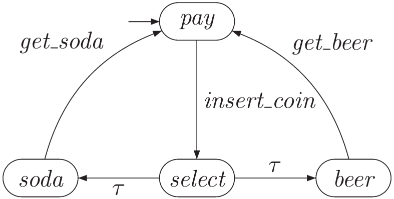
Outline

Beverage-vending transition system



- Model describing the behaviour of a system
- **State**: current mode of the system, current values of program variables, current colour of a traffic light...

Beverage-vending transition system



- Model describing the behaviour of a system
- **State**: current mode of the system, current values of program variables, current colour of a traffic light...
- **Transitions** as atomic actions: mode switching, execution of a program instruction, change of colour...

Formal definition

Definition: Transition system (TS)

Tuple $\mathcal{T} = (S, A, \longrightarrow, I, P, L)$ with

- S the set of states,
- A the set of actions,
- $\longrightarrow \subseteq S \times A \times S$ the transition relation,
- $I \subseteq S$ the set of initial states,
- P the set of atomic propositions, and
- $L\colon S \longrightarrow 2^P$ the labelling function

Formal definition

Definition: Transition system (TS)

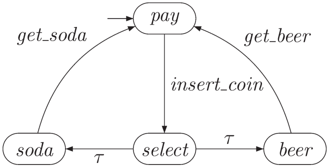
Tuple $\mathcal{T} = (S, A, \longrightarrow, I, P, L)$ with

- S the set of states,
- A the set of actions,
- $\longrightarrow \subseteq S \times A \times S$ the transition relation,
- $I \subseteq S$ the set of initial states,
- P the set of atomic propositions, and
- $L \colon S \longrightarrow 2^P$ the labelling function

Notation: sometimes we write $s \xrightarrow{a} s'$ instead of $(s, a, s') \in \longrightarrow$.

University of Antwerp

Back to the example



- $S = \{pay, select, beer, soda\}$
- $A = \{insert\_coin, get\_beer, get\_soda, \tau\}$
- Some transitions: $pay \xrightarrow{insert\_coin} select$, $select \xrightarrow{\tau} beer$
- $I = \{pay\}$

What about the propositions and the labelling?

Labelling the example TS

- Simple choice: $\forall s\,, L(s) = \{s\}$.
- Say the property is "the vending machine only delivers a drink after providing a coin"
  - $\hookrightarrow$ $P = \{paid, drink\}$, $L(pay) = \varnothing$, $L(select) = \{paid\}$ and $L(soda) = L(beer) = \{paid, drink\}$.

$\Rightarrow$ **useful to model check logic formulas**

Labelling the example TS

- Simple choice: $\forall s\,, L(s) = \{s\}$.
- Say the property is "the vending machine only delivers a drink after providing a coin"

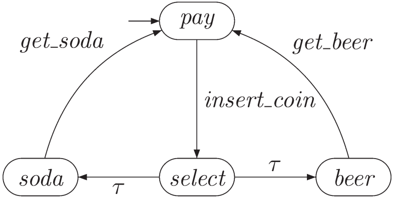
  $\hookrightarrow$ $P = \{paid, drink\}$, $L(pay) = \varnothing$, $L(select) = \{paid\}$ and $L(soda) = L(beer) = \{paid, drink\}$.

  $\Rightarrow$ **useful to model check logic formulas**

- When the labelling is not important, we often omit it
- We do the same for actions or simply use *internal actions* (ε or τ)

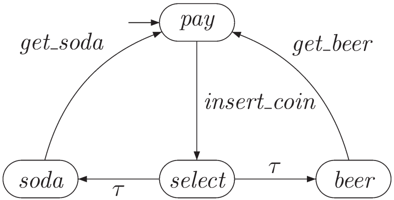
Actions are often used to model communication mechanism (e.g., parallel processes)

Semantics of TSs: non-determinism



When two actions are possible (*select*), the choice is made **non-deterministically**!

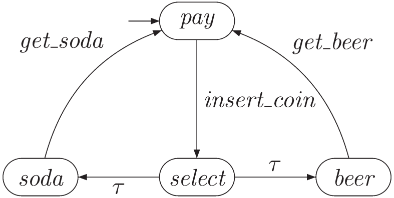
Semantics of TSs: non-determinism



When two actions are possible (*select*), the choice is made **non-deterministically**!

- Also true for the initial state if $|I| > 1$

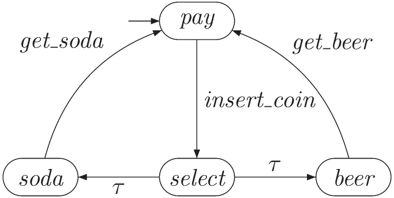
Semantics of TSs: non-determinism



When two actions are possible (*select*), the choice is made **non-deterministically**!

- Also true for the initial state if $|I| > 1$
- Meaningful to model e.g. *interleaving* of parallel executions

Semantics of TSs: non-determinism



When two actions are possible (*select*), the choice is made **non-deterministically**!

- Also true for the initial state if $|I| > 1$
- Meaningful to model e.g. *interleaving* of parallel executions
- Also for *abstraction* or to model an *uncontrollable environment* (here, drink choice by the user)

Predecessors and successors

Let $\mathcal{T}$ be a TS. For $s \in S$ and $a \in A$, we define the following sets.

University of Antwerp

Predecessors and successors

Let $\mathcal{T}$ be a TS. For $s \in S$ and $a \in A$, we define the following sets.

Direct (a-)successors of s:

$$\mathrm{Post}(s,a) = \left\{ s' \in S \;\middle|\; s \xrightarrow{a} s' \right\}, \quad \mathrm{Post}(s) = \bigcup_{a \in A} \mathrm{Post}(s,a).$$

Predecessors and successors

Let $\mathcal{T}$ be a TS. For $s \in S$ and $a \in A$, we define the following sets.

Direct (a-)successors of s:

$$\mathrm{Post}(s, a) = \left\{ s' \in S \;\middle|\; s \xrightarrow{a} s' \right\}, \quad \mathrm{Post}(s) = \bigcup_{a \in A} \mathrm{Post}(s, a).$$

Direct (a-)predecessors of s:

$$\mathrm{Pre}(s, a) = \left\{ s' \in S \;\middle|\; s' \xrightarrow{a} s \right\}, \quad \mathrm{Pre}(s) = \bigcup_{a \in A} \mathrm{Pre}(s, a).$$

Predecessors and successors

Let $\mathcal{T}$ be a TS. For $s \in S$ and $a \in A$, we define the following sets.

Direct (a-)successors of s:

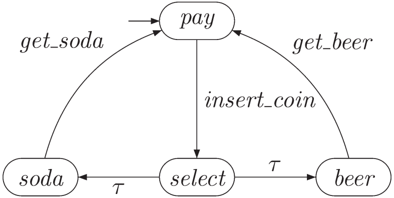
$$\mathrm{Post}(s, a) = \left\{ s' \in S \;\middle|\; s \xrightarrow{a} s' \right\}, \quad \mathrm{Post}(s) = \bigcup_{a \in A} \mathrm{Post}(s, a).$$

Direct (a-)predecessors of s:

$$\mathrm{Pre}(s, a) = \left\{ s' \in S \;\middle|\; s' \xrightarrow{a} s \right\}, \quad \mathrm{Pre}(s) = \bigcup_{a \in A} \mathrm{Pre}(s, a).$$

$+$ natural extensions to subsets of S

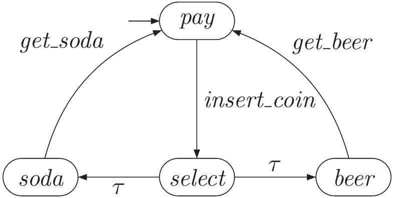
Back to the example



Some examples:

- $\mathrm{Post}(\mathit{select}) = \{\mathit{soda}, \mathit{beer}\}$,

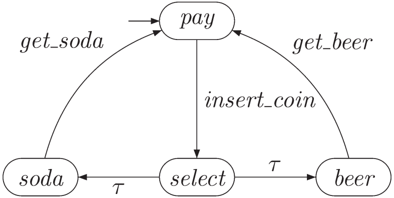
Back to the example



Some examples:

- $\mathrm{Post}(\mathit{select}) = \{\mathit{soda}, \mathit{beer}\}$,
- $\mathrm{Pre}(\mathit{pay}, \mathit{get\_beer}) = \{\mathit{beer}\}$,

Back to the example



Some examples:

- $\mathrm{Post}(\mathit{select}) = \{\mathit{soda}, \mathit{beer}\}$,
- $\mathrm{Pre}(\mathit{pay}, \mathit{get\_beer}) = \{\mathit{beer}\}$,
- $\mathrm{Post}(\mathit{beer}, \tau) = \varnothing$.

Terminal states

A state $s \in S$ is called terminal iff $\mathrm{Post}(s) = \varnothing$.

- For *reactive systems*, those states should in general be avoided
- A.k.a. sinks or trapping states

$\Rightarrow$ **deadlocks**

Executions (1/2)

Let $\mathcal{T}$ be a TS.

Executions (1/2)

Let $\mathcal{T}$ be a TS.

Finite execution fragment:

$h = s_0 a_1 s_1 a_2 \dots a_n s_n$ such that $s_0 \xrightarrow{a_1} \dots \xrightarrow{a_n} s_n$.

Executions (1/2)

Let $\mathcal{T}$ be a TS.

Finite execution fragment:

$h = s_0 a_1 s_1 a_2 \dots a_n s_n$ such that $s_0 \xrightarrow{a_1} \dots \xrightarrow{a_n} s_n$.

Infinite execution fragment:

$\rho = s_0 a_1 s_1 a_2 \dots$ such that $s_i \xrightarrow{a_{i+1}} s_{i+1}$ for all $i \geq 0$

Executions (1/2)

Let $\mathcal{T}$ be a TS.

Finite execution fragment:

$h = s_0 a_1 s_1 a_2 \ldots a_n s_n$ such that $s_0 \xrightarrow{a_1} \ldots \xrightarrow{a_n} s_n$.

Infinite execution fragment:

$\rho = s_0 a_1 s_1 a_2 \ldots$ such that $s_i \xrightarrow{a_{i+1}} s_{i+1}$ for all $i \geq 0$

Maximal execution fragment:

Fragment that cannot be prolonged

Initial execution fragment:

Fragment starting in $s_0 \in I$

Executions (2/2)

Execution:

Initial *and* maximal execution fragment

Executions (2/2)

Execution:

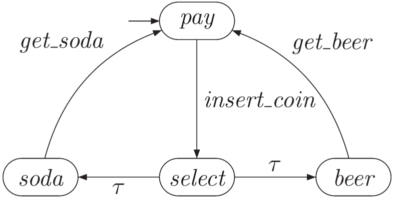
Initial *and* maximal execution fragment

Reachable states:

$$
\begin{aligned}
\mathrm{Reach}(\mathcal{T}) &= \left\{ s \in S \;\middle|\; \exists\, s_0 \in I \wedge s_0 \xrightarrow{a_1} \ldots \xrightarrow{a_n} s_n = s \right\} \\
&= \mathrm{Post}^*(I)
\end{aligned}
$$

University of Antwerp

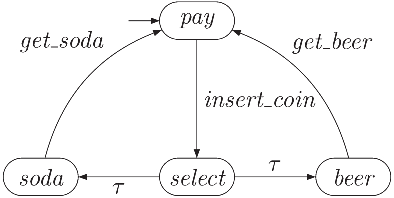
Back to the example



- $\rho_1 = pay \xrightarrow{insert\_coin} select \xrightarrow{\tau} beer \xrightarrow{get\_beer} pay \xrightarrow{insert\_coin} \ldots$

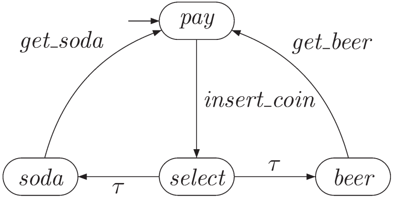
  $\hookrightarrow$ ρ_1 is an execution

Back to the example



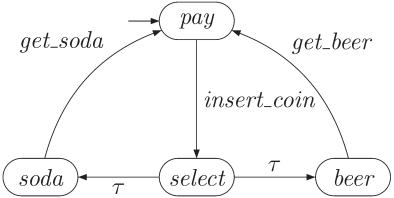
- $\rho_1 = pay \xrightarrow{insert\_coin} select \xrightarrow{\tau} beer \xrightarrow{get\_beer} pay \xrightarrow{insert\_coin} \ldots$
  $\hookrightarrow$ ρ_1 is an execution
- $\rho_2 = beer \xrightarrow{get\_beer} pay \xrightarrow{insert\_coin} select \xrightarrow{\tau} beer \xrightarrow{get\_beer} \ldots$
  $\hookrightarrow$ ρ_2 is not (maximal but not initial)

Back to the example



- $\rho_1 = pay \xrightarrow{insert\_coin} select \xrightarrow{\tau} beer \xrightarrow{get\_beer} pay \xrightarrow{insert\_coin} \ldots$
  $\hookrightarrow$ ρ_1 is an execution
- $\rho_2 = beer \xrightarrow{get\_beer} pay \xrightarrow{insert\_coin} select \xrightarrow{\tau} beer \xrightarrow{get\_beer} \ldots$
  $\hookrightarrow$ ρ_2 is not (maximal but not initial)
- $h_3 = pay \xrightarrow{insert\_coin} select \xrightarrow{\tau} soda \xrightarrow{get\_soda} pay$
  $\hookrightarrow$ h_3 is not (initial but not maximal)

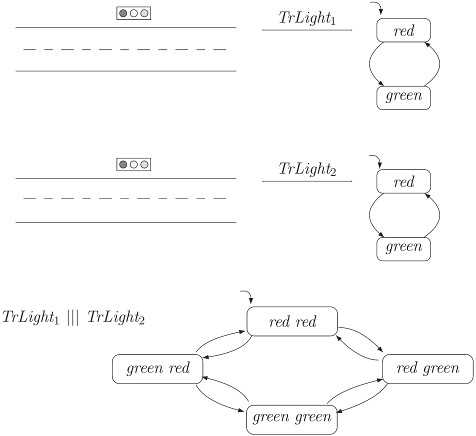
Back to the example



- $\rho_1 = \mathit{pay} \xrightarrow{\mathit{insert\_coin}} \mathit{select} \xrightarrow{\tau} \mathit{beer} \xrightarrow{\mathit{get\_beer}} \mathit{pay} \xrightarrow{\mathit{insert\_coin}} \ldots$
  $\hookrightarrow$ ρ_1 is an execution
- $\rho_2 = \mathit{beer} \xrightarrow{\mathit{get\_beer}} \mathit{pay} \xrightarrow{\mathit{insert\_coin}} \mathit{select} \xrightarrow{\tau} \mathit{beer} \xrightarrow{\mathit{get\_beer}} \ldots$
  $\hookrightarrow$ ρ_2 is not (maximal but not initial)
- $h_3 = \mathit{pay} \xrightarrow{\mathit{insert\_coin}} \mathit{select} \xrightarrow{\tau} \mathit{soda} \xrightarrow{\mathit{get\_soda}} \mathit{pay}$
  $\hookrightarrow$ h_3 is not (initial but not maximal)
- $\mathrm{Reach}(\mathcal{T}) = S$

Outline

Independent traffic lights



- Concurrency is represented by **interleaving**
- Non-deterministic choice between activities of simultaneously acting processes
- In general, needs to be complemented with **fairness** assumptions

Mutex with semaphores (1/3)

P_i **loop forever**

$\vdots$ (* noncritical actions *)

request

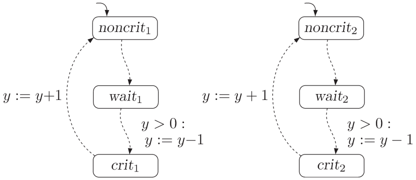
critical section

release

$\vdots$ (* noncritical actions *)

end loop

PG_1 : PG_2 :

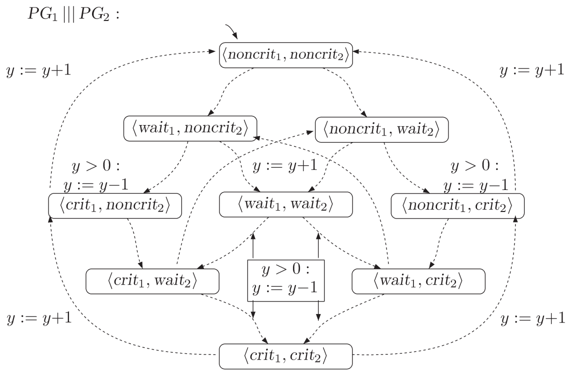


- **Program graphs (PGs)** retain conditional transitions

$\hookrightarrow$ Interleaving must be done at this level to deal with *shared variables*

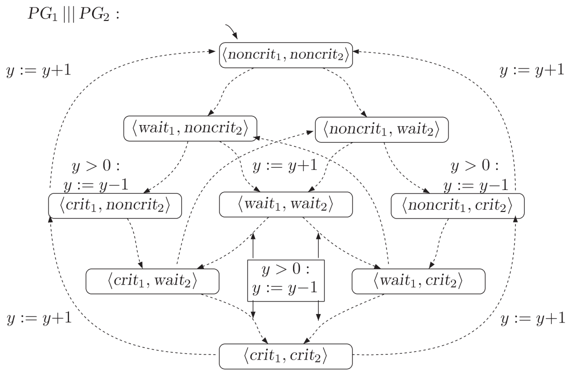
$\Rightarrow$ Then we consider the TS $\mathcal{T}(PG_1 \parallel\mid PG_2)$

Mutex with semaphores (2/3)



$PG_1 \mid\mid\mid PG_2$ for semaphore-based mutex

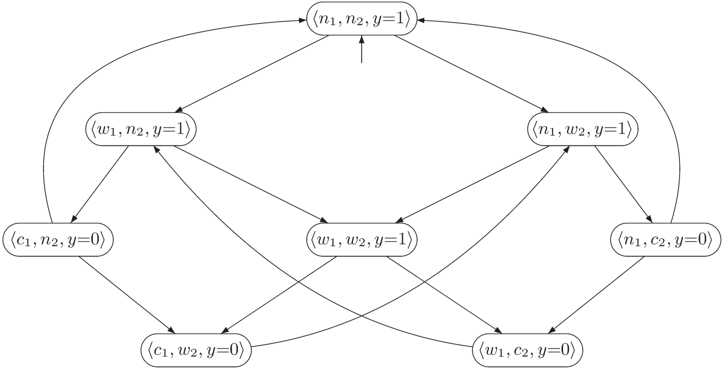
Mutex with semaphores (2/3)



$PG_1 \mid\mid\mid PG_2$ for semaphore-based mutex

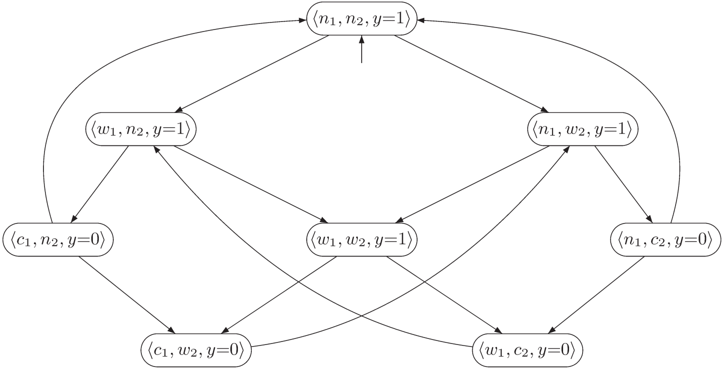
The TS unfolding will tell us if $\langle crit_1, crit_2 \rangle$ is reachable (which we want to avoid obviously)

Mutex with semaphores (3/3)



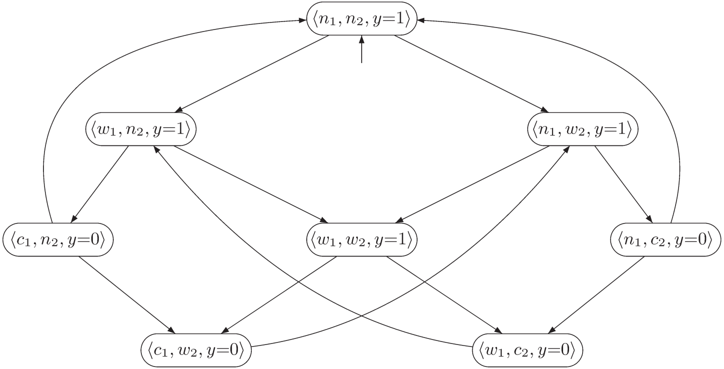
$\mathcal{T}(PG_1 \;|||\; PG_2)$ for semaphore-based mutex

Mutex with semaphores (3/3)



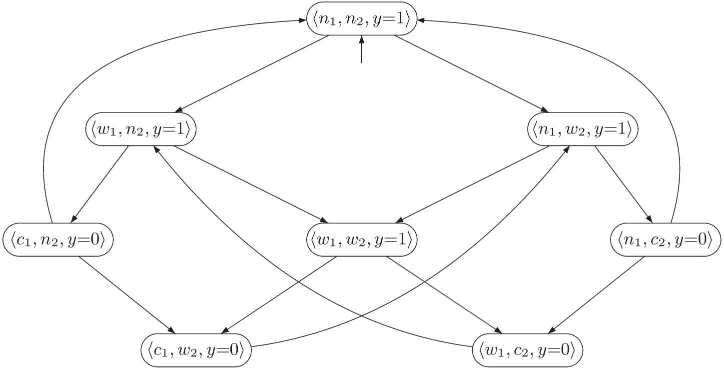
Mutual exclusion is verified: $\langle c_1, c_2, y = \ldots\rangle \notin \mathrm{Reach}(\mathcal{T}(PG_1 \mid\mid\mid PG_2))$

Mutex with semaphores (3/3)



The scheduling problem in $\langle w_1, w_2, y = 1\rangle$ is left open

Mutex with semaphores (3/3)



The scheduling problem in $\langle w_1, w_2, y = 1\rangle$ is left open

$\hookrightarrow$ implement a discipline later (LIFO, FIFO, etc) or use an algorithm solving the issue explicitly: **Peterson's mutex**

Peterson's algorithm (1/2)

P_1 **loop forever**

$\vdots$ (* noncritical actions *)

$\langle b_1 := \text{true};\ x := 2 \rangle$; (* request *)

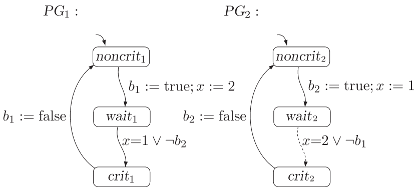
wait until $(x = 1 \ \vee \ \neg b_2)$

do critical section **od**

$b_1 := \text{false}$ (* release *)

$\vdots$ (* noncritical actions *)

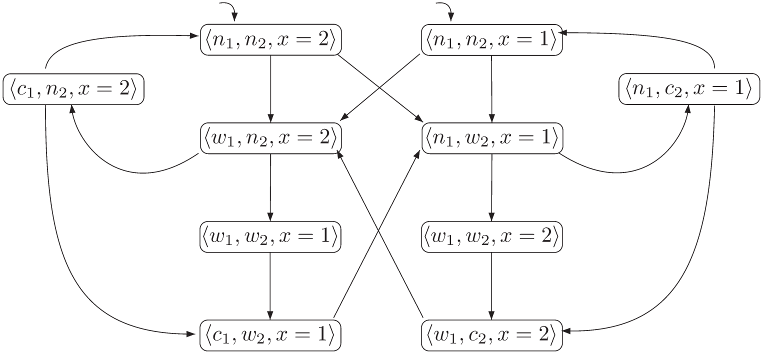
end loop



Program graphs for Peterson's mutex

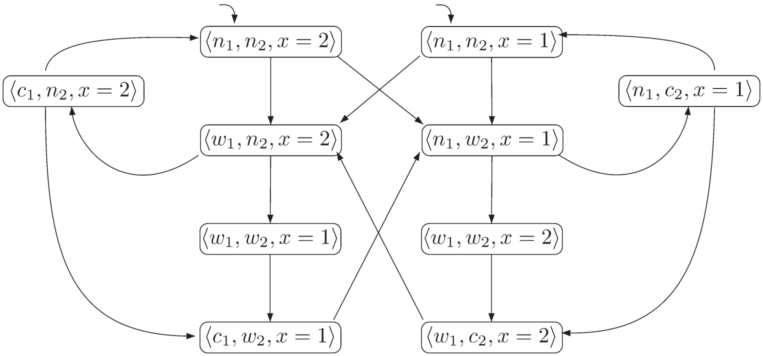
$\Rightarrow$ The value of x determines who will enter the critical section

Peterson's algorithm (2/2)



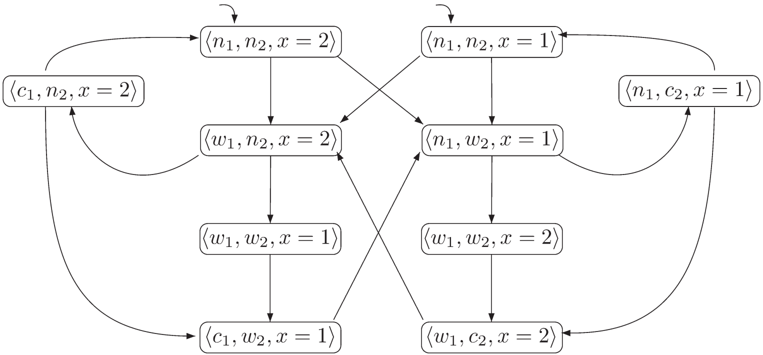
$\mathcal{T}(PG_1 \mid\mid\mid PG_2)$ for Peterson's mutex

Peterson's algorithm (2/2)



Mutual exclusion is verified: $\langle c_1, c_2, x = \ldots\rangle \notin \mathrm{Reach}(\mathcal{T}(PG_1 \mid\mid\mid PG_2))$

Peterson's algorithm (2/2)



Peterson's also has **bounded waiting**, hence **fairness** is satisfied

Not true for semaphore-based: processes could starve

The state(-space) explosion problem

Verification techniques operate on TSs obtained from programs or program graphs. Their size can be **huge**, or they can even be **infinite**. Some sources:

The state(-space) explosion problem

Verification techniques operate on TSs obtained from programs or program graphs. Their size can be **huge**, or they can even be **infinite**. Some sources:

- **Variables**
  - PG with 10 locations, three Boolean variables and five integers in $\{0, \ldots, 9\}$ already contains $10 \cdot 2^3 \cdot 10^5 = 8.000.000$ states
  - Variables in infinite domain $\Rightarrow$ infinite TS!

The state(-space) explosion problem

Verification techniques operate on TSs obtained from programs or program graphs. Their size can be **huge**, or they can even be **infinite**. Some sources:

- **Variables**
  - PG with 10 locations, three Boolean variables and five integers in $\{0, \ldots, 9\}$ already contains $10 \cdot 2^3 \cdot 10^5 = 8.000.000$ states
  - Variables in infinite domain $\Rightarrow$ infinite TS!
- **Parallelism**
  - $\mathcal{T} = \mathcal{T}_1 \mid\mid\mid \cdots \mid\mid\mid \mathcal{T}_n \;\Rightarrow\; |S| = |S_1| \cdots |S_n|$.

    $\hookrightarrow$ **Exponential blow-up!**

The state(-space) explosion problem

Verification techniques operate on TSs obtained from programs or program graphs. Their size can be **huge**, or they can even be **infinite**. Some sources:

- **Variables**
  - PG with 10 locations, three Boolean variables and five integers in $\{0, \dots, 9\}$ already contains $10 \cdot 2^3 \cdot 10^5 = 8.000.000$ states
  - Variables in infinite domain $\Rightarrow$ infinite TS!
- **Parallelism**
  - $\mathcal{T} = \mathcal{T}_1 \parallel\!\mid \cdots \parallel\!\mid \mathcal{T}_n \;\Rightarrow\; |S| = |S_1| \cdots |S_n|$.

    $\hookrightarrow$ **Exponential blow-up!**

$\Rightarrow$ Need for (a lot of) **abstraction** and efficient **symbolic** techniques

Pause

A short break?

Outline

Why?

- To see if two TSs are *similar*.
  - Is one a **refinement** or an **abstraction** of the other?
  - Are the two *indistinguishable* w.r.t. observable properties?

Why?

- To see if two TSs are *similar*.
  - Is one a **refinement** or an **abstraction** of the other?
  - Are the two *indistinguishable* w.r.t. observable properties?

- To be able to *model check large systems*
  - If $\mathcal{T}_1$ is a small abstraction of $\mathcal{T}_2$ that preserves the property to be checked, then model checking $\mathcal{T}_1$ is more efficient!

  $\hookrightarrow$ Can help for large or infinite systems: not all complexity is necessary!

Why?

- To see if two TSs are *similar*.
  - Is one a **refinement** or an **abstraction** of the other?
  - Are the two *indistinguishable* w.r.t. observable properties?

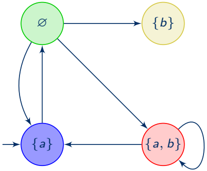
- To be able to *model check large systems*
  - If $\mathcal{T}_1$ is a small abstraction of $\mathcal{T}_2$ that preserves the property to be checked, then model checking $\mathcal{T}_1$ is more efficient!

  $\hookrightarrow$ Can help for large or infinite systems: not all complexity is necessary!

- What does it mean to *preserve a property*?
  - Each type of relation preserves a different logical fragment (intuitively, a different kind of properties)

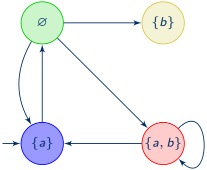
  $\hookrightarrow$ Depends on what we are interested in

Linear vs. branching time semantics (1/2)



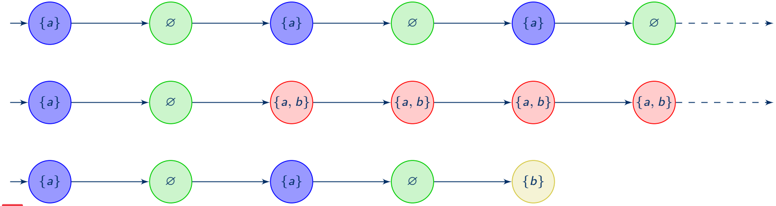
TS $\mathcal{T}$ with state labels $P = \{a, b\}$ (state and action names are omitted)

Linear vs. branching time semantics (1/2)

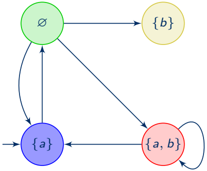


TS $\mathcal{T}$ with state labels $P = \{a, b\}$ (state and action names are omitted)

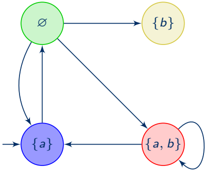
- **Linear time semantics** deal with *traces* of executions, e.g. the language of (in)finite words described by $\mathcal{T}$
- E.g., *do all executions eventually reach $\{b\}$?*

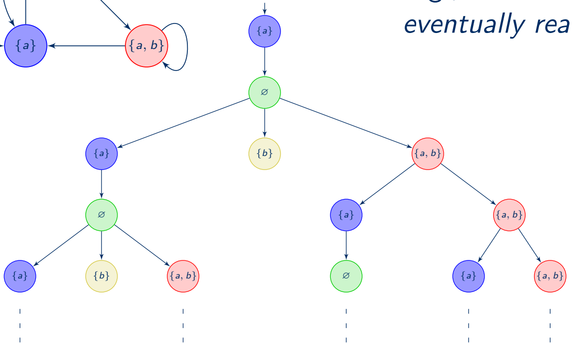


Linear vs. branching time semantics (2/2)



Linear vs. branching time semantics (2/2)





- **Branching time semantics** deals with the *execution tree*, i.e. infinite unfolding of all branching possibilities
- E.g., *do all executions always have the possibility to eventually reach* $\{b\}$*?*

How should we compare TSs?

- **Linear time properties**

  $\Rightarrow$ **Trace equivalence/inclusion** is an obvious choice

How should we compare TSs?

- **Linear time properties**

  $\Rightarrow$ **Trace equivalence/inclusion** is an obvious choice

  - But **language inclusion is costly!** (PSPACE-complete)

How should we compare TSs?

- **Linear time properties**

  $\Rightarrow$ **Trace equivalence/inclusion** is an obvious choice

  - But **language inclusion is costly!** (PSPACE-complete)

  $\hookrightarrow$ Other relations provide a *more efficient alternative* (P-complete)

How should we compare TSs?

- **Linear time properties**

  $\Rightarrow$ **Trace equivalence/inclusion** is an obvious choice

  - But **language inclusion is costly!** (PSPACE-complete)

  $\hookrightarrow$ Other relations provide a *more efficient alternative* (P-complete)

- **Branching time semantics**

  $\Rightarrow$ **Bisimulation**: related states can mutually mimic all individual transitions

  $\Rightarrow$ **Simulation**: one state can mimic all step-wise behaviour of the other, but the reverse is not necessary

How should we compare TSs?

- **Linear time properties**
  - $\Rightarrow$ **Trace equivalence/inclusion** is an obvious choice
    - But **language inclusion is costly!** (PSPACE-complete)
  - $\hookrightarrow$ Other relations provide a *more efficient alternative* (P-complete)
- **Branching time semantics**
  - $\Rightarrow$ **Bisimulation**: related states can mutually mimic all individual transitions
  - $\Rightarrow$ **Simulation**: one state can mimic all step-wise behaviour of the other, but the reverse is not necessary

In the following, we assume state-based labelling and often that there is no deadlock (self-loops otherwise)

Outline

Trace inclusion and equivalence (1/3)

What is a trace? An execution seen through its labelling

Trace inclusion and equivalence (1/3)

What is a trace? An execution seen through its labelling

Definition: paths and traces

Let $\mathcal{T}$ be a TS and $\rho = s_0 a_1 s_1 a_2 \ldots$ one of its executions:

- its *path* is $\pi = \mathrm{path}(\rho) = s_0 s_1 s_2 \ldots$,
- its *trace* is $\mathrm{trace}(\pi) = L(\pi) = L(s_0)L(s_1)L(s_2)\ldots$

We denote $\mathrm{Paths}(\mathcal{T})$ (resp. $\mathrm{Traces}(\mathcal{T})$) the set of all paths (resp. traces) in $\mathcal{T}$.

Trace inclusion and equivalence (1/3)

What is a trace? An execution seen through its labelling

Definition: paths and traces

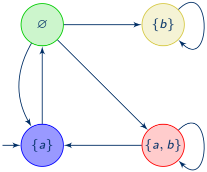
Let $\mathcal{T}$ be a TS and $\rho = s_0 a_1 s_1 a_2 \dots$ one of its executions:

- its *path* is $\pi = \mathrm{path}(\rho) = s_0 s_1 s_2 \dots$,
- its *trace* is $\mathrm{trace}(\pi) = L(\pi) = L(s_0)L(s_1)L(s_2)\dots$

We denote $\mathrm{Paths}(\mathcal{T})$ (resp. $\mathrm{Traces}(\mathcal{T})$) the set of all paths (resp. traces) in $\mathcal{T}$.

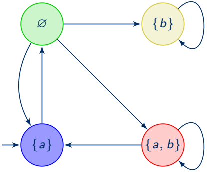
Defined for fragments starting in a state s ($\mathrm{Paths}(s)$ and $\mathrm{Traces}(s)$), a subset of states $S' \subseteq S$ ($\mathrm{Paths}(S')$ and $\mathrm{Traces}(S')$), as well as for *finite* fragments ($\mathrm{Paths}_{\mathrm{fin}}$ and $\mathrm{Traces}_{\mathrm{fin}}$).

Trace inclusion and equivalence (2/3)



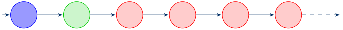
- Notice the added self-loop on $\{b\}$

Trace inclusion and equivalence (2/3)



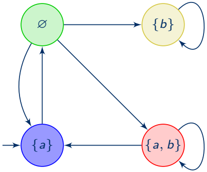
- Notice the added self-loop on $\{b\}$
- Paths:

$\pi_1 =$ 

$\pi_2 =$ 

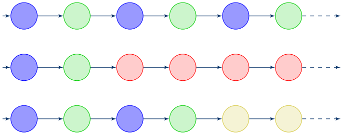
$\pi_3 =$ 

Trace inclusion and equivalence (2/3)



- Notice the added self-loop on {b}
- Paths:

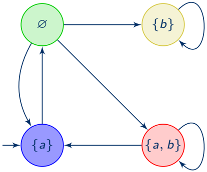
$\pi_1 =$, $\pi_2 =$, $\pi_3 =$



- Corresponding traces:

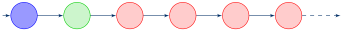
$$\mathrm{trace}(\pi_1) = \{a\}\varnothing\{a\}\varnothing\{a\}\varnothing\ldots = (\{a\}\varnothing)^\omega$$
$$\mathrm{trace}(\pi_2) = \{a\}\varnothing\{a,b\}\{a,b\}\{a,b\}\{a,b\}\ldots = \{a\}\varnothing\{a,b\}^\omega$$
$$\mathrm{trace}(\pi_3) = \{a\}\varnothing\{a\}\varnothing\{b\}\{b\}\ldots = \{a\}\varnothing\{a\}\varnothing\{b\}^\omega$$

Trace inclusion and equivalence (2/3)



- Notice the added self-loop on {b}
- Paths:

$\pi_1 =$ 

$\pi_2 =$ 

$\pi_3 =$ 

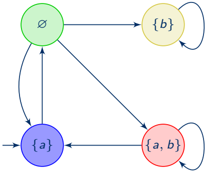
- Corresponding traces:

$$\mathrm{trace}(\pi_1) = \{a\}\varnothing\{a\}\varnothing\{a\}\varnothing \ldots = (\{a\}\varnothing)^\omega$$
$$\mathrm{trace}(\pi_2) = \{a\}\varnothing\{a,b\}\{a,b\}\{a,b\}\{a,b\} \ldots = \{a\}\varnothing\{a,b\}^\omega$$
$$\mathrm{trace}(\pi_3) = \{a\}\varnothing\{a\}\varnothing\{b\}\{b\} \ldots = \{a\}\varnothing\{a\}\varnothing\{b\}^\omega$$

Traces are (infinite) words on alphabet 2^P

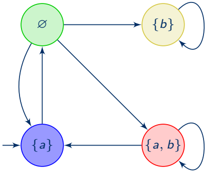
$\hookrightarrow$ **alphabet exponential in $|P|$**

Trace inclusion and equivalence (3/3)



What are the trace languages of this TS?

Trace inclusion and equivalence (3/3)

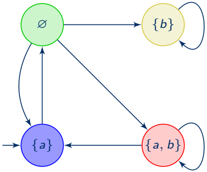


What are the trace languages of this TS?

- Finite traces:

$$\mathrm{Traces}_{\mathrm{fin}}(\mathcal{T}) = \{a\}(\varnothing\{a,b\}^*\{a\})^*\,[\varepsilon \mid \varnothing\,(\{b\}^* | \{a,b\}^*)]$$

Trace inclusion and equivalence (3/3)



What are the trace languages of this TS?

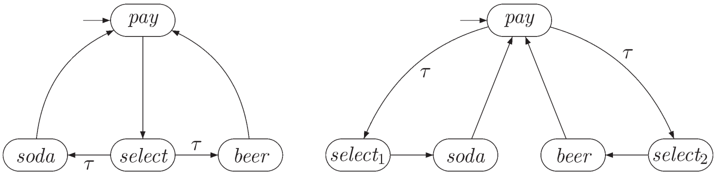
- Finite traces:

$$\mathrm{Traces}_{\mathrm{fin}}(\mathcal{T}) = \{a\}(\varnothing\{a,b\}^*\{a\})^* \left[\varepsilon \mid \varnothing\left(\{b\}^* | \{a,b\}^*\right)\right]$$

- Traces:

$$R = (\varnothing\{a,b\}^*\{a\})$$
$$\mathrm{Traces}(\mathcal{T}) = \{a\}R^* \left[R^\omega \mid (\varnothing\{a,b\}^\omega) \mid \varnothing\{b\}^\omega\right]$$

Trace inclusion and equivalence example



Trace-equivalent systems

For $P = \{pay, soda, beer\}$, these TSs are *trace-equivalent*.

$\hookrightarrow$ They are indistinguishable by LT properties

Summary and conclusions

How do we compare TSs?

It depends on what kind of behaviour we want to focus on:

- For linear-time properties we can look at the trace languages
- For branching-time properties we can look at simulation and bisimulation

Verification

- TSs are compared when we include model refinement in our design process
- There are algorithms to decide trace equivalence and inclusion
- Deciding the existence of a (bi)simulation is less complex
- We will focus on specifying desired properties via logic