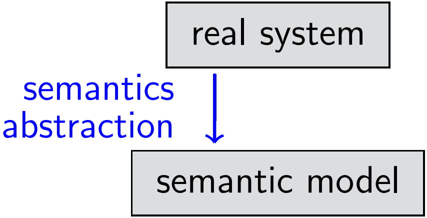
Overview

- Introduction
- **Modelling parallel systems**
    - Transition systems ⟵
    - Modeling hard- and software systems
    - Parallelism and communication
- Linear Time Properties
- Regular Properties
- Linear Temporal Logic
- Computation-Tree Logic
- Equivalences and Abstraction

Transition systems

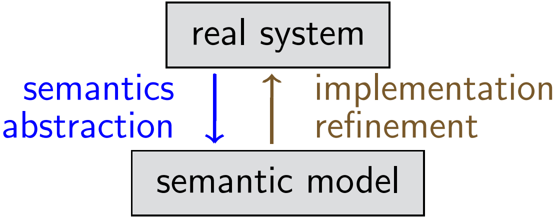


real system

semantics
abstraction

semantic model

Transition systems



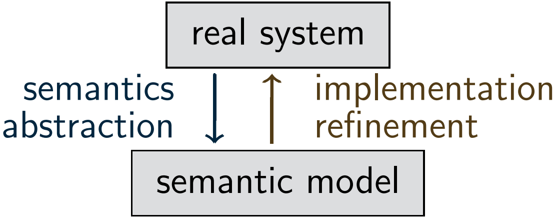
real system

semantics
abstraction

implementation
refinement

semantic model

Transition systems


real system

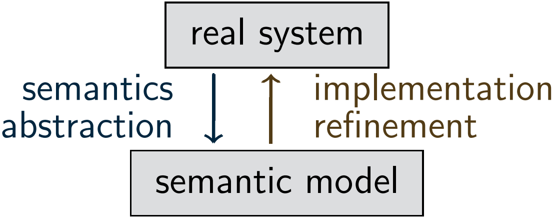
semantics
abstraction

implementation
refinement

semantic model

The semantic model yields a formal representation of:

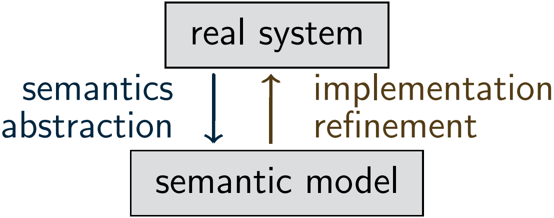
Transition systems



The semantic model yields a formal representation of:

- the states of the system
- the stepwise behaviour
- the initial states

Transition systems


real system

semantics
abstraction

implementation
refinement

semantic model

The semantic model yields a formal representation of:

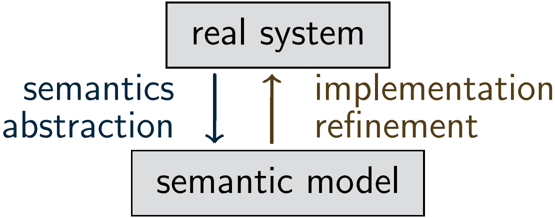
- the states of the system

↑

control component + information on "relevant" data

- the stepwise behaviour
- the initial states

Transition systems $\widehat{=}$ extended digraphs



real system

semantics
abstraction

implementation
refinement

semantic model

The semantic model yields a formal representation of:

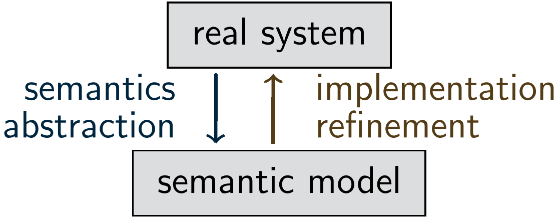
- the states of the system ⟵ **nodes**

↑

control component + information on “relevant” data

- the stepwise behaviour ⟵ **edges**
- the initial states

Transition systems $\widehat{=}$ extended digraphs



real system

semantics abstraction ↓

↑ implementation refinement

semantic model

The semantic model yields a formal representation of:

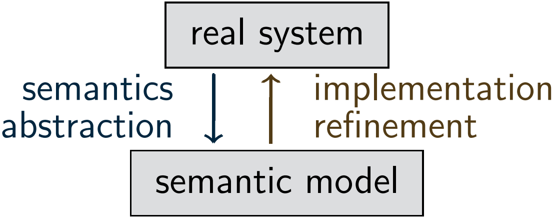
- the states of the system ⟵ **nodes**

↑

control component + information on “relevant” data

- the stepwise behaviour ⟵ **transitions**
- the initial states

Transition systems $\widehat{=}$ extended digraphs



real system

semantics
abstraction

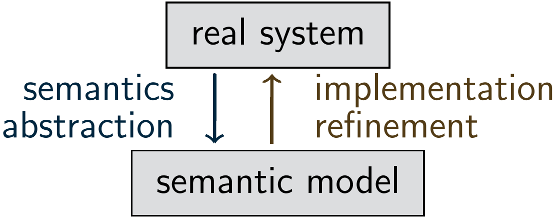
implementation
refinement

semantic model

The semantic model yields a formal representation of:

- the states of the system ⟵ **nodes**
- the stepwise behaviour ⟵ **transitions**
- the initial states
- additional information on
  communication
  state properties

Transition systems $\widehat{=}$ extended digraphs



The semantic model yields a formal representation of:

- the states of the system ⟵ **nodes**
- the stepwise behaviour ⟵ **transitions**
- the initial states
- additional information on
  communication ⟵ **actions**
  state properties ⟵ **atomic proposition**

Transition system (TS)

A transition system is a tuple

$$\mathcal{T} = (S, Act, \longrightarrow, S_0, AP, L)$$

Transition system (TS)

A transition system is a tuple

$$\mathcal{T} = (S, Act, \longrightarrow, S_0, AP, L)$$

- S is the state space, i.e., set of states,

Transition system (TS)

A transition system is a tuple

$$\mathcal{T} = (S, Act, \longrightarrow, S_0, AP, L)$$

- S is the state space, i.e., set of states,
- Act is a set of actions,

Transition system (TS)

A transition system is a tuple

$$\mathcal{T} = (S, Act, \longrightarrow, S_0, AP, L)$$

- S is the state space, i.e., set of states,
- Act is a set of actions,
- $\longrightarrow \subseteq S \times Act \times S$ is the transition relation,

Transition system (TS)

A transition system is a tuple

$$\mathcal{T} = (S, Act, \longrightarrow, S_0, AP, L)$$

- S is the state space, i.e., set of states,
- Act is a set of actions,
- $\longrightarrow \subseteq S \times Act \times S$ is the transition relation,

> i.e., transitions have the form $s \xrightarrow{\alpha} s'$
> where $s, s' \in S$ and $\alpha \in Act$

Transition system (TS)

A transition system is a tuple

$$\mathcal{T} = (S, Act, \longrightarrow, S_0, AP, L)$$

- S is the state space, i.e., set of states,
- Act is a set of actions,
- $\longrightarrow \subseteq S \times Act \times S$ is the transition relation,

  i.e., transitions have the form $s \xrightarrow{\alpha} s'$
  where $s, s' \in S$ and $\alpha \in Act$

- $S_0 \subseteq S$ the set of initial states,

Transition system (TS)

A transition system is a tuple

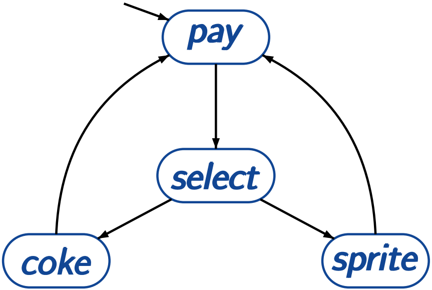
$$\mathcal{T} = (S, Act, \longrightarrow, S_0, AP, L)$$

- S is the state space, i.e., set of states,
- Act is a set of actions,
- $\longrightarrow \subseteq S \times Act \times S$ is the transition relation,

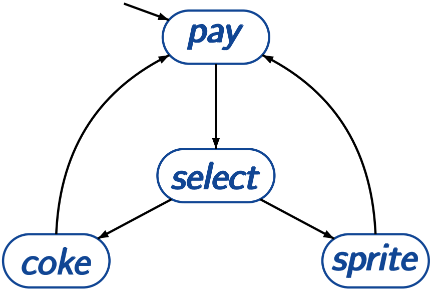
$\boxed{\text{i.e., transitions have the form } s \xrightarrow{\alpha} s' \text{ where } s, s' \in S \text{ and } \alpha \in Act}$

- $S_0 \subseteq S$ the set of initial states,
- AP a set of atomic propositions,
- $L : S \to 2^{AP}$ the labeling function

Transition system for beverage machine



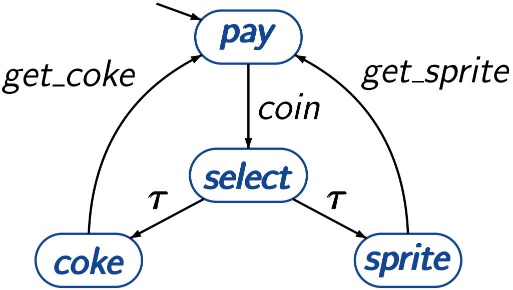
Transition system for beverage machine



state space $S = \{pay, select, coke, sprite\}$

set of initial states: $S_0 = \{pay\}$

Transition system for beverage machine



actions:

coin

τ

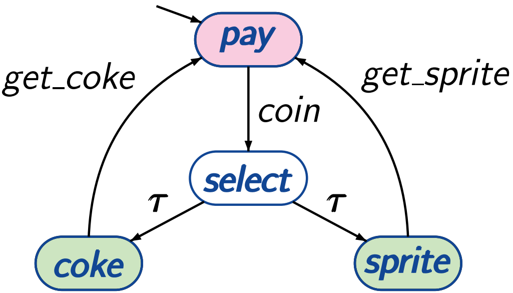
get_sprite

get_coke

state space $S = \{pay, select, coke, sprite\}$

set of initial states: $S_0 = \{pay\}$

Transition system for beverage machine



actions:
- *coin*
- τ
- *get_sprite*
- *get_coke*

state space $S = \{pay, select, coke, sprite\}$

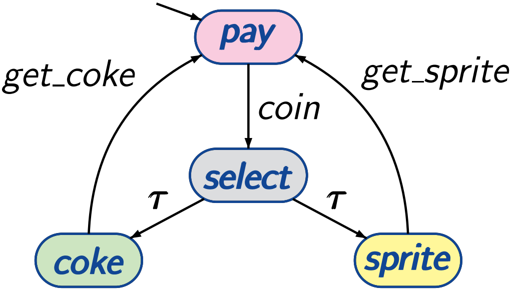
set of initial states: $S_0 = \{pay\}$

set of atomic propositions: $AP = \{pay, drink\}$

labeling function: $L(coke) = L(sprite) = \{drink\}$

$L(pay) = \{pay\}$, $L(select) = \emptyset$

Transition system for beverage machine



actions:

coin

τ

get_sprite

get_coke

state space $S = \{pay, select, coke, sprite\}$

set of initial states: $S_0 = \{pay\}$

set of atomic propositions: $AP = S$

labeling function: $L(s) = \{s\}$ for each state s

“Behaviour” of transition systems

possible behaviours of a TS result from:

```
select nondeterministically an initial state $s \in S_0$
WHILE  $s$ is non-terminal DO
        select nondeterministically a transition $s \xrightarrow{\alpha} s'$
        execute the action $\alpha$ and put $s := s'$
OD
```

“Behaviour” of transition systems

TS1.4-3

possible behaviours of a TS result from:

```
select nondeterministically an initial state $s \in S_0$
WHILE $s$ is non-terminal DO
    select nondeterministically a transition $s \xrightarrow{\alpha} s'$
    execute the action $\alpha$ and put $s := s'$
OD
```

executions: maximal “transition sequences”

$$s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \xrightarrow{\alpha_3} \ldots \text{ with } s_0 \in S_0$$

"Behaviour" of transition systems

TS1.4-3

possible behaviours of a TS result from:

```
select nondeterministically an initial state $s \in S_0$
WHILE $s$ is non-terminal DO
        select nondeterministically a transition $s \xrightarrow{\alpha} s'$
        execute the action $\alpha$ and put $s := s'$
OD
```

executions: maximal "transition sequences"

$$s_0 \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \xrightarrow{\alpha_3} \ldots \text{ with } s_0 \in S_0$$

reachable fragment:

$\mathit{Reach}(\mathcal{T})$ = set of all states that are reachable from an initial state through some execution

Possible meanings of nondeterminism in TS

Possible meanings of nondeterminism in TS

- (true) concurrency modeled by interleaving
- competition of parallel dependent actions
- implementational freedom, underspecification
- incomplete information on system environment

Transition system for parallel actions

parallel execution of independent actions

parallel execution of dependent actions

Transition system for parallel actions

parallel execution of independent actions

e.g. $\underbrace{x := x+1}_{\text{action } \alpha} \;|||\; \underbrace{y := y-3}_{\text{action } \beta}$ $\quad \alpha, \beta$ independent

parallel execution of dependent actions

Transition system for parallel actions

parallel execution of independent actions

e.g. $\underbrace{x := x+1}_{\text{action } \alpha} \;|||\; \underbrace{y := y-3}_{\text{action } \beta}$ α, β independent

parallel execution of dependent actions

e.g. $\underbrace{x := x+1}_{\text{action } \alpha} \;|||\; \underbrace{y := 2*x}_{\text{action } \beta}$ α, β dependent

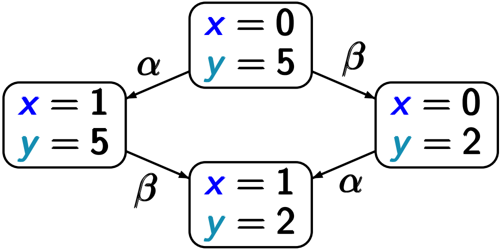
Transition system for parallel actions

parallel execution of independent actions ← $\boxed{\text{interleaving}}$

e.g. $\underbrace{x := x+1}_{\text{action } \alpha} \;|||\; \underbrace{y := y-3}_{\text{action } \beta}$ α, β independent

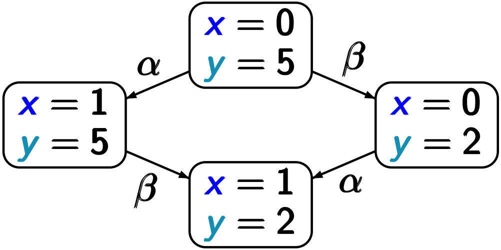
parallel execution of dependent actions ← $\boxed{\text{competition}}$

e.g. $\underbrace{x := x+1}_{\text{action } \alpha} \;|||\; \underbrace{y := 2*x}_{\text{action } \beta}$ α, β dependent parallel execution of independent actions ← $\boxed{\text{interleaving}}$



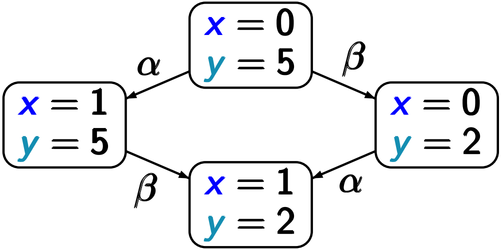
$$\underbrace{x := x+1}_{\text{action } \alpha} \;|||\; \underbrace{y := y-3}_{\text{action } \beta}$$

parallel execution of independent actions ← interleaving



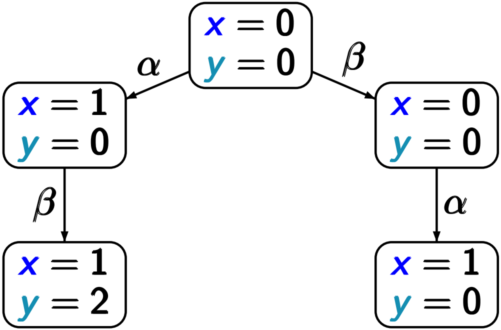
$$\underbrace{x := x{+}1}_{\text{action } \alpha} \;|||\; \underbrace{y := y{-}3}_{\text{action } \beta}$$

parallel execution of dependent actions ← competition parallel execution of independent actions ← interleaving



$$\underbrace{x := x+1}_{\text{action } \alpha} \; ||| \; \underbrace{y := y-3}_{\text{action } \beta}$$

parallel execution of dependent actions ← competition



$$\underbrace{x := x+1}_{\text{action } \alpha} \; ||| \; \underbrace{y := 2{*}x}_{\text{action } \beta}$$

Possible meanings of nondeterminism in TS

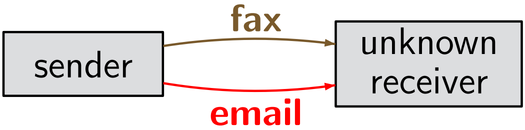
- (true) concurrency modeled by interleaving
- competition of parallel dependent actions
- implementational freedom, underspecification
- incomplete information on system environment

Implementation freedom

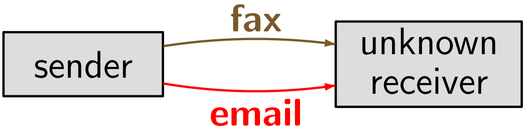
TS1.4-5

... modelled by nondeterminism

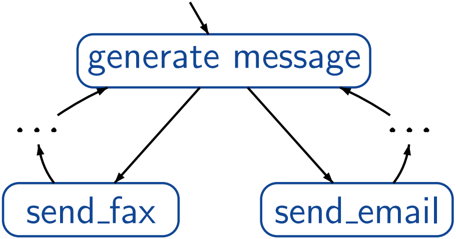
Implementation freedom



Implementation freedom

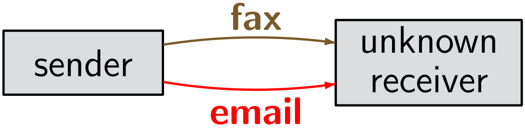


realization by a TS:

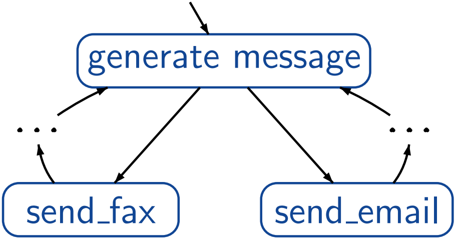


Implementation freedom

TS1.4-5

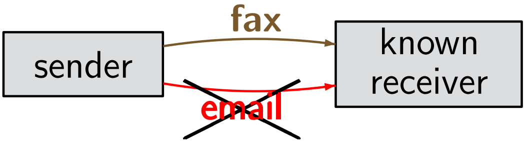


realization by a TS:



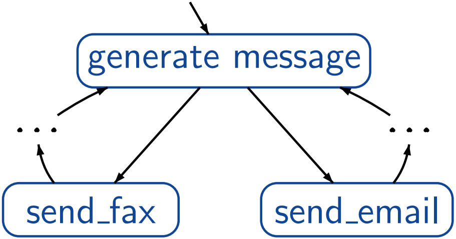
at a future refinement step the nondeterminism is replaced with one of the alternatives

Implementation freedom



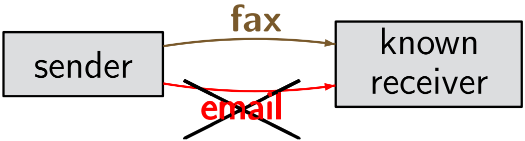
without email access

realization by a TS:



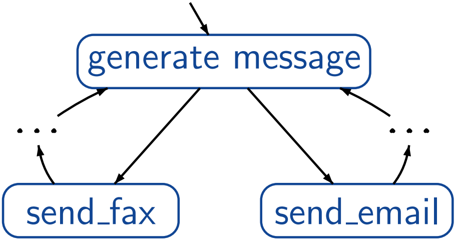
at a future refinement step the nondeterminism is replaced with one of the alternatives

Implementation freedom

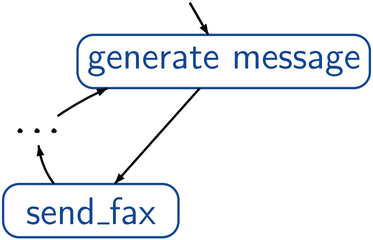


without
email access

realization by a TS:



refined TS:

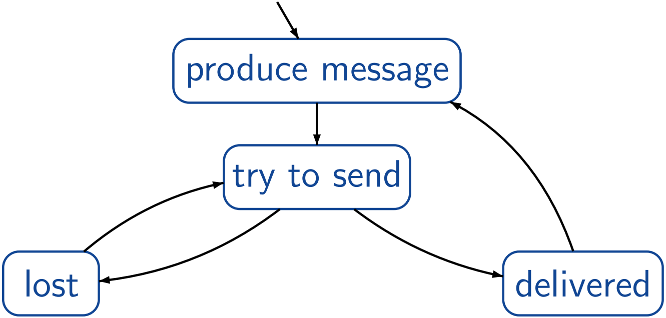


at a future refinement step the nondeterminism
is replaced with one of the alternatives

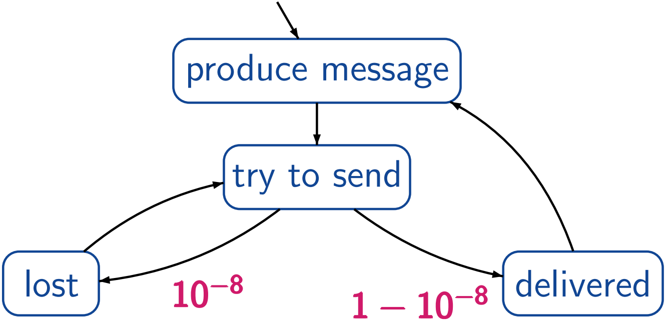
Underspecification

TS1.4-6

Underspecification



Underspecification



at a future refinement step the nondeterminism is replaced with probabilism

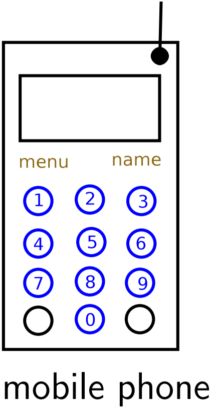
Possible meanings of nondeterminism in TS

- (true) concurrency modeled by interleaving
- competition of parallel dependent actions
- implementational freedom, underspecification
- incomplete information on system environment

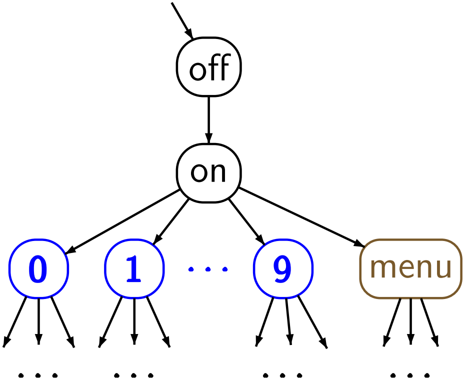
Possible meanings of nondeterminism in TS

- (true) concurrency modeled by interleaving
- competition of parallel dependent actions
- implementational freedom, underspecification
- incomplete information on system environment, e.g., interfaces with other programs, human users, sensors

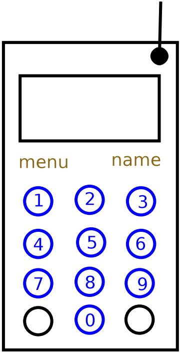
Incomplete information on the environment



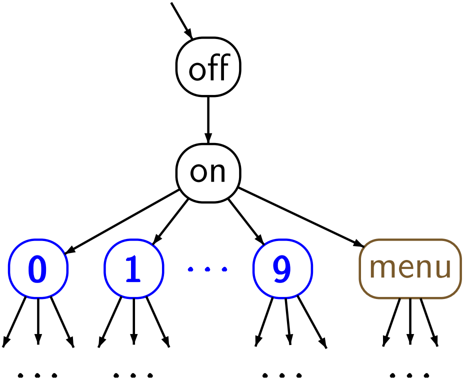
mobile phone



Incomplete information on the environment



mobile phone

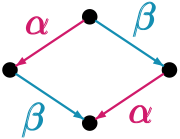


resolution of the nondeterministic choices
by a human user

Possible meanings of nondeterminism in TS

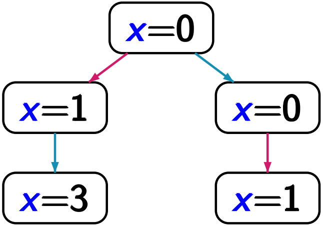
concurrency (interleaving)

$\alpha \,|||\, \beta$ is represented by



competitions

to be resolved by a scheduler
e.g. $x{:=}x{+}1 \parallel x{:=}3x$



underspecification, implementational freedom

incomplete information on system environment, e.g., interfaces with other programs, human users, sensors

Overview

Introduction

Modelling parallel systems

- Transition systems
- Modeling hard- and software systems ⟵
- Parallelism and communication

Linear Time Properties

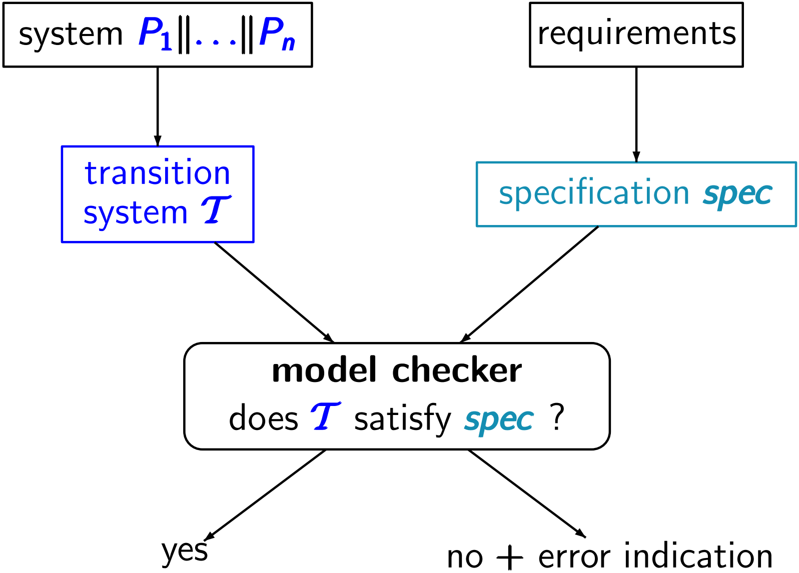
Regular Properties

Linear Temporal Logic

Computation-Tree Logic

Equivalences and Abstraction

Model checking



system $P_1 \| \ldots \| P_n$

requirements

transition system $\mathcal{T}$

specification *spec*

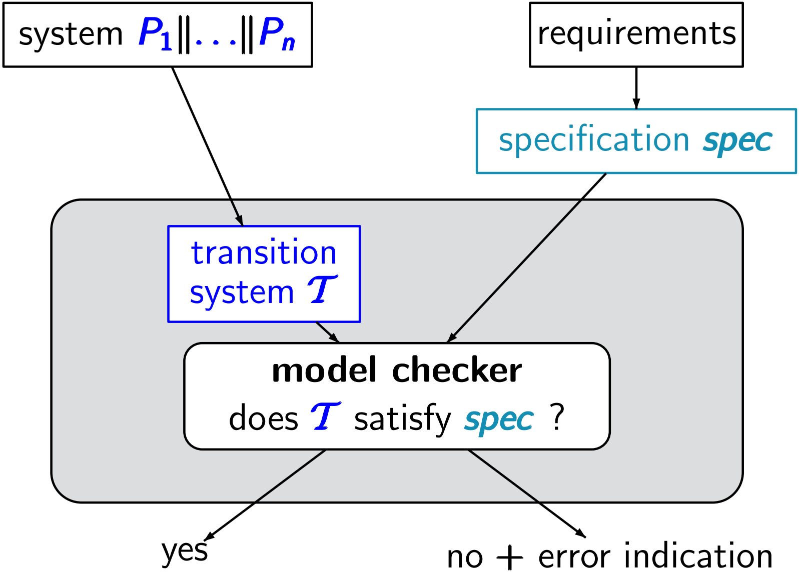
model checker

does $\mathcal{T}$ satisfy *spec* ?

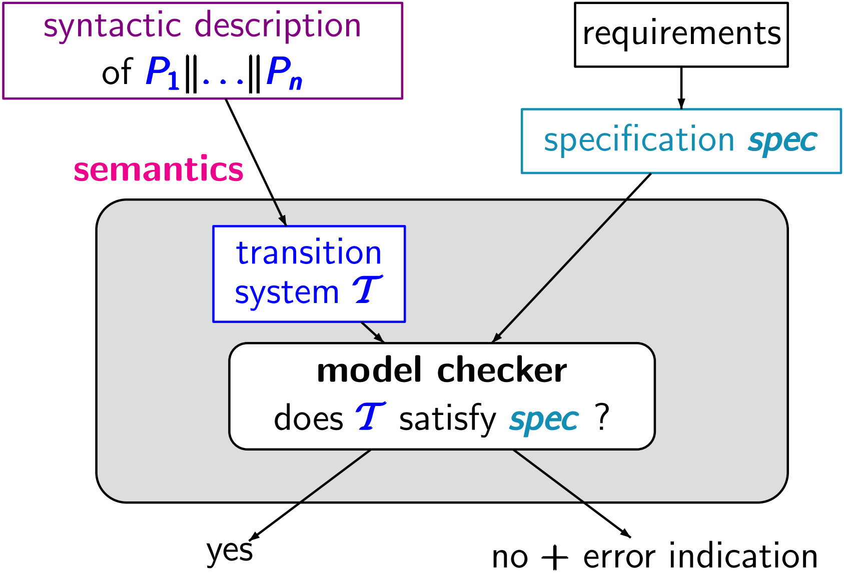
yes

no + error indication

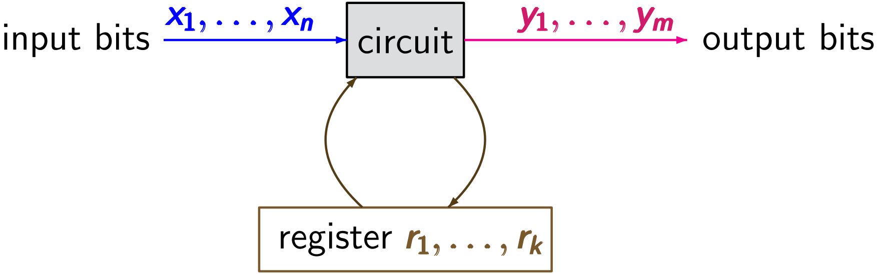
Model checking


system $P_1 \| \ldots \| P_n$
requirements
specification *spec*
transition system $\mathcal{T}$
model checker
does $\mathcal{T}$ satisfy *spec* ?
yes
no + error indication

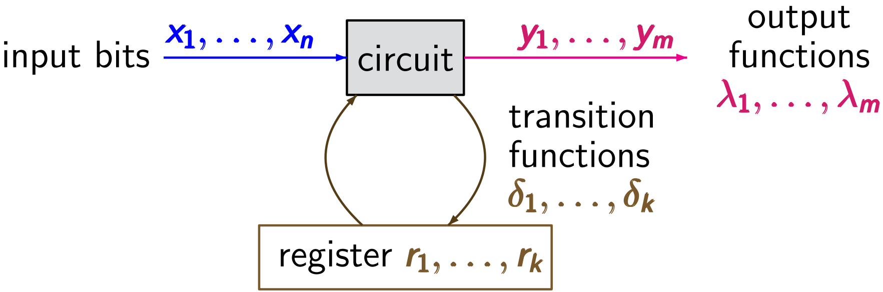
Model checking



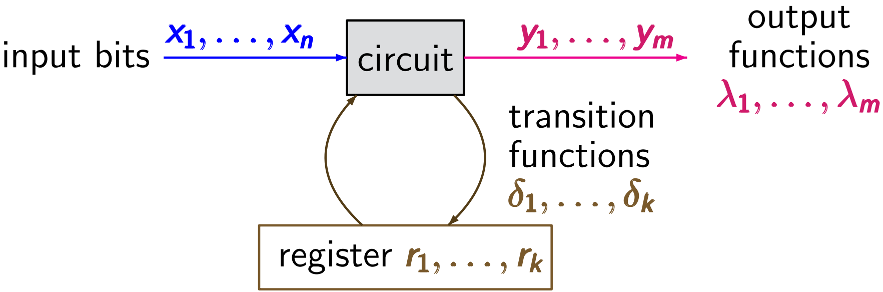
Modelling of sequential circuits by TS



Modelling of sequential circuits by TS

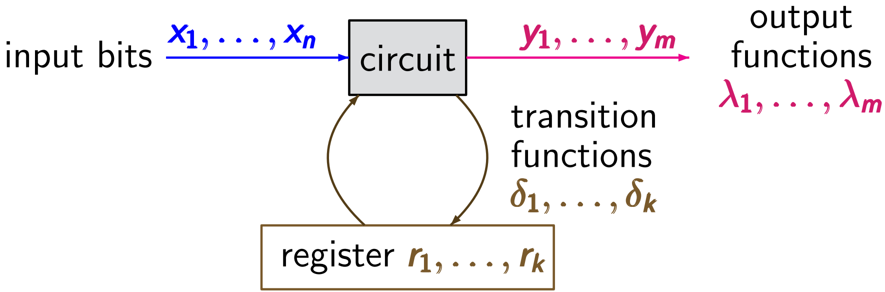


Modelling of sequential circuits by TS



$\delta_j, \lambda_i \mathrel{\widehat{=}}$ switching functions $\{0,1\}^n \times \{0,1\}^k \longrightarrow \{0,1\}$

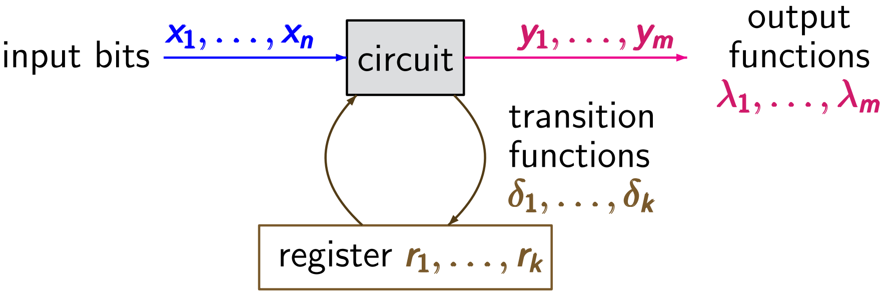
Modelling of sequential circuits by TS



$\delta_j, \lambda_i \mathrel{\widehat{=}}$ switching functions $\{0,1\}^n \times \{0,1\}^k \longrightarrow \{0,1\}$

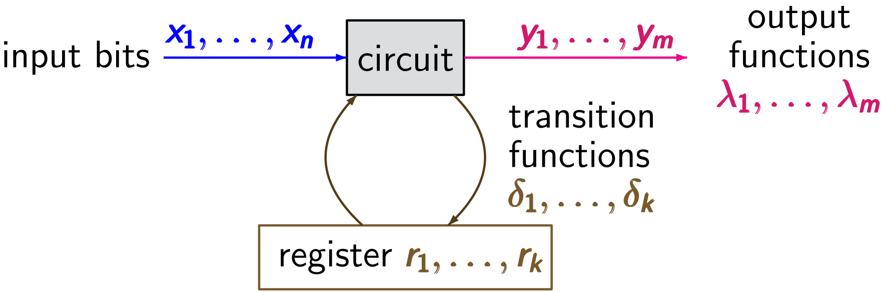
input values $a_1, \ldots, a_n$
for the input variables
+ current values $c_1, \ldots, c_k$
of the registers
	$\mapsto$	output value $\lambda_i(\ldots)$
		for output variable y_i
		next value $\delta_j(\ldots)$
		for register r_j

Modelling of sequential circuits by TS



initial register evaluation $[r_1 = c_{01}, \ldots, r_k = c_{0k}]$

Modelling of sequential circuits by TS

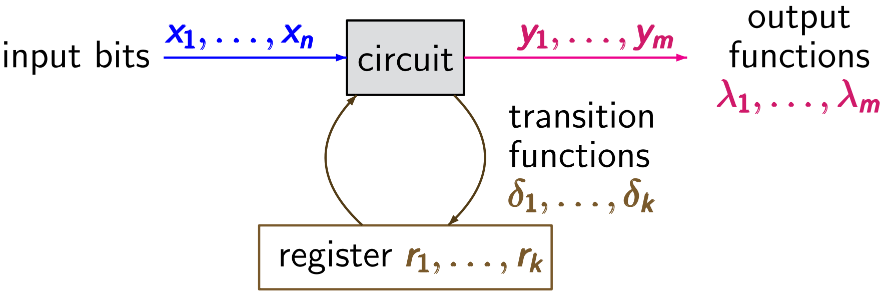


initial register evaluation $[r_1{=}c_{01}, \ldots, r_k{=}c_{0k}]$

transition system:

- states: evaluations of $x_1, \ldots, x_n, r_1, \ldots, r_k$

Modelling of sequential circuits by TS



input bits $x_1, \ldots, x_n$

circuit

$y_1, \ldots, y_m$ output functions $\lambda_1, \ldots, \lambda_m$

transition functions $\delta_1, \ldots, \delta_k$

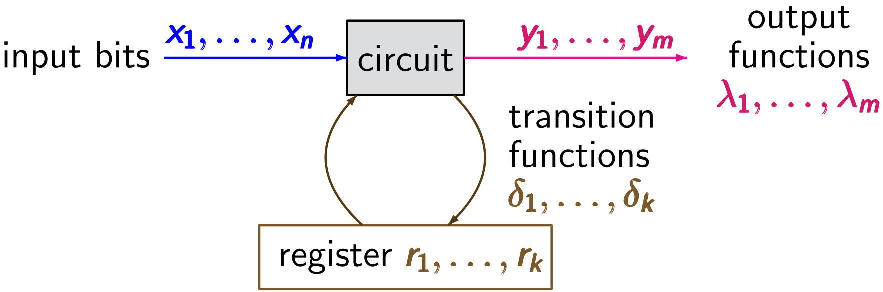
register $r_1, \ldots, r_k$

initial register evaluation $[r_1{=}c_{01}, \ldots, r_k{=}c_{0k}]$

transition system:

- states: evaluations of $x_1, \ldots, x_n, r_1, \ldots, r_k$
- transitions represent the stepwise behavior

Modelling of sequential circuits by TS

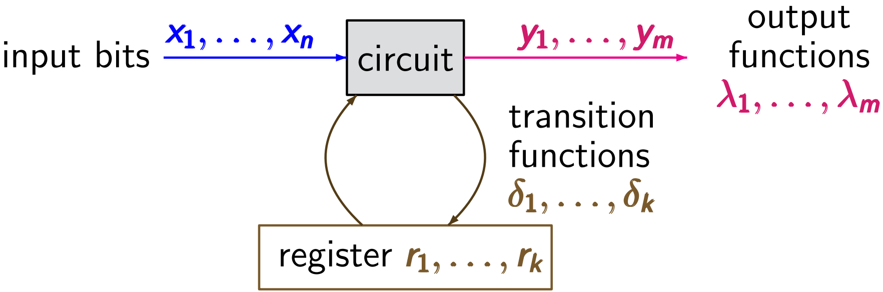


initial register evaluation $[r_1{=}c_{01}, \ldots, r_k{=}c_{0k}]$

transition system:

- states: evaluations of $x_1, \ldots, x_n, r_1, \ldots, r_k$
- transitions represent the stepwise behavior
- values of input bits change nondeterministically

Modelling of sequential circuits by TS

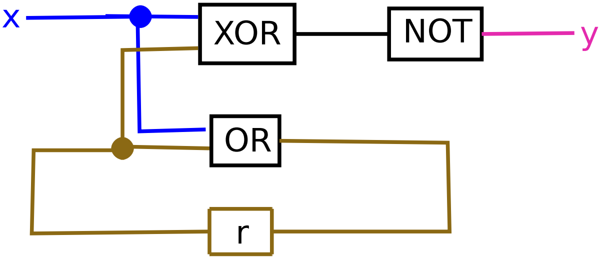


initial register evaluation $[r_1{=}c_{01}, \ldots, r_k{=}c_{0k}]$

transition system:

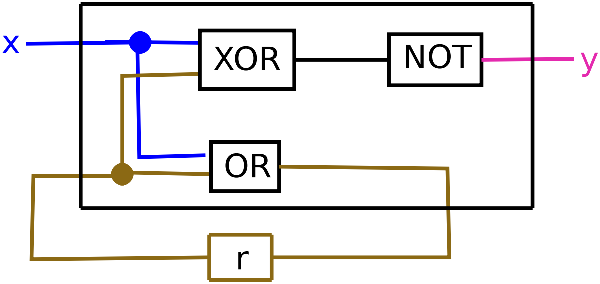
- states: evaluations of $x_1, \ldots, x_n, r_1, \ldots, r_k$
- transitions represent the stepwise behavior
- values of input bits change nondeterministically
- atomic propositions: $x_1, \ldots, x_n, y_1, \ldots, y_m, r_1, \ldots, r_k$

Example: sequential circuit



Example: sequential circuit

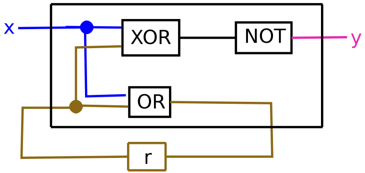
TS1.4-11A



output function: $\lambda_y = \neg(x \oplus r)$

transition function: $\delta_r = x \vee r$

Example: TS for sequential circuit



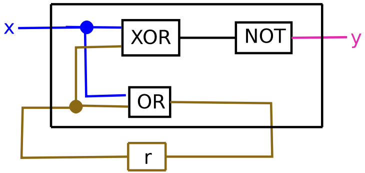
output function

$$\lambda_y = \neg(x \oplus r)$$

transition function

$$\delta_r = x \vee r$$

Example: TS for sequential circuit



output function

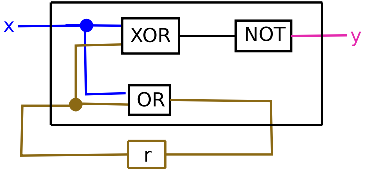
$\lambda_y = \neg(x \oplus r)$

transition function

$\delta_r = x \vee r$

transition system

Example: TS for sequential circuit



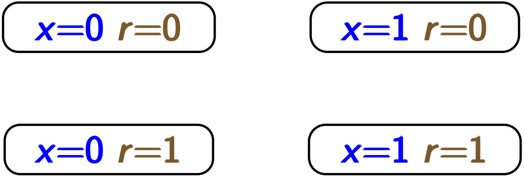
output function

$\lambda_y = \neg(x \oplus r)$

transition function

$\delta_r = x \vee r$

transition system



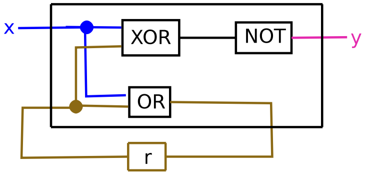
$x{=}0 \;\; r{=}0$

$x{=}1 \;\; r{=}0$

$x{=}0 \;\; r{=}1$

$x{=}1 \;\; r{=}1$

Example: TS for sequential circuit



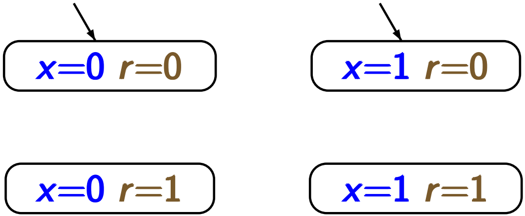
output function

$\lambda_y = \neg(x \oplus r)$

transition function

$\delta_r = x \vee r$

transition system


$x{=}0\ r{=}0$

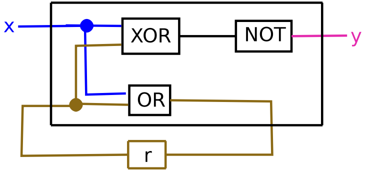
$x{=}1\ r{=}0$

$x{=}0\ r{=}1$

$x{=}1\ r{=}1$

initial register evaluation: $r{=}0$

Example: TS for sequential circuit



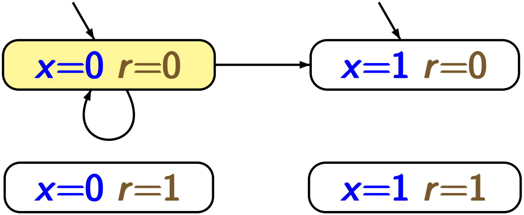
output function

$\lambda_y = \neg(x \oplus r)$

transition function

$\delta_r = x \vee r$

transition system


$x{=}0\ r{=}0$

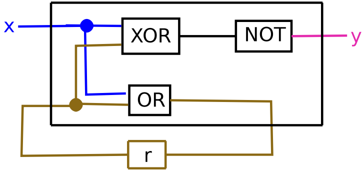
$x{=}1\ r{=}0$

$x{=}0\ r{=}1$

$x{=}1\ r{=}1$

initial register evaluation: $r{=}0$

Example: TS for sequential circuit



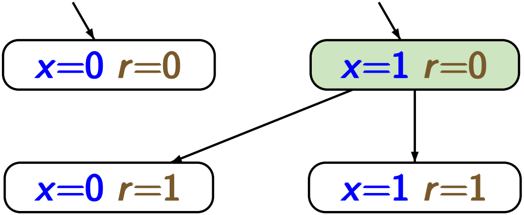
output function

$\lambda_y = \neg(x \oplus r)$

transition function

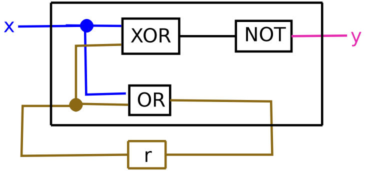
$\delta_r = x \vee r$

transition system



initial register evaluation: r=0

Example: TS for sequential circuit



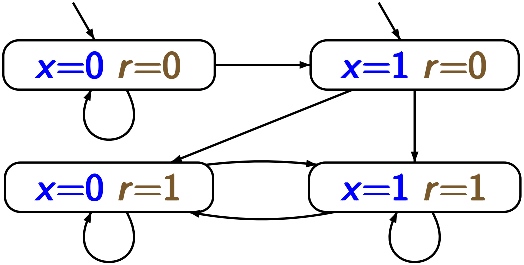
output function

$\lambda_y = \neg(x \oplus r)$

transition function

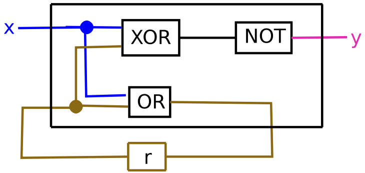
$\delta_r = x \vee r$

transition system



initial register evaluation: r=0

Example: TS for sequential circuit



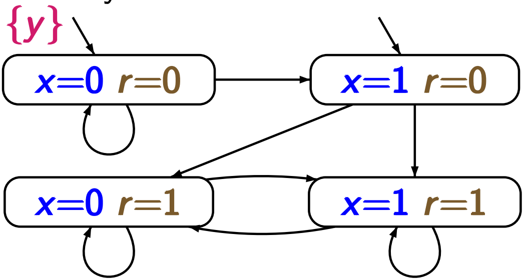
output function

$\lambda_y = \neg(x \oplus r)$

transition function

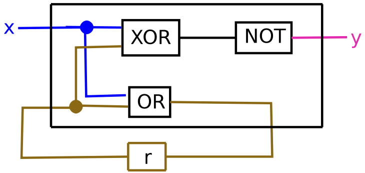
$\delta_r = x \vee r$

transition system



initial register evaluation: $r{=}0$

Example: TS for sequential circuit



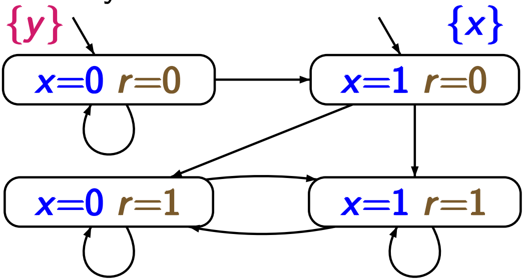
output function

$\lambda_y = \neg(x \oplus r)$

transition function

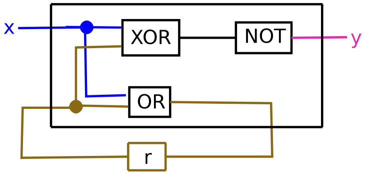
$\delta_r = x \vee r$

transition system



initial register evaluation: $r{=}0$

Example: TS for sequential circuit



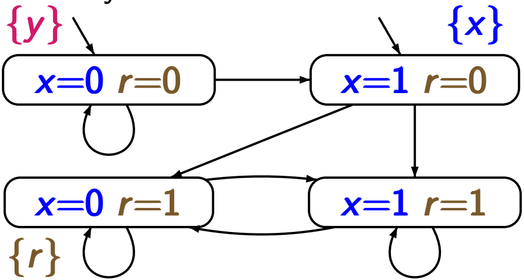
output function

$\lambda_y = \neg(x \oplus r)$

transition function

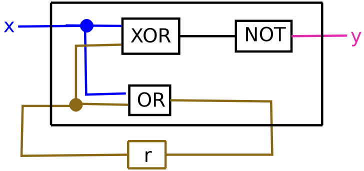
$\delta_r = x \vee r$

transition system



initial register evaluation: $r{=}0$

Example: TS for sequential circuit



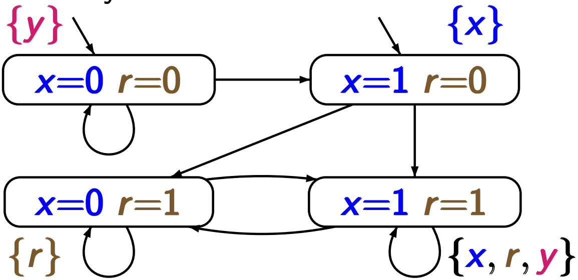
output function

$\lambda_y = \neg(x \oplus r)$

transition function

$\delta_r = x \vee r$

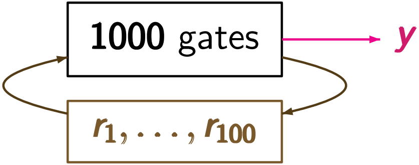
transition system



initial register evaluation: r=0

How many states . . .

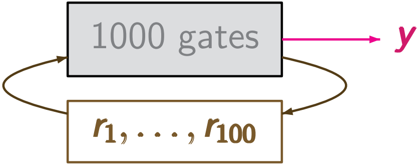
. . . has the transition system for a circuit of the form?



1 output bit
no input
100 registers

How many states . . .

. . . has the transition system for a circuit of the form?

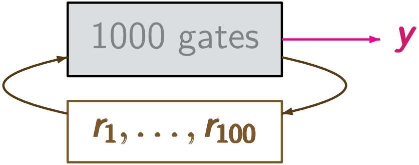


1 output bit
no input
100 registers

answer: 2^{100}

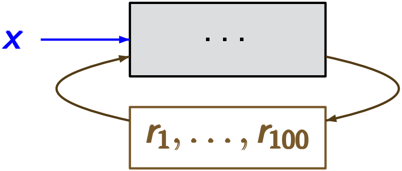
How many states . . .

. . . has the transition system for a circuit of the form?



1 output bit
no input
100 registers

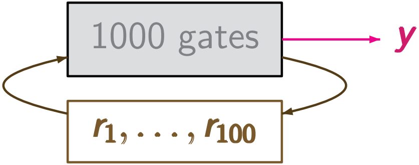
answer: 2^{100}



no output
1 input bit
100 registers

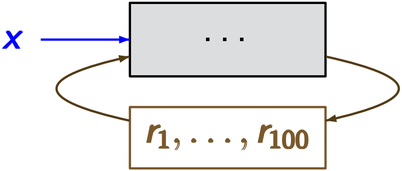
How many states . . .

. . . has the transition system for a circuit of the form?



1 output bit
no input
100 registers

answer: 2^{100}

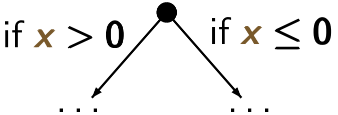


no output
1 input bit
100 registers

answer: $2^{100} * 2^1 = 2^{101}$

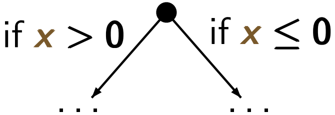
Data-dependent systems

problem: TS-representation of conditional branchings **?**



Data-dependent systems

problem: TS-representation of conditional branchings **?**

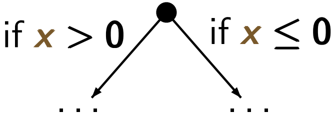


example: sequential program

```
WHILE  $x > 0$ DO
      $x := x - 1$;
      $y := y + 1$
OD
...
```

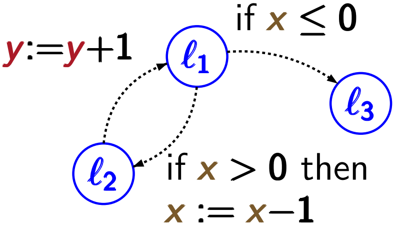
Data-dependent systems

problem: TS-representation of conditional branchings **?**



example: sequential program

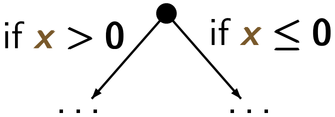
```
WHILE  $x > 0$ DO
       $x := x-1$;
       $y := y+1$
OD
...
```



Data-dependent systems

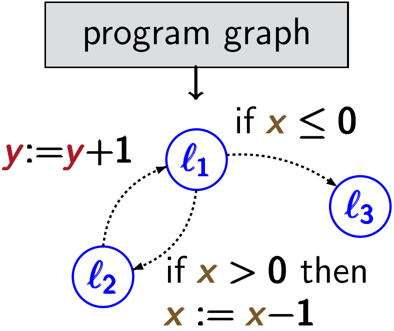
TS1.4-13

problem: TS-representation of conditional branchings **?**



example: sequential program

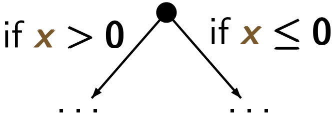
```
WHILE  $x > 0$ DO
        $x := x-1$;
        $y := y+1$
OD
...
```



Data-dependent systems

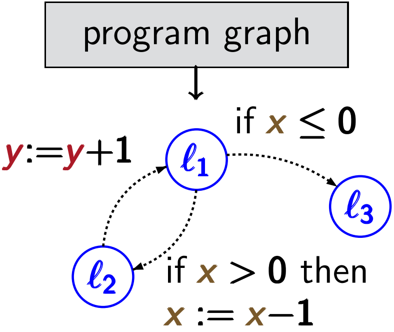
TS1.4-13

problem: TS-representation of conditional branchings **?**



example: sequential program

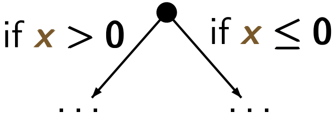
```
$\ell_1$→   WHILE  $x > 0$ DO
                $x := x-1$;
$\ell_2$→       $y := y+1$
            OD
$\ell_3$→   ...
```



ℓ_1, ℓ_2, ℓ_3 are locations,
i.e., control states

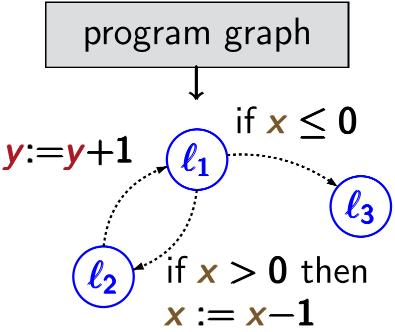
Data-dependent systems

problem: TS-representation of conditional branchings **?**



example: sequential program

```
ℓ1→  WHILE x > 0 DO
             x := x−1;
ℓ2→          y := y+1
     OD
ℓ3→  ...
```



states of the transition system:

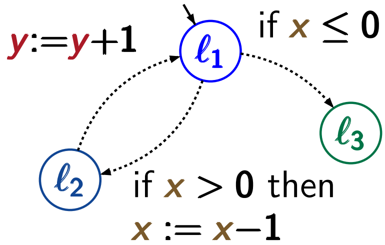
locations + relevant data (*here:* values for x and y)

Example: TS for sequential program

initially: $x = 2$, $y = 0$

```
ℓ1→  WHILE  $x > 0$ DO
               $x := x-1$
ℓ2→            $y := y+1$
     OD
ℓ3→  ...
```

program graph

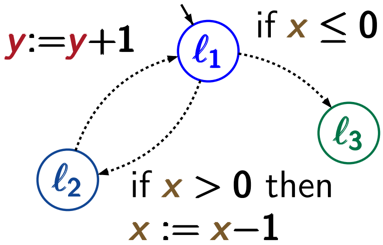


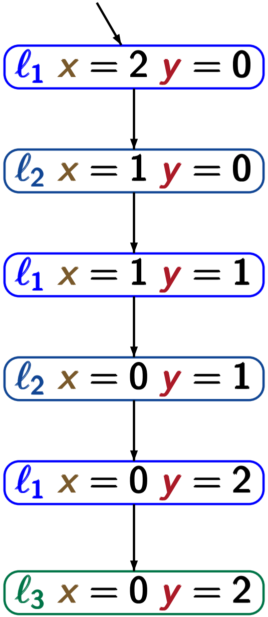
Example: TS for sequential program

initially: $x = 2$, $y = 0$

```
$\ell_1 \rightarrow$  WHILE $x > 0$ DO
                 $x := x - 1$
$\ell_2 \rightarrow$                 $y := y + 1$
          OD
$\ell_3 \rightarrow$  $\ldots$
```

program graph



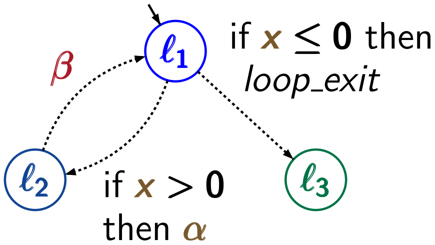


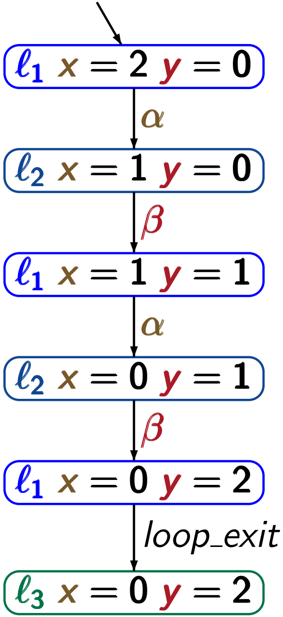
Example: TS for sequential program

initially: $x = 2$, $y = 0$

```
$\ell_1 \rightarrow$   WHILE $x > 0$ DO
                 $x := x - 1$   $\leftarrow$ action $\alpha$
$\ell_2 \rightarrow$             $y := y + 1$   $\leftarrow$ action $\beta$
         OD
$\ell_3 \rightarrow$   ...
```

program graph





Typed variables

typed variable: variable $\boldsymbol{x}$ + data domain $\boldsymbol{Dom(x)}$

Typed variables

typed variable: variable x + data domain $Dom(x)$

- Boolean variable: variable x with $Dom(x) = \{0, 1\}$
- integer variable: variable y with $Dom(y) = \mathbb{N}$
- variable z with $Dom(z) = \{\textbf{yellow}, \textbf{red}, \textbf{blue}\}$

Typed variables

typed variable: variable x + data domain $Dom(x)$

- Boolean variable: variable x with $Dom(x) = \{0, 1\}$
- integer variable: variable y with $Dom(y) = \mathbb{N}$
- variable z with $Dom(z) = \{\text{yellow, red, blue}\}$

evaluation for a set Var of typed variables:

type-consistent function $\eta : Var \rightarrow Values$

Typed variables

typed variable: variable x + data domain $Dom(x)$

- Boolean variable: variable x with $Dom(x) = \{0, 1\}$
- integer variable: variable y with $Dom(y) = \mathbb{N}$
- variable z with $Dom(z) = \{\text{yellow, red, blue}\}$

evaluation for a set Var of typed variables:

type-consistent function $\eta : Var \longrightarrow Values$

↑ $\eta(x) \in Dom(x)$ for all $x \in Var$

↑ $Values = \bigcup\limits_{x \in Var} Dom(x)$

Typed variables

typed variable: variable x + data domain $Dom(x)$

- Boolean variable: variable x with $Dom(x) = \{0, 1\}$
- integer variable: variable y with $Dom(y) = \mathbb{N}$
- variable z with $Dom(z) = \{\text{yellow, red, blue}\}$

evaluation for a set Var of typed variables:

type-consistent function $\eta : Var \to Values$

$\uparrow$

$\eta(x) \in Dom(x)$
for all $x \in Var$

$\uparrow$

$$Values = \bigcup_{x \in Var} Dom(x)$$

Notation: $Eval(Var)$ = set of evaluations for Var

Conditions on typed variables

If *Var* is a set of typed variables then

$$\boxed{\mathit{Cond}(\mathit{Var}) = \begin{array}{l}\text{set of Boolean conditions} \\ \quad\text{on the variables in } \mathit{Var}\end{array}}$$

Conditions on typed variables

If *Var* is a set of typed variables then

$$\boxed{\mathit{Cond}(\mathit{Var}) = \text{ set of Boolean conditions on the variables in } \mathit{Var}}$$

Example: $(\neg x \wedge y{<}z{+}3) \vee w{=}\mathit{red}$

where $\mathit{Dom}(x) = \{0, 1\}$, $\mathit{Dom}(y) = \mathit{Dom}(z) = \mathbb{N}$, $\mathit{Dom}(w) = \{\mathit{yellow}, \mathit{red}, \mathit{blue}\}$

Conditions on typed variables

If *Var* is a set of typed variables then

$$\boxed{\textit{Cond}(\textit{Var}) = \text{ set of Boolean conditions on the variables in } \textit{Var}}$$

Example: $(\neg x \wedge y < z+3) \vee w = red$

where $Dom(x) = \{0,1\}$, $Dom(y) = Dom(z) = \mathbb{N}$, $Dom(w) = \{yellow, red, blue\}$

satisfaction relation $\models$ for evaluations and conditions

Conditions on typed variables

If Var is a set of typed variables then

$$\boxed{Cond(Var) = \text{ set of Boolean conditions on the variables in } Var}$$

Example: $\big(\neg x \wedge y{<}z{+}3\big) \vee w{=}red$

where $Dom(x) = \{0,1\}$, $Dom(y) = Dom(z) = \mathbb{N}$, $Dom(w) = \{yellow, red, blue\}$

satisfaction relation $\models$ for evaluations and conditions

Example:

$$[x{=}0, y{=}3, z{=}6] \models \neg x \wedge y{<}z$$

$$[x{=}0, y{=}3, z{=}6] \not\models x \vee y{=}z$$

Effect-function for actions

Given a set ***Act*** of actions that operate on the variables in ***Var***, the effect of the actions is formalized by:

Effect-function for actions

Given a set *Act* of actions that operate on the variables in *Var*, the effect of the actions is formalized by:

$$\mathit{Effect} : \mathit{Act} \times \mathit{Eval}(\mathit{Var}) \to \mathit{Eval}(\mathit{Var})$$

Effect-function for actions

Given a set *Act* of actions that operate on the variables in *Var*, the effect of the actions is formalized by:

$$\mathit{Effect} : \mathit{Act} \times \mathit{Eval}(\mathit{Var}) \to \mathit{Eval}(\mathit{Var})$$

if α is "$x{:=}2x{+}y$" then:

$$\mathit{Effect}(\alpha, [x{=}1, y{=}3, \ldots]) \;=\; [x{=}5, y{=}3, \ldots]$$

Effect-function for actions

Given a set *Act* of actions that operate on the variables in *Var*, the effect of the actions is formalized by:

$$Effect : Act \times Eval(Var) \rightarrow Eval(Var)$$

if α is "$x{:=}2x{+}y$" then:

$$Effect(\alpha, [x{=}1, y{=}3, \ldots]) \;=\; [x{=}5, y{=}3, \ldots]$$

if β is "$x{:=}2x{+}y\,;\; y{:=}1{-}x$" then:

$$Effect(\beta, [x{=}1, y{=}3, \ldots]) \;=\; [x{=}5, y{=}{-}4, \ldots]$$

Effect-function for actions

Given a set *Act* of actions that operate on the variables in *Var*, the effect of the actions is formalized by:

$$Effect : Act \times Eval(Var) \to Eval(Var)$$

if α is "$x{:=}2x{+}y$" then:

$$Effect(\alpha, [x{=}1, y{=}3, \ldots]) \;=\; [x{=}5, y{=}3, \ldots]$$

if β is "$x{:=}2x{+}y\,;\; y{:=}1{-}x$" then:

$$Effect(\beta, [x{=}1, y{=}3, \ldots]) \;=\; [x{=}5, y{=}{-}4, \ldots]$$

if γ is "$(x, y) := (2x{+}y, 1{-}x)$" then:

$$Effect(\gamma, [x{=}1, y{=}3, \ldots]) \;=\; [x{=}5, y{=}0, \ldots]$$

Program graph (PG)

Program graph (PG)

Let ***Var*** be a set of typed variables.

A *program graph* over ***Var*** is a tuple

$$\mathcal{P} \;=\; (\mathit{Loc}, \mathit{Act}, \mathit{Effect}, \hookrightarrow, \mathit{Loc}_0, g_0) \text{ where}$$

Program graph (PG)

Let Var be a set of typed variables.

A *program graph* over Var is a tuple

$$\mathcal{P} \ = \ (\mathit{Loc}, \mathit{Act}, \mathit{Effect}, \hookrightarrow, \mathit{Loc}_0, g_0) \text{ where}$$

- Loc is a (finite) set of locations, i.e., control states,

Program graph (PG)

Let Var be a set of typed variables.

A *program graph* over Var is a tuple

$$\mathcal{P} \;=\; (\mathit{Loc}, \mathit{Act}, \mathit{Effect}, \hookrightarrow, \mathit{Loc}_0, g_0) \text{ where}$$

- Loc is a (finite) set of locations, i.e., control states,
- Act a set of actions,

Program graph (PG)

Let Var be a set of typed variables.

A *program graph* over Var is a tuple

$$\mathcal{P} \; = \; (\mathit{Loc}, \mathit{Act}, \mathit{Effect}, \hookrightarrow, \mathit{Loc}_0, g_0) \text{ where}$$

- Loc is a (finite) set of locations, i.e., control states,
- Act a set of actions,
- $\mathit{Effect} : \mathit{Act} \times \mathit{Eval}(\mathit{Var}) \to \mathit{Eval}(\mathit{Var})$

Program graph (PG)

Let Var be a set of typed variables.

A *program graph* over Var is a tuple

$$\mathcal{P} = (\mathit{Loc}, \mathit{Act}, \mathit{Effect}, \hookrightarrow, \mathit{Loc}_0, g_0) \text{ where}$$

- Loc is a (finite) set of locations, i.e., control states,
- Act a set of actions,
- $\mathit{Effect} : \mathit{Act} \times \mathit{Eval}(\mathit{Var}) \to \mathit{Eval}(\mathit{Var})$

↑

function that formalizes the effect of the actions

Program graph (PG)

TRANSSYS/TS-PROGRAM-GRAPH-DEF

Let Var be a set of typed variables.

A *program graph* over Var is a tuple

$$\mathcal{P} \;=\; (\mathit{Loc}, \mathit{Act}, \mathit{Effect}, \hookrightarrow, \mathit{Loc}_0, g_0) \text{ where}$$

- Loc is a (finite) set of locations, i.e., control states,
- Act a set of actions,
- $\mathit{Effect} : \mathit{Act} \times \mathit{Eval}(\mathit{Var}) \rightarrow \mathit{Eval}(\mathit{Var})$

↑

function that formalizes the effect of the actions

example: if α is the assignment $x{:=}x{+}y$ then

$$\mathit{Effect}(\alpha, [x{=}1, y{=}7]) \;=\; [x{=}8, y{=}7]$$

Program graph (PG)

Let Var be a set of typed variables.

A *program graph* over Var is a tuple

$$\mathcal{P} = (\mathit{Loc}, \mathit{Act}, \mathit{Effect}, \hookrightarrow, \mathit{Loc}_0, g_0) \text{ where}$$

- Loc is a (finite) set of locations, i.e., control states,
- Act a set of actions,
- $\mathit{Effect} : \mathit{Act} \times \mathit{Eval}(\mathit{Var}) \to \mathit{Eval}(\mathit{Var})$
- $\hookrightarrow \subseteq \mathit{Loc} \times \mathit{Cond}(\mathit{Var}) \times \mathit{Act} \times \mathit{Loc}$

Program graph (PG)

Let *Var* be a set of typed variables.

A *program graph* over *Var* is a tuple

$$\mathcal{P} \;=\; (\mathit{Loc}, \mathit{Act}, \mathit{Effect}, \hookrightarrow, \mathit{Loc}_0, g_0) \text{ where}$$

- Loc is a (finite) set of locations, i.e., control states,
- Act a set of actions,
- $\mathit{Effect} : \mathit{Act} \times \mathit{Eval}(\mathit{Var}) \to \mathit{Eval}(\mathit{Var})$
- $\hookrightarrow \;\subseteq\; \mathit{Loc} \times \mathit{Cond}(\mathit{Var}) \times \mathit{Act} \times \mathit{Loc}$

  specifies conditional transitions of the form $\ell \xhookrightarrow{g:\alpha} \ell'$

ℓ, ℓ' are locations, $g \in \mathit{Cond}(\mathit{Var})$, $\alpha \in \mathit{Act}$

Program graph (PG)

Let Var be a set of typed variables.

A *program graph* over Var is a tuple

$$\mathcal{P} \; = \; (\mathit{Loc}, \mathit{Act}, \mathit{Effect}, \hookrightarrow, \mathit{Loc}_0, g_0) \text{ where}$$

- Loc is a (finite) set of locations, i.e., control states,
- Act a set of actions,
- $\mathit{Effect} : \mathit{Act} \times \mathit{Eval}(\mathit{Var}) \rightarrow \mathit{Eval}(\mathit{Var})$
- $\hookrightarrow \; \subseteq \; \mathit{Loc} \times \mathit{Cond}(\mathit{Var}) \times \mathit{Act} \times \mathit{Loc}$

  specifies conditional transitions of the form $\ell \xrightarrow{g:\alpha} \ell'$
- $\mathit{Loc}_0 \subseteq \mathit{Loc}$ is the set of initial locations,

Program graph (PG)

Let Var be a set of typed variables.

A *program graph* over Var is a tuple

$$\mathcal{P} \;=\; (\mathit{Loc}, \mathit{Act}, \mathit{Effect}, \hookrightarrow, \mathit{Loc}_0, g_0) \text{ where}$$

- Loc is a (finite) set of locations, i.e., control states,
- Act a set of actions,
- $\mathit{Effect} : \mathit{Act} \times \mathit{Eval}(\mathit{Var}) \to \mathit{Eval}(\mathit{Var})$
- $\hookrightarrow \;\subseteq\; \mathit{Loc} \times \mathit{Cond}(\mathit{Var}) \times \mathit{Act} \times \mathit{Loc}$

  specifies conditional transitions of the form $\ell \xhookrightarrow{g:\alpha} \ell'$
- $\mathit{Loc}_0 \subseteq \mathit{Loc}$ is the set of initial locations,
- $g_0 \in \mathit{Cond}(\mathit{Var})$ initial condition on the variables

Program graph (PG)

Let Var be a set of typed variables.

A *program graph* over Var is a tuple

$$\mathcal{P} \;=\; (\mathit{Loc}, \mathit{Act}, \mathit{Effect}, \hookrightarrow, \mathit{Loc}_0, g_0) \text{ where}$$

- Loc is a (finite) set of locations, i.e., control states,
- Act a set of actions,
- $\mathit{Effect} : \mathit{Act} \times \mathit{Eval}(\mathit{Var}) \rightarrow \mathit{Eval}(\mathit{Var})$
- $\hookrightarrow \; \subseteq \; \mathit{Loc} \times \mathit{Cond}(\mathit{Var}) \times \mathit{Act} \times \mathit{Loc}$

  specifies conditional transitions of the form $\ell \overset{g:\alpha}{\hookrightarrow} \ell'$
- $\mathit{Loc}_0 \subseteq \mathit{Loc}$ is the set of initial locations,
- $g_0 \in \mathit{Cond}(\mathit{Var})$ initial condition on the variables.

TS-semantics of a program graph

TS-PG-SEM

TS-semantics of a program graph

$$\boxed{\begin{array}{c}\text{program graph } \mathcal{P} \text{ over } \textbf{\textit{Var}} \\ \Downarrow \\ \text{transition system } \mathcal{T}_{\mathcal{P}}\end{array}}$$

TS-semantics of a program graph

program graph $\mathcal{P}$ over ***Var***

$$\Downarrow$$

transition system $\mathcal{T}_{\mathcal{P}}$

states in $\mathcal{T}_{\mathcal{P}}$ have the form

$$\langle \ell, \eta \rangle$$

location   variable evaluation

TS-semantics of a program graph

Let $\mathcal{P} = (\mathit{Loc}, \mathit{Act}, \mathit{Effect}, \hookrightarrow, \mathit{Loc}_0, g_0)$ be a PG.
The transition system of $\mathcal{P}$ is:

$$\mathcal{T}_{\mathcal{P}} \; = \; (S, \mathit{Act}, \longrightarrow, S_0, AP, L)$$

TS-semantics of a program graph

Let $\mathcal{P} = (Loc, Act, Effect, \hookrightarrow, Loc_0, g_0)$ be a PG.
The transition system of $\mathcal{P}$ is:

$$\mathcal{T}_{\mathcal{P}} \; = \; (S, Act, \longrightarrow, S_0, AP, L)$$

- state space: $S = Loc \times Eval(Var)$

TS-semantics of a program graph

Let $\mathcal{P} = (Loc, Act, Effect, \hookrightarrow, Loc_0, g_0)$ be a PG.
The transition system of $\mathcal{P}$ is:

$$\mathcal{T}_{\mathcal{P}} \;=\; (S, Act, \longrightarrow, S_0, AP, L)$$

- state space: $S = Loc \times Eval(Var)$
- initial states: $S_0 = \{\langle \ell, \eta \rangle : \ell \in Loc_0, \eta \models g_0\}$

TS-semantics of a program graph

Let $\mathcal{P} = (Loc, Act, Effect, \hookrightarrow, Loc_0, g_0)$ be a PG.
The transition system of $\mathcal{P}$ is:

$$\mathcal{T}_{\mathcal{P}} \;=\; (S, Act, \longrightarrow, S_0, AP, L)$$

- state space: $S = Loc \times Eval(Var)$
- initial states: $S_0 = \big\{ \langle \ell, \eta \rangle : \ell \in Loc_0, \eta \models g_0 \big\}$

The transition relation $\longrightarrow$ is given by the following rule:

$$\frac{\ell \overset{g:\alpha}{\hookrightarrow} \ell' \;\wedge\; \eta \models g}{\langle \ell, \eta \rangle \xrightarrow{\alpha} \langle \ell', Effect(\alpha, \eta) \rangle}$$

Structured operational semantics (SOS)

The transition system of a program graph $\mathcal{P}$ is

$$\mathcal{T}_{\mathcal{P}} = (S, Act, \longrightarrow, S_0, AP, L) \text{ where}$$

the transition relation $\longrightarrow$ is given by the following rule

$$\frac{\ell \xhookrightarrow{g:\alpha} \ell' \ \wedge \ \eta \models g}{\langle \ell, \eta \rangle \xrightarrow{\alpha} \langle \ell', Effect(\alpha, \eta) \rangle}$$

is a shortform notation in **SOS**-style.

$$\frac{\textit{premise}}{\textit{conclusion}}$$

Structured operational semantics (SOS)

The transition system of a program graph $\mathcal{P}$ is

$$\mathcal{T}_{\mathcal{P}} \ = \ (S, Act, \longrightarrow, S_0, AP, L) \ \text{ where}$$

the transition relation $\longrightarrow$ is given by the following rule

$$\frac{\ell \xhookrightarrow{g:\alpha} \ell' \ \wedge \ \eta \models g}{\langle \ell, \eta \rangle \xrightarrow{\alpha} \langle \ell', Effect(\alpha, \eta) \rangle}$$

is a shortform notation in **SOS**-style.

It means that $\longrightarrow$ is the smallest relation such that:

$$\boxed{\text{if } \ell \xhookrightarrow{g:\alpha} \ell' \ \wedge \ \eta \models g \text{ then } \langle \ell, \eta \rangle \xrightarrow{\alpha} \langle \ell', Effect(\alpha, \eta) \rangle}$$

TS-semantics of a program graph

Let $\mathcal{P} = (Loc, Act, Effect, \hookrightarrow, Loc_0, g_0)$ be a PG.

transition system $\mathcal{T}_{\mathcal{P}} \; = \; (S, Act, \longrightarrow, S_0, AP, L)$

- state space: $S = Loc \times Eval(Var)$
- initial states: $S_0 = \big\{ \langle \ell, \eta \rangle : \ell \in Loc_0, \eta \models g_0 \big\}$
- $\longrightarrow$ is given by the following rule:

$$\frac{\ell \overset{g:\alpha}{\hookrightarrow} \ell' \;\wedge\; \eta \models g}{\langle \ell, \eta \rangle \xrightarrow{\alpha} \langle \ell', Effect(\alpha, \eta) \rangle}$$

Labeling of the states

Let $\mathcal{P} = (Loc, Act, Effect, \hookrightarrow, Loc_0, g_0)$ be a PG.

transition system $\mathcal{T}_{\mathcal{P}} = (S, Act, \longrightarrow, S_0, AP, L)$

- state space: $S = Loc \times Eval(Var)$
- initial states: $S_0 = \{\langle \ell, \eta \rangle : \ell \in Loc_0, \eta \models g_0\}$
- $\longrightarrow$ is given by the following rule:

$$\frac{\ell \xhookrightarrow{g:\alpha} \ell' \ \wedge \ \eta \models g}{\langle \ell, \eta \rangle \xrightarrow{\alpha} \langle \ell', Effect(\alpha, \eta) \rangle}$$

- atomic propositions: $AP = Loc \cup Cond(Var)$

Labeling of the states

Let $\mathcal{P} = (Loc, Act, Effect, \hookrightarrow, Loc_0, g_0)$ be a PG.

transition system $\mathcal{T}_{\mathcal{P}} = (S, Act, \longrightarrow, S_0, AP, L)$

- state space: $S = Loc \times Eval(Var)$
- initial states: $S_0 = \{\langle \ell, \eta \rangle : \ell \in Loc_0, \eta \models g_0\}$
- $\longrightarrow$ is given by the following rule:

$$\frac{\ell \xhookrightarrow{g:\alpha} \ell' \ \wedge \ \eta \models g}{\langle \ell, \eta \rangle \xrightarrow{\alpha} \langle \ell', Effect(\alpha, \eta) \rangle}$$

- atomic propositions: $AP = Loc \cup Cond(Var)$
- labeling function:

$$L(\langle \ell, \eta \rangle) = \{\ell\} \cup \{g \in Cond(Var) : \eta \models g\}$$

TS-semantics of a program graph

Let $\mathcal{P} = (Loc, Act, Effect, \hookrightarrow, Loc_0, g_0)$ be a PG.

transition system $\mathcal{T}_{\mathcal{P}} \ = \ (S, Act, \longrightarrow, S_0, AP, L)$

- state space: $S = Loc \times Eval(Var)$
- initial states: $S_0 = \{\langle \ell, \eta \rangle : \ell \in Loc_0, \eta \models g_0\}$
- $\longrightarrow$ is given by the following rule:

$$\frac{\ell \xhookrightarrow{g:\alpha} \ell' \ \wedge \ \eta \models g}{\langle \ell, \eta \rangle \xrightarrow{\alpha} \langle \ell', Effect(\eta, \alpha) \rangle}$$

- atomic propositions: $AP = Loc \cup Cond(Var)$
- labeling function:

$$L(\langle \ell, \eta \rangle) \ = \ \{\ell\} \cup \{g \in Cond(Var) : \eta \models g\}$$

Guarded Command Language (GCL)

by Dijkstra

Guarded Command Language (GCL)

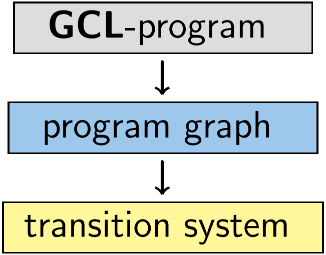
by Dijkstra

- high-level modeling language that contains features of imperative languages and nondeterministic choice

Guarded Command Language (GCL)

by Dijkstra

- high-level modeling language that contains features of imperative languages and nondeterministic choice
- semantics:



GCL-program

↓

program graph

↓

transition system

Guarded Command Language (GCL)

guarded command $g \Rightarrow \textit{stmt}$

g	:	guard, i.e., Boolean condition on the program variables
stmt	:	statement

Guarded Command Language (GCL)

guarded command $g \Rightarrow \textit{stmt}$ ← enabled if g is true

g	:	guard, i.e., Boolean condition on the program variables
stmt	:	statement

Guarded Command Language (GCL)

guarded command $g \Rightarrow$ ***stmt*** ⟵ enabled if g is true

g : guard, i.e., Boolean condition on the program variables

stmt : statement

repetitive command/loop:

DO :: $g \Rightarrow$ ***stmt*** OD

Guarded Command Language (GCL)

guarded command $g \Rightarrow \textit{stmt}$ ← enabled if g is true

g : guard, i.e., Boolean condition on the program variables

$\textit{stmt}$: statement

repetitive command/loop:

DO :: $g \Rightarrow \textit{stmt}$ OD ← WHILE g DO $\textit{stmt}$ OD

Guarded Command Language (GCL)

guarded command $g \Rightarrow \mathit{stmt}$ ← enabled if g is true

g : guard, i.e., Boolean condition on the program variables

stmt : statement

repetitive command/loop:

`DO` $:: g \Rightarrow \mathit{stmt}$ `OD` ← `WHILE` g `DO` stmt `OD`

conditional command:

`IF` $:: \; g \Rightarrow \mathit{stmt}_1$

$\quad :: \neg g \Rightarrow \mathit{stmt}_2$

`FI`

Guarded Command Language (GCL)

TS1.4-15

guarded command $g \Rightarrow \mathit{stmt}$ ← enabled if g is true

g	:	guard, i.e., Boolean condition on the program variables
stmt	:	statement

repetitive command/loop:

```
DO :: g ⇒ stmt OD        ←   WHILE g DO stmt OD
```

conditional command:

```
IF  ::  g ⇒ stmt₁              IF g THEN stmt₁
    :: ¬g ⇒ stmt₂        ←          ELSE stmt₂
FI                             FI
```

Guarded Command Language (GCL)

guarded command $g \Rightarrow \mathit{stmt}$ ← enabled if g is true

repetitive command/loop:

$\mathtt{DO} \;::\; g \Rightarrow \mathit{stmt}\; \mathtt{OD}$ ← $\mathtt{WHILE}\; g\; \mathtt{DO}\; \mathit{stmt}\; \mathtt{OD}$

conditional command:

$$\begin{array}{lll} \mathtt{IF} & :: & g \Rightarrow \mathit{stmt}_1 \\ & :: & \neg g \Rightarrow \mathit{stmt}_2 \\ \mathtt{FI} & & \end{array} \quad \longleftarrow \quad \begin{array}{llll} \mathtt{IF} & g & \mathtt{THEN} & \mathit{stmt}_1 \\ & & \mathtt{ELSE} & \mathit{stmt}_2 \\ \mathtt{FI} & & & \end{array}$$

symbol :: stands for the nondeterministic choice between enabled guarded commands

Guarded Command Language (GCL)

modeling language with nondeterministic choice

$$
\begin{array}{lll}
stmt & \overset{\text{def}}{=} & x := expr \quad \mid \quad stmt_1 ; stmt_2 \quad \mid \\
 & & \texttt{DO} :: g_1 \Rightarrow stmt_1 \quad \ldots \quad :: g_n \Rightarrow stmt_n \ \texttt{OD} \\
 & & \texttt{IF} :: g_1 \Rightarrow stmt_1 \quad \ldots \quad :: g_n \Rightarrow stmt_n \ \texttt{FI} \\
 & & \vdots
\end{array}
$$

where x is a typed variable and $expr$ an expression of the same type

Guarded Command Language (GCL)

modeling language with nondeterministic choice

$$
\begin{array}{lcl}
\mathit{stmt} & \stackrel{\text{def}}{=} & x := \mathit{expr} \quad \big| \quad \mathit{stmt}_1 ; \mathit{stmt}_2 \quad \big| \\
 & & \texttt{DO} :: g_1 \Rightarrow \mathit{stmt}_1 \quad \ldots \quad :: g_n \Rightarrow \mathit{stmt}_n \ \texttt{OD} \\
 & & \texttt{IF} :: g_1 \Rightarrow \mathit{stmt}_1 \quad \ldots \quad :: g_n \Rightarrow \mathit{stmt}_n \ \texttt{FI} \\
 & & \vdots
\end{array}
$$

where *x* is a typed variable and *expr* an expression of the same type

semantics of a **GCL**-program: program graph

GCL-program for beverage machine

GCL-program for beverage machine

uses two variables $\#\mathit{sprite}, \#\mathit{coke} \in \{0, 1, \ldots, \mathit{max}\}$
for the number of available drinks (sprite or coke)

GCL-program for beverage machine

pc2.1-GCL-getraenkeautomat

uses two variables $\#\mathit{sprite}, \#\mathit{coke} \in \{0, 1, \ldots, \mathit{max}\}$
for the number of available drinks (sprite or coke)

uses the following actions:

	enabled	effect
get_coke	if $\#\mathit{coke} > 0$	$\#\mathit{coke} := \#\mathit{coke} - 1$
get_sprite	if $\#\mathit{sprite} > 0$	$\#\mathit{sprite} := \#\mathit{sprite} - 1$

GCL-program for beverage machine

uses two variables $\#sprite, \#coke \in \{0, 1, \ldots, max\}$
for the number of available drinks (sprite or coke)

uses the following actions:

	enabled	effect
get_coke	if $\#coke > 0$	$\#coke := \#coke - 1$
get_sprite	if $\#sprite > 0$	$\#sprite := \#sprite - 1$
refill	any time	$\#sprite := max$ $\#coke := max$

GCL-program for beverage machine

uses two variables $\#sprite, \#coke \in \{0, 1, \ldots, max\}$
for the number of available drinks (sprite or coke)

uses the following actions:

	enabled	effect
get_coke	if $\#coke > 0$	$\#coke := \#coke - 1$
get_sprite	if $\#sprite > 0$	$\#sprite := \#sprite - 1$
refill	any time	$\#sprite := max$ $\#coke := max$
insert_coin	any time	no effect on variables

GCL-program for beverage machine

uses two variables $\#sprite, \#coke \in \{0, 1, \ldots, max\}$
for the number of available drinks (sprite or coke)

uses the following actions:

	enabled	effect
get_coke	if $\#coke > 0$	$\#coke := \#coke - 1$
get_sprite	if $\#sprite > 0$	$\#sprite := \#sprite - 1$
refill	any time	$\#sprite := max$ $\#coke := max$
insert_coin	any time	no effect on variables
return_coin	if machine is empty and user has entered a coin (no effect on variables)	

GCL-program for beverage machine

```
DO  ::  true $\Rightarrow$ insert_coin;
        IF  ::  $\#sprite = \#coke = 0 \Rightarrow$ return_coin

            ::  $\#coke > 0 \Rightarrow \#coke := \#coke - 1$

            ::  $\#sprite > 0 \Rightarrow \#sprite := \#sprite - 1$

        FI
    ::  true $\Rightarrow \#sprite := max; \#coke := max$

OD
```

GCL-program for beverage machine

```
DO  ::  true ⇒ insert_coin; (* user inserts a coin *)
        IF  ::  $\#sprite = \#coke = 0 \Rightarrow$ return_coin

            ::  $\#coke > 0 \Rightarrow \#coke := \#coke - 1$

            ::  $\#sprite > 0 \Rightarrow \#sprite := \#sprite - 1$

        FI
    ::  true ⇒ $\#sprite := max; \#coke := max$
OD
```

GCL-program for beverage machine

```
DO :: true $\Rightarrow$ insert_coin; (* user inserts a coin *)
        IF :: $\#sprite = \#coke = 0 \Rightarrow$ return_coin
                                  (* no beverage available *)
           :: $\#coke > 0 \Rightarrow \#coke := \#coke - 1$

           :: $\#sprite > 0 \Rightarrow \#sprite := \#sprite - 1$
        FI
   :: true $\Rightarrow \#sprite := max; \#coke := max$
OD
```

GCL-program for beverage machine

```
DO  ::  $\text{true} \Rightarrow \text{insert\_coin}$; (* user inserts a coin *)
        IF  ::  $\#\mathit{sprite} = \#\mathit{coke} = 0 \Rightarrow \text{return\_coin}$
                                (* no beverage available *)
            ::  $\#\mathit{coke} > 0 \Rightarrow \#\mathit{coke} := \#\mathit{coke} - 1$
                                (* user selects coke *)
            ::  $\#\mathit{sprite} > 0 \Rightarrow \#\mathit{sprite} := \#\mathit{sprite} - 1$
                                (* user selects sprite *)
        FI
    ::  $\text{true} \Rightarrow \#\mathit{sprite} := \mathit{max}; \#\mathit{coke} := \mathit{max}$
                                (* refilling of the machine *)
OD
```

GCL-program for beverage machine

```
DO :: true ⇒ insert_coin; (* user inserts a coin *)
        IF :: $\#sprite = \#coke = 0$ ⇒ return_coin
                                  (* no beverage available *)
           :: $\#coke > 0$ ⇒ get_coke
                                  (* user selects coke *)
           :: $\#sprite > 0$ ⇒ get_sprite
                                  (* user selects sprite *)
        FI
   :: true ⇒ refill
                                  (* refilling of the machine *)
OD
```

GCL-program for beverage machine

```
DO :: true ⇒ insert_coin;
      IF :: $\#sprite = \#coke = 0$ ⇒ return_coin

         :: $\#coke > 0$ ⇒ get_coke

         :: $\#sprite > 0$ ⇒ get_sprite

      FI
   :: true ⇒ refill

OD
```

GCL-program for beverage machine

PC2.1-GCL-2

```
DO :: true ⇒ insert_coin;
        IF :: $\#sprite = \#coke = 0$
                                ⇒ return_coin
           :: $\#coke > 0$ ⇒ get_coke
           :: $\#sprite > 0$ ⇒ get_sprite
        FI
   :: true ⇒ refill
OD
```

... yields a program graph with

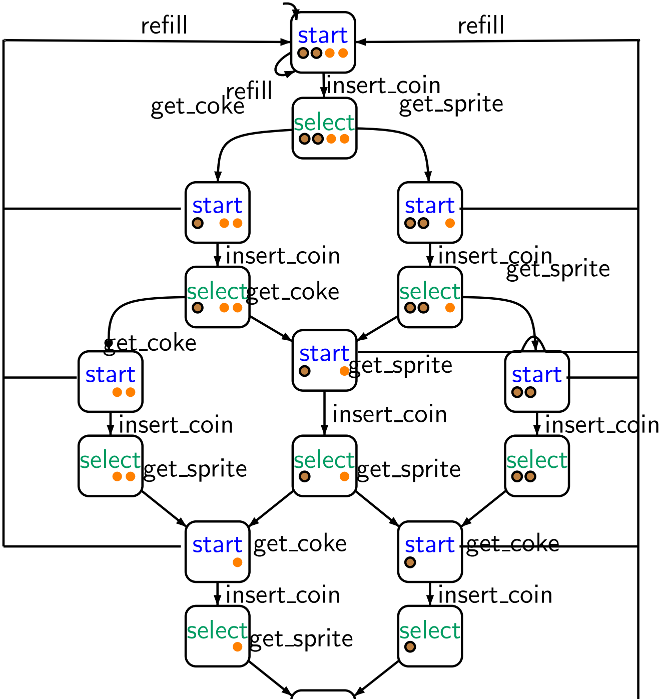
- two variables $\#sprite, \#coke \in \{0, 1, \ldots, max\}$

GCL-program for beverage machine

```
start  → DO :: true ⇒ insert_coin;
select →       IF :: #sprite = #coke = 0
                                 ⇒ return_coin
                  :: #coke  > 0  ⇒ get_coke
                  :: #sprite > 0 ⇒ get_sprite
               FI
            :: true ⇒ refill
         OD
```

... yields a program graph with

- two variables *#sprite*, *#coke* $\in \{0, 1, \ldots, \mathit{max}\}$
- two locations *start* and *select*


refill
start
refill
refill
get_coke
insert_coin
get_sprite
select
start
start
insert_coin
insert_coin
get_sprite
select
get_coke
select
get_coke
start
get_sprite
start
start
insert_coin
insert_coin
insert_coin
select
get_sprite
select
get_sprite
select
start
get_coke
start
get_coke
insert_coin
insert_coin
select
get_sprite
select