Overview

Introduction

Modelling parallel systems

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

Linear Time Properties

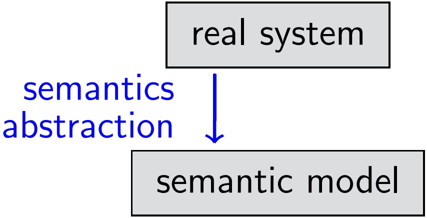
Regular Properties

Linear Temporal Logic

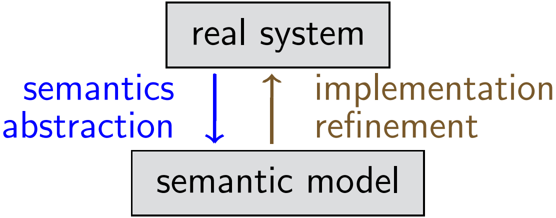
Computation-Tree Logic

Equivalences and Abstraction

Transition systems



Transition systems



Transition systems

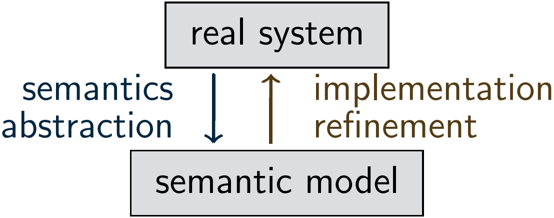
real system

semantics ↓ ↑ implementation
abstraction ↓ ↑ 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 ↓ ↑ implementation

abstraction ↓ ↑ 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 $\downarrow$ $\uparrow$ implementation, refinement

semantic model

The semantic model yields a formal representation of:

- the states of the system $\longleftarrow$ **nodes**

$\boxed{\text{control component} + \text{information on “relevant” data}}$

- the stepwise behaviour $\longleftarrow$ **edges**
- the initial states

Transition systems $\widehat{=}$ extended digraphs

real system

semantics abstraction $\downarrow$ $\uparrow$ implementation refinement

semantic model

The semantic model yields a formal representation of:

- the states of the system $\longleftarrow$ **nodes**

$\uparrow$

$\boxed{\text{control component} + \text{information on “relevant” data}}$

- the stepwise behaviour $\longleftarrow$ **transitions**
- the initial states

Transition systems $\widehat{=}$ extended digraphs

real system

semantics / abstraction $\downarrow$ $\uparrow$ implementation / refinement

semantic model

The semantic model yields a formal representation of:

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

Transition systems $\widehat{=}$ extended digraphs

$$\begin{tikzcd} \boxed{\text{real system}} \arrow[d, shift right=2, "\substack{\text{semantics}\\\text{abstraction}}"'] \\ \boxed{\text{semantic model}} \arrow[u, shift right=2, "\substack{\text{implementation}\\\text{refinement}}"'] \end{tikzcd}$$

The semantic model yields a formal representation of:

- the states of the system $\longleftarrow$ **nodes**
- the stepwise behaviour $\longleftarrow$ **transitions**
- the initial states
- additional information on
  communication $\longleftarrow$ **actions**
  state properties $\longleftarrow$ **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

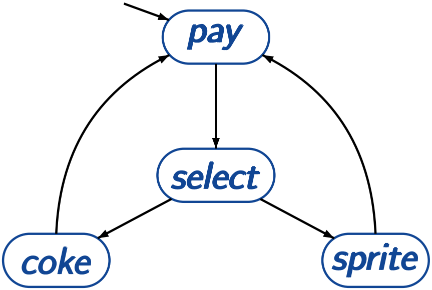
$$\mathcal{T} = (S, Act, \longrightarrow, S_0, AP, L)$$

- 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,

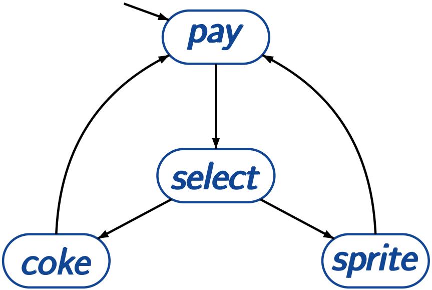
  i.e., transitions have the form $s \xrightarrow{\alpha} s'$
  where $s, s' \in S$ and $\alpha \in Act$

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

Transition system for beverage machine



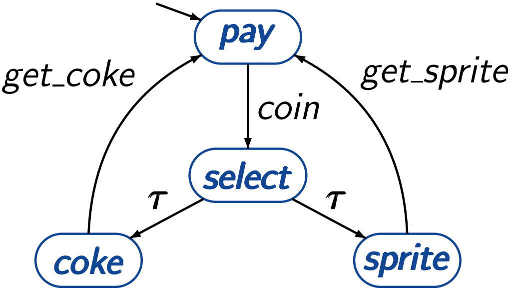
Transition system for beverage machine



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

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

Transition system for beverage machine

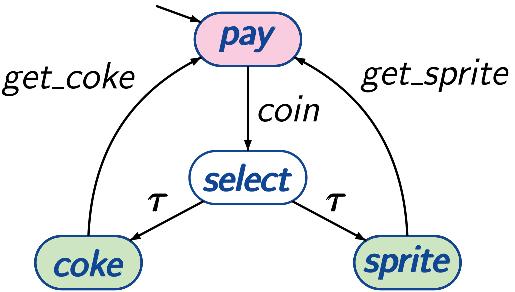


actions:
coin
τ
get_sprite
get_coke

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

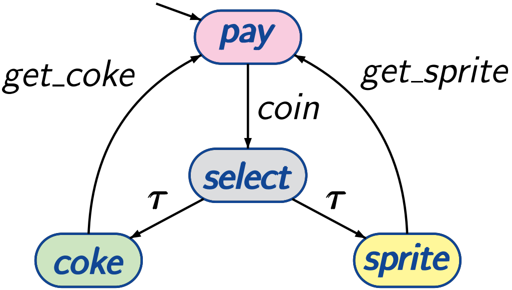
set of initial states: $S_0 = \{pay\}$

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 α and put $s := s'$
`OD`

"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 α 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

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 α 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:

$Reach(\mathcal{T})$ = 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. $\underbrace{x := x+1}_{\text{action } \alpha} \;|||\; \underbrace{y := y-3}_{\text{action } \beta}$ α, β independent

parallel execution of dependent actions

Transition system for parallel actions

parallel execution of independent actions

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

parallel execution of dependent actions

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

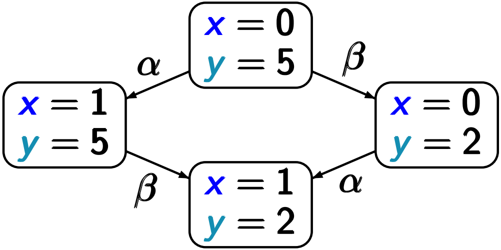
Transition system for parallel actions

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

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

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

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



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

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

$$
\begin{tikzcd}
 & \boxed{\begin{array}{l} x = 0 \\ y = 5 \end{array}} \arrow[dl, "\alpha"'] \arrow[dr, "\beta"] & \\
\boxed{\begin{array}{l} x = 1 \\ y = 5 \end{array}} \arrow[dr, "\beta"'] & & \boxed{\begin{array}{l} x = 0 \\ y = 2 \end{array}} \arrow[dl, "\alpha"] \\
 & \boxed{\begin{array}{l} x = 1 \\ y = 2 \end{array}} &
\end{tikzcd}
$$

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

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

parallel execution of independent actions $\longleftarrow$ interleaving

$$
\begin{tikzcd}
& \boxed{\begin{array}{l} x = 0 \\ y = 5 \end{array}} \arrow[dl, "\alpha"'] \arrow[dr, "\beta"] & \\
\boxed{\begin{array}{l} x = 1 \\ y = 5 \end{array}} \arrow[dr, "\beta"'] & & \boxed{\begin{array}{l} x = 0 \\ y = 2 \end{array}} \arrow[dl, "\alpha"] \\
& \boxed{\begin{array}{l} x = 1 \\ y = 2 \end{array}} &
\end{tikzcd}
$$

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

parallel execution of dependent actions $\longleftarrow$ competition

$$
\begin{tikzcd}
& \boxed{\begin{array}{l} x = 0 \\ y = 0 \end{array}} \arrow[dl, "\alpha"'] \arrow[dr, "\beta"] & \\
\boxed{\begin{array}{l} x = 1 \\ y = 0 \end{array}} \arrow[d, "\beta"'] & & \boxed{\begin{array}{l} x = 0 \\ y = 0 \end{array}} \arrow[d, "\alpha"] \\
\boxed{\begin{array}{l} x = 1 \\ y = 2 \end{array}} & & \boxed{\begin{array}{l} x = 1 \\ y = 0 \end{array}}
\end{tikzcd}
$$

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

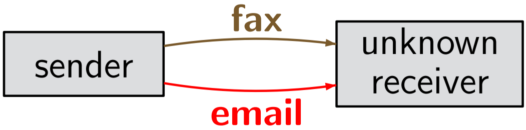
Possible meanings of nondeterminism in TS

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

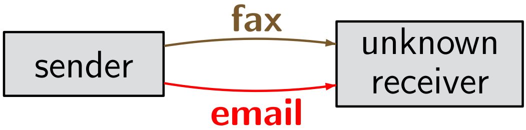
Implementation freedom

… modelled by nondeterminism

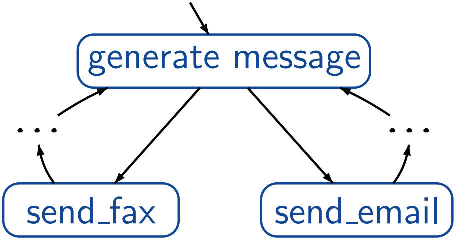
Implementation freedom



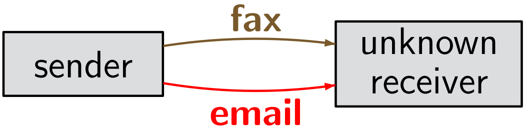
Implementation freedom



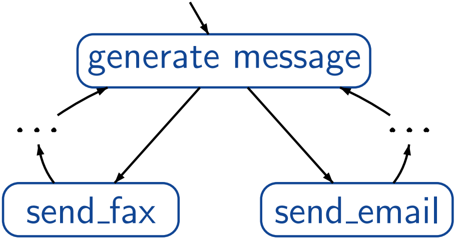
realization by a TS:



Implementation freedom



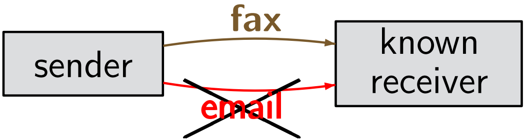
realization by a TS:



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

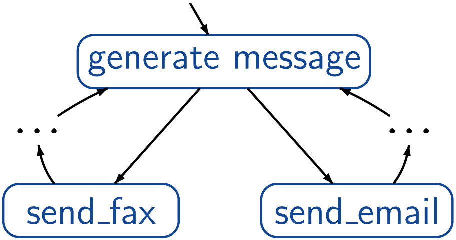
Implementation freedom

TS1.4-5



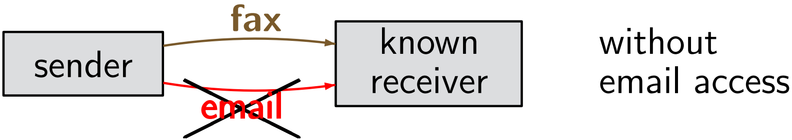
without
email access

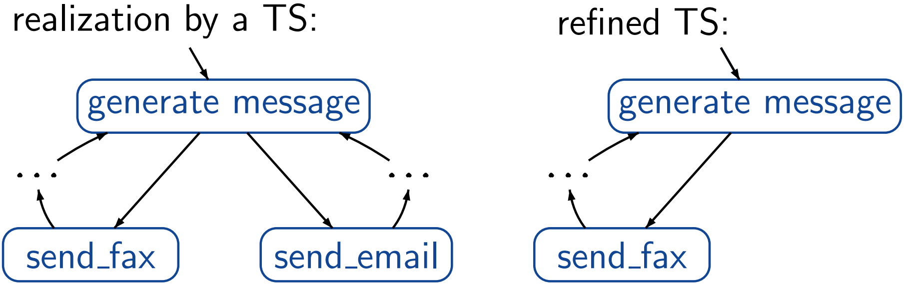
realization by a TS:



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

Implementation freedom

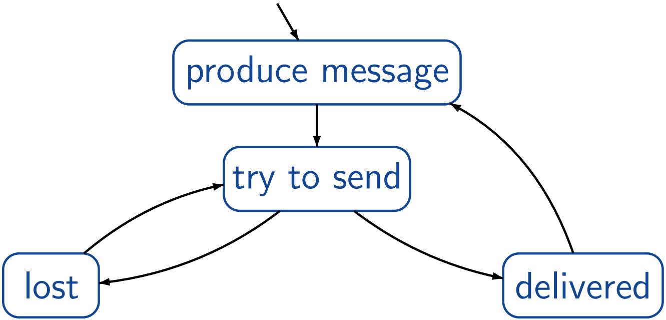




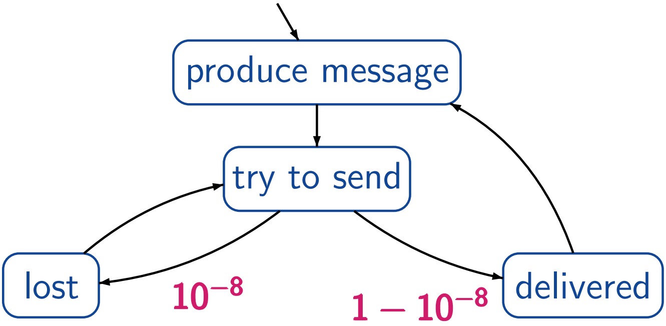
at a future refinement step the nondeterminism is replaced with one of the alternatives

Underspecification

Underspecification



Underspecification



at a future refinement step the nondeterminism is replaced with probabilism

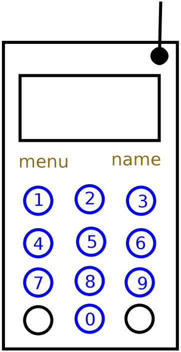
Possible meanings of nondeterminism in TS

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

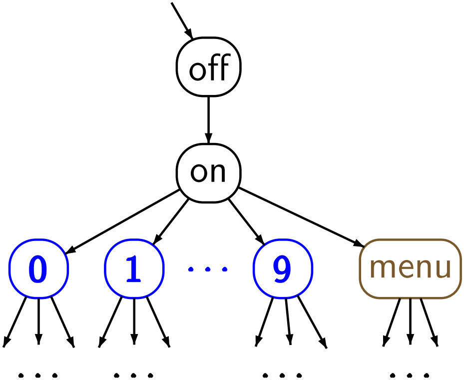
Possible meanings of nondeterminism in TS

- (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

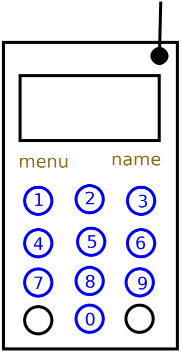
Incomplete information on the environment



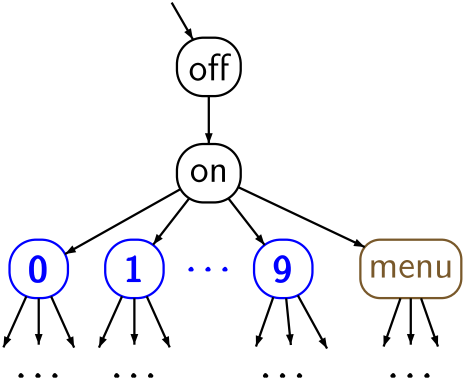
mobile phone



Incomplete information on the environment



mobile phone

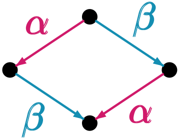


resolution of the nondeterministic choices
by a human user

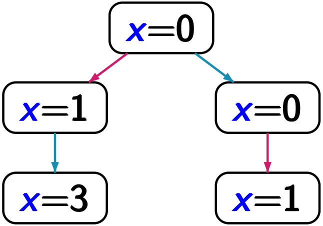
Possible meanings of nondeterminism in TS

concurrency (interleaving)

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



competitions



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

underspecification, implementational freedom

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

Introduction

Modelling parallel systems

- Transition systems
- Modeling hard- and software systems $\longleftarrow$
- Parallelism and communication

Linear Time Properties

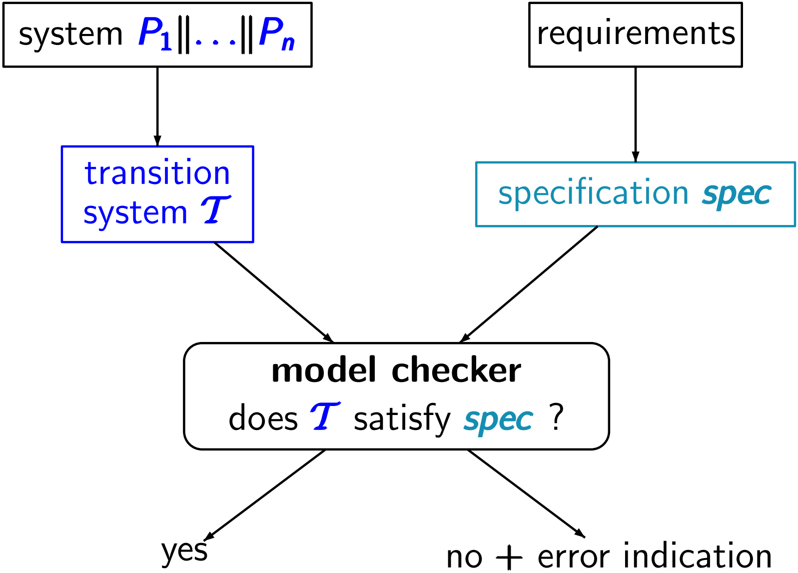
Regular Properties

Linear Temporal Logic

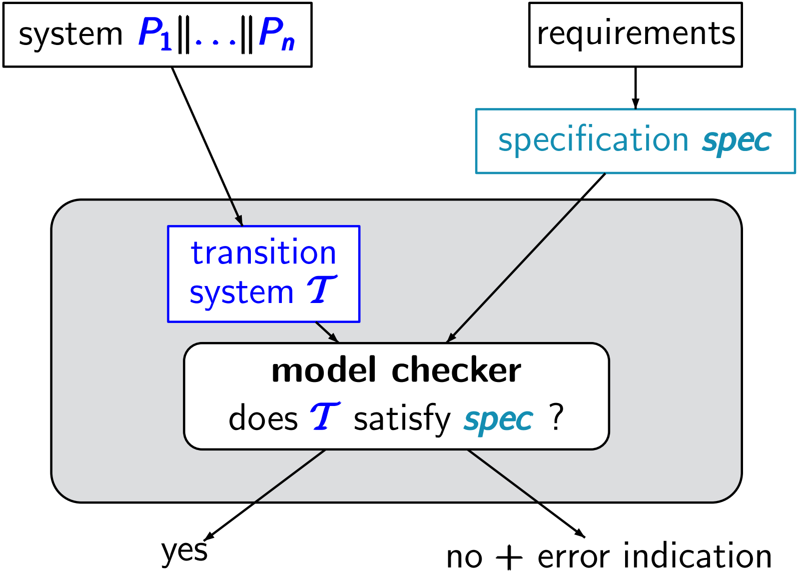
Computation-Tree Logic

Equivalences and Abstraction

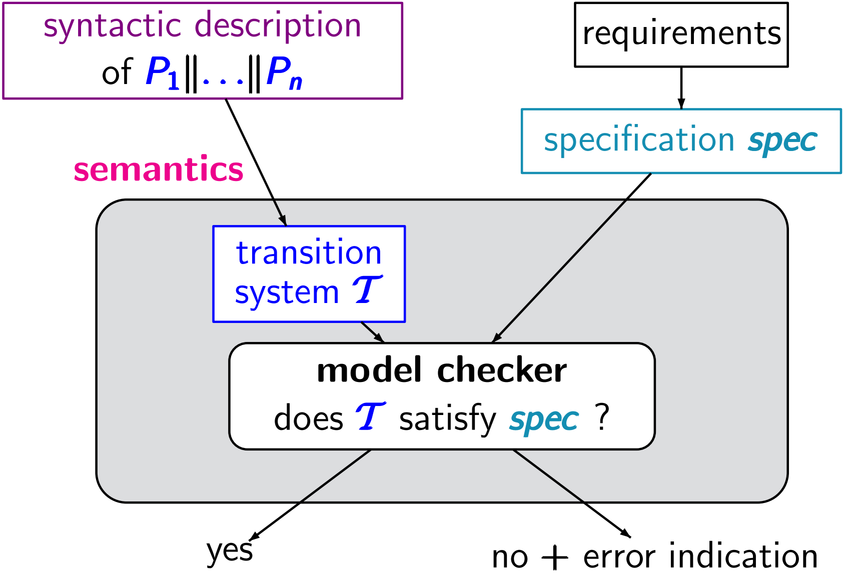
Model checking



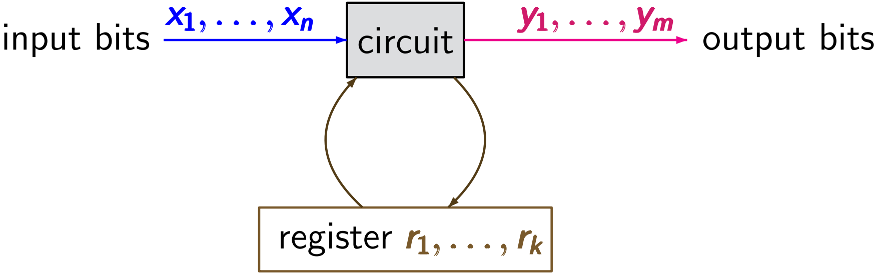
Model checking



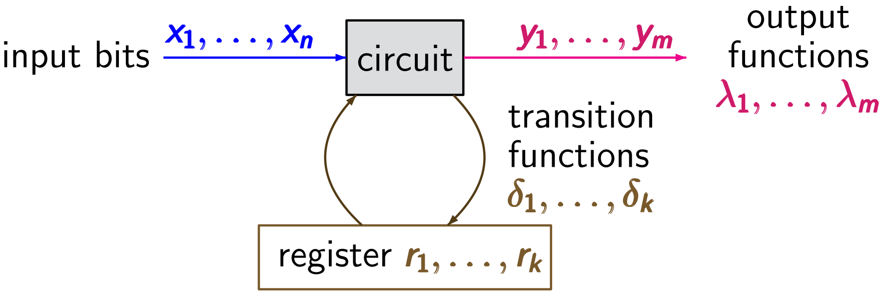
Model checking



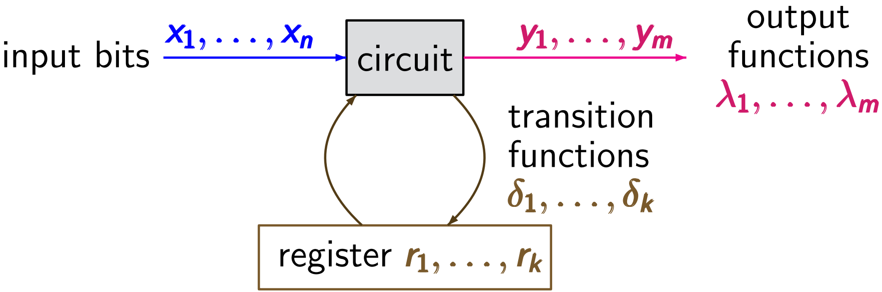
Modelling of sequential circuits by TS



Modelling of sequential circuits by TS

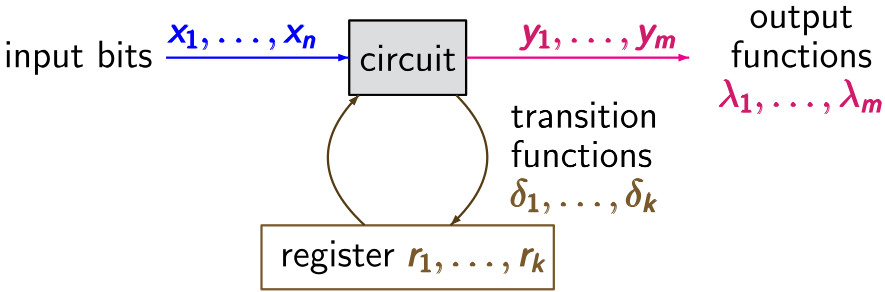


Modelling of sequential circuits by TS



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

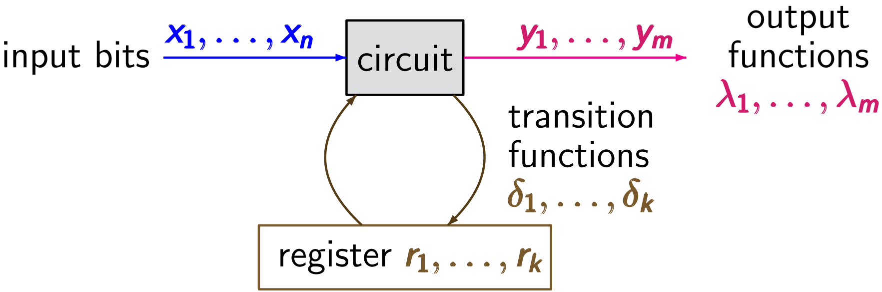
Modelling of sequential circuits by TS



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

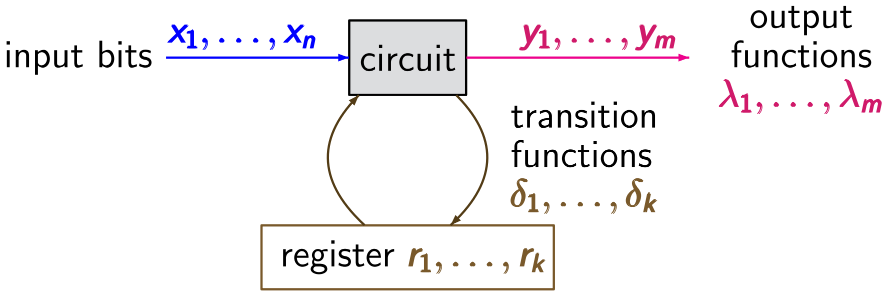
input values $a_1, \ldots, a_n$ for the input variables		output value $\lambda_i(\ldots)$ for output variable y_i
	$\mapsto$	
$+$ current values $c_1, \ldots, c_k$ of the registers		next value $\delta_j(\ldots)$ for register r_j

Modelling of sequential circuits by TS



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

Modelling of sequential circuits by TS

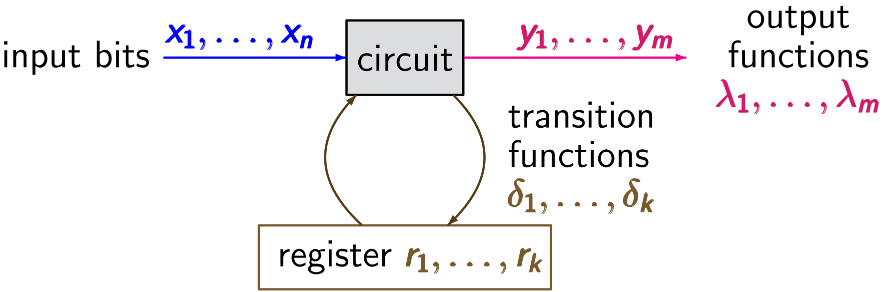


initial register evaluation $[r_1 = c_{01}, \ldots, r_k = c_{0k}]$

transition system:

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

Modelling of sequential circuits by TS

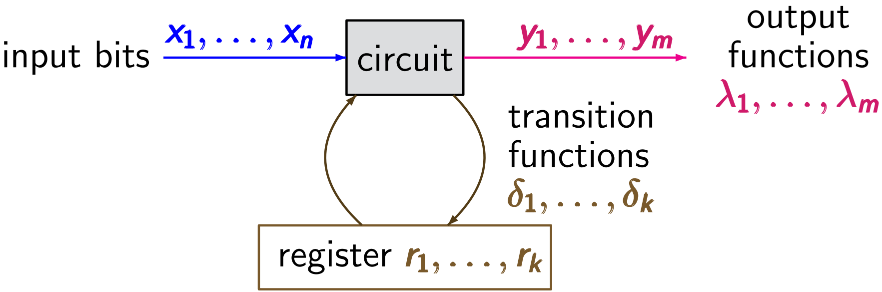


initial register evaluation $[r_1 = c_{01}, \ldots, r_k = c_{0k}]$

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

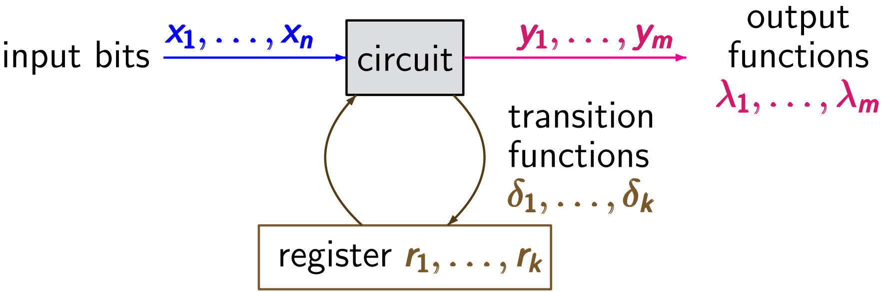


initial register evaluation $[r_1{=}c_{01}, \ldots, r_k{=}c_{0k}]$

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

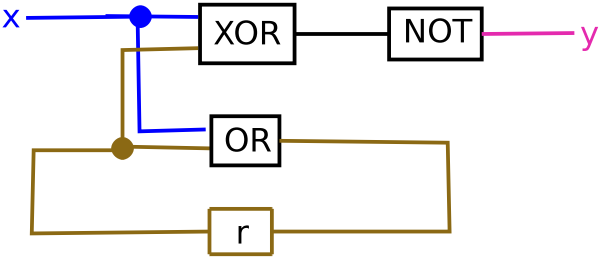


initial register evaluation $[r_1{=}c_{01}, \ldots, r_k{=}c_{0k}]$

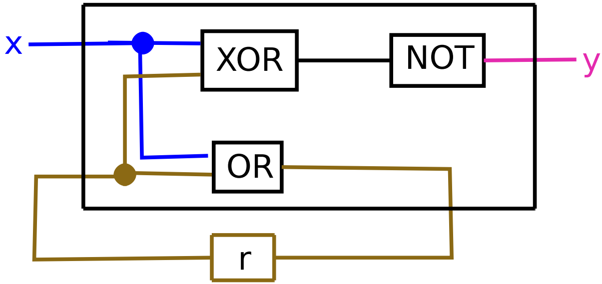
transition system:

- 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



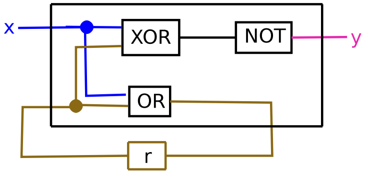
Example: sequential circuit



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

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

Example: TS for sequential circuit



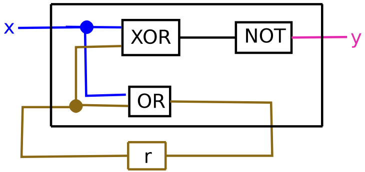
output function

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

transition function

$\delta_r = x \vee r$

Example: TS for sequential circuit



output function

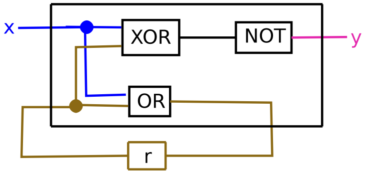
$\lambda_y = \neg(x \oplus r)$

transition function

$\delta_r = x \vee r$

transition system

Example: TS for sequential circuit



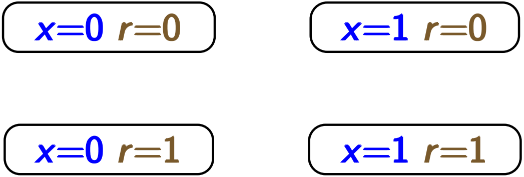
output function

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

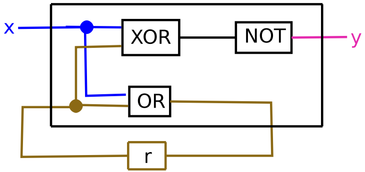
transition function

$\delta_r = x \vee r$

transition system



Example: TS for sequential circuit



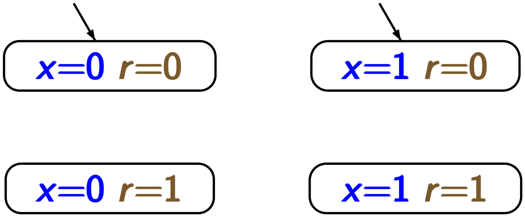
output function

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

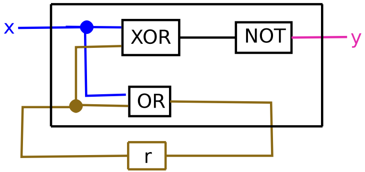
transition function

$\delta_r = x \vee r$

transition system



initial register evaluation: r=0



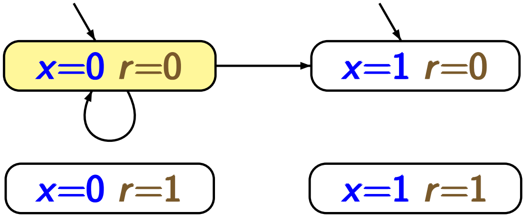
output function

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

transition function

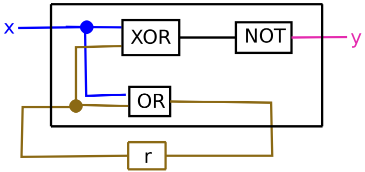
$\delta_r = x \vee r$

transition system



initial register evaluation: r=0

Example: TS for sequential circuit



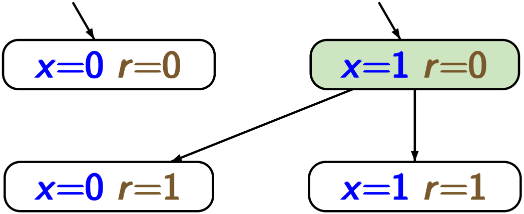
output function

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

transition function

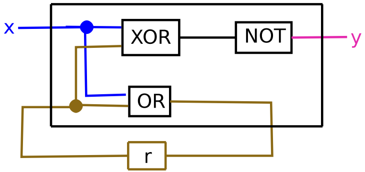
$\delta_r = x \vee r$

transition system



initial register evaluation: r=0

Example: TS for sequential circuit



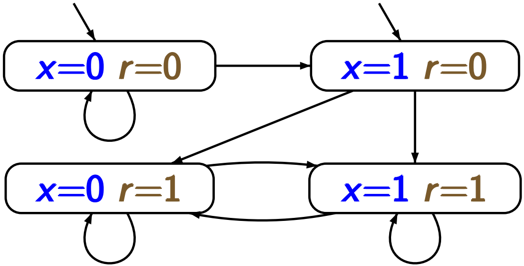
output function

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

transition function

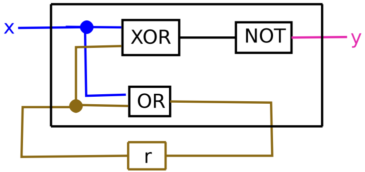
$\delta_r = x \vee r$

transition system



initial register evaluation: r=0

Example: TS for sequential circuit



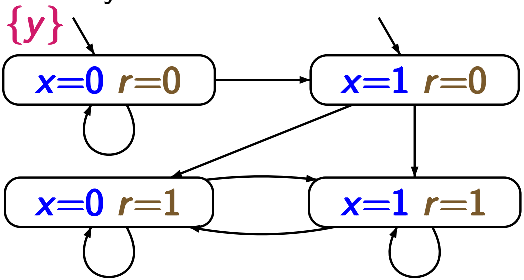
output function

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

transition function

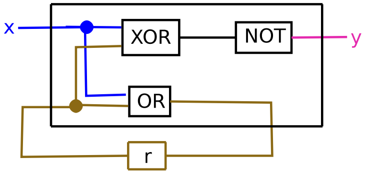
$\delta_r = x \vee r$

transition system



initial register evaluation: r=0

Example: TS for sequential circuit



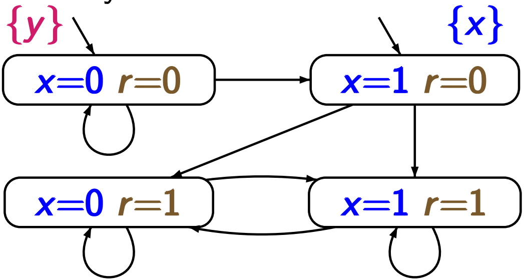
output function

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

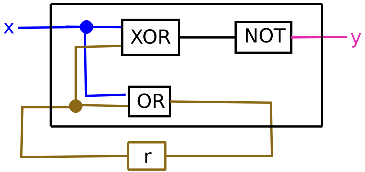
transition function

$\delta_r = x \vee r$

transition system



initial register evaluation: r=0



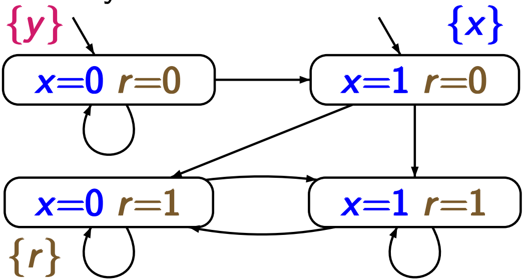
output function

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

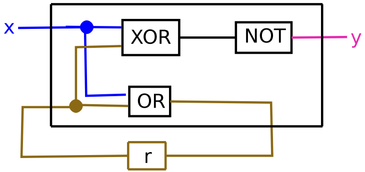
transition function

$\delta_r = x \vee r$

transition system



initial register evaluation: $r{=}0$



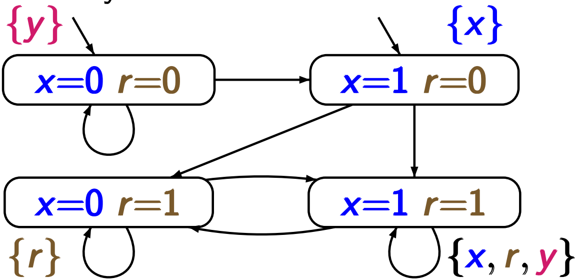
output function

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

transition function

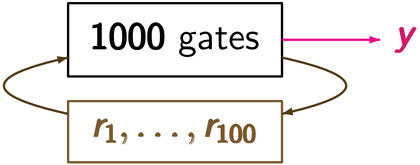
$\delta_r = x \vee r$

transition system



initial register evaluation: r=0

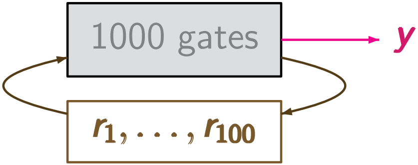
. . . has the transition system for a circuit of the form?



1 output bit
no input
100 registers

How many states . . .

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

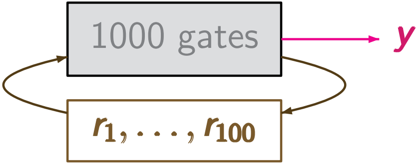


1 output bit
no input
100 registers

answer: 2^{100}

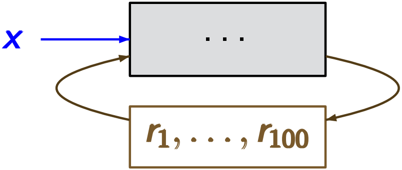
How many states ...

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



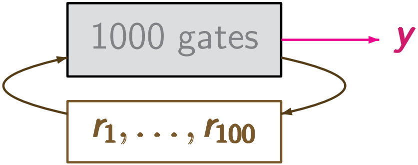
1 output bit
no input
100 registers

answer: 2^{100}



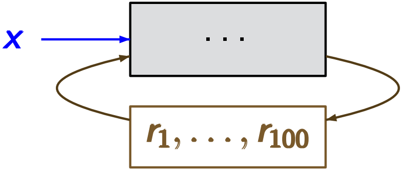
no output
1 input bit
100 registers

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



1 output bit
no input
100 registers

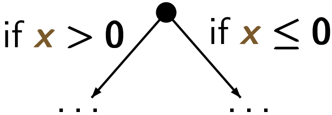
answer: 2^{100}



no output
1 input bit
100 registers

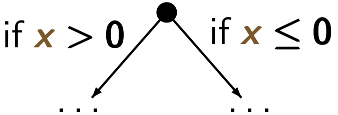
answer: $2^{100} * 2^1 = 2^{101}$

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



Data-dependent systems

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

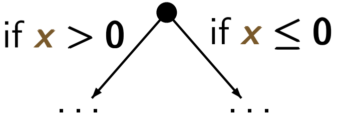


example: sequential program

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

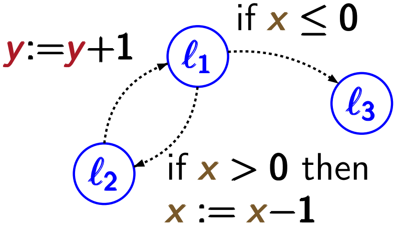
Data-dependent systems

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



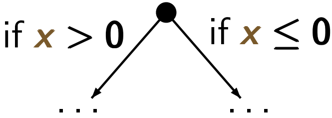
example: sequential program

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



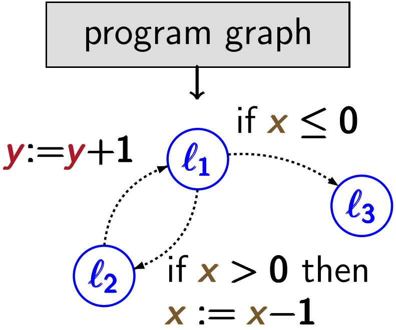
Data-dependent systems

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



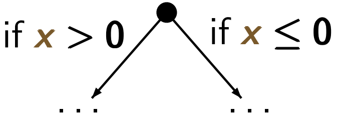
example: sequential program

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



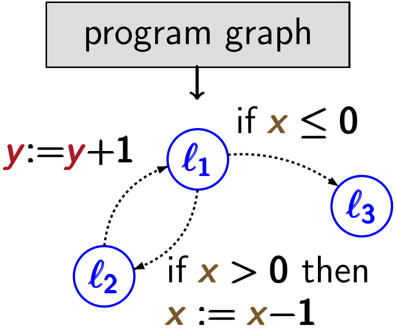
Data-dependent systems

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



example: sequential program

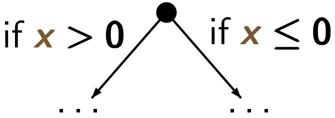
$\ell_1 \rightarrow$ `WHILE` $x > 0$ `DO`
$\quad\quad x := x-1;$
$\ell_2 \rightarrow$ $\quad\quad y := y+1$
`OD`
$\ell_3 \rightarrow$ ...



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

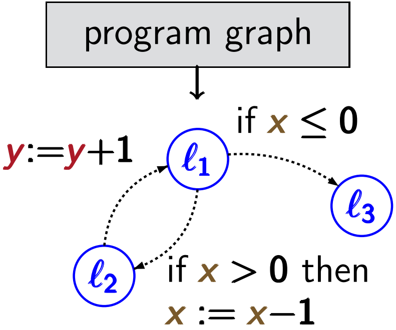
Data-dependent systems

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



example: sequential program

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



states of the transition system:

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

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

$\ell_1 \rightarrow$ `WHILE` $x > 0$ `DO`

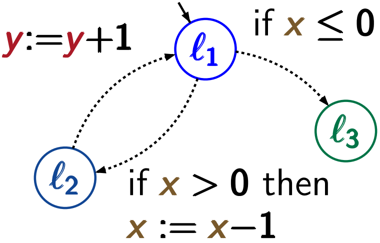
$\qquad x := x - 1$

$\ell_2 \rightarrow \qquad y := y + 1$

`OD`

$\ell_3 \rightarrow \ \ldots$

program graph



Example: TS for sequential program

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

$\ell_1 \rightarrow$ `WHILE` $x > 0$ `DO`

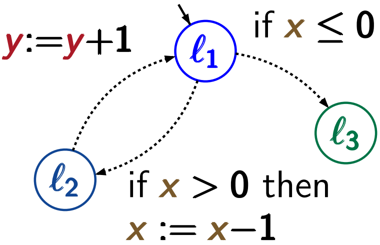
$\qquad\qquad x := x-1$

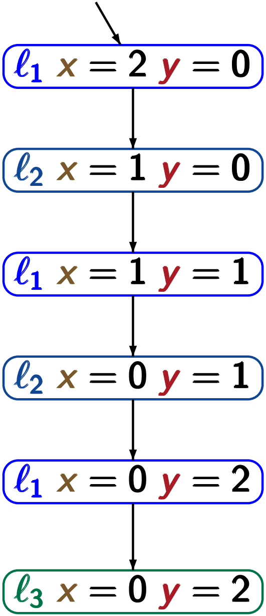
$\ell_2 \rightarrow \qquad y := y+1$

`OD`

$\ell_3 \rightarrow \quad \ldots$

program graph





Example: TS for sequential program

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

$\ell_1 \longrightarrow$ `WHILE` $x > 0$ `DO`

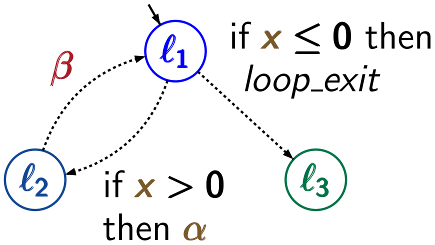
$x := x - 1 \quad \longleftarrow$ $\boxed{\text{action } \alpha}$

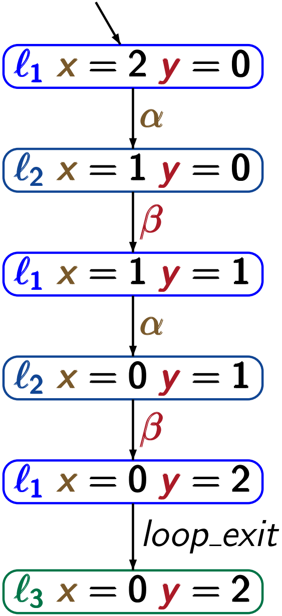
$\ell_2 \longrightarrow$ $y := y + 1 \quad \longleftarrow$ $\boxed{\text{action } \beta}$

`OD`

$\ell_3 \longrightarrow \ldots$

program graph





Typed variables

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

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 $Dom(z) = \{\mathbf{yellow}, \mathbf{red}, \mathbf{blue}\}$

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 $\eta : Var \longrightarrow Values$

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 $\eta : Var \to Values$

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

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

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 $\eta : Var \to Values$

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

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

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

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

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

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

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

Conditions on typed variables

If Var is a set of typed variables then

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

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

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:

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

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

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

$$\mathcal{P} \;=\; (Loc, Act, Effect, \hookrightarrow, Loc_0, g_0) \text{ where}$$

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

Program graph (PG)

Let Var be a set of typed variables.

A *program graph* over Var is a tuple

$$\mathcal{P} = (Loc, Act, Effect, \hookrightarrow, Loc_0, g_0) \text{ where}$$

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

function that formalizes the effect of the actions

Program graph (PG)

Let Var be a set of typed variables.

A *program graph* over Var is a tuple

$$\mathcal{P} = (Loc, Act, Effect, \hookrightarrow, Loc_0, g_0) \text{ where}$$

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

$\uparrow$

function that formalizes the effect of the actions

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

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

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,
- $\mathit{Effect} : \mathit{Act} \times \mathit{Eval}(\mathit{Var}) \rightarrow \mathit{Eval}(\mathit{Var})$
- $\hookrightarrow \; \subseteq \; \mathit{Loc} \times \mathit{Cond}(\mathit{Var}) \times \mathit{Act} \times \mathit{Loc}$

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 $\ell \xrightarrow{g:\alpha} \ell'$

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,
- $\mathit{Effect} : \mathit{Act} \times \mathit{Eval}(\mathit{Var}) \to \mathit{Eval}(\mathit{Var})$
- $\hookrightarrow \subseteq \mathit{Loc} \times \mathit{Cond}(\mathit{Var}) \times \mathit{Act} \times \mathit{Loc}$

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

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 $\ell \xrightarrow{g:\alpha} \ell'$
- $\mathit{Loc}_0 \subseteq \mathit{Loc}$ 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

$$\mathcal{P} = (Loc, Act, Effect, \hookrightarrow, Loc_0, g_0) \text{ where}$$

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

  specifies conditional transitions of the form $\ell \xhookrightarrow{g:\alpha} \ell'$
- $Loc_0 \subseteq Loc$ is the set of initial locations,
- $g_0 \in Cond(Var)$ initial condition on the variables.

TS-semantics of a program graph

TS-semantics of a program graph

program graph $\mathcal{P}$ over Var

$$\Downarrow$$

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

TS-semantics of a program graph

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

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

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

location variable evaluation

TS-semantics of a program graph

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

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

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

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

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

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

Structured operational semantics (SOS)

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

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

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

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

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

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

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

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: $S_0 = \{\langle \ell, \eta \rangle : \ell \in Loc_0, \eta \models g_0\}$
- $\longrightarrow$ 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:

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

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

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:

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

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

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

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:

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

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

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

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 $g \Rightarrow$ ***stmt***

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

Guarded Command Language (GCL)

guarded command $g \Rightarrow \mathit{stmt}$ $\longleftarrow$ $\boxed{\text{enabled if } g \text{ is true}}$

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

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

Guarded Command Language (GCL)

guarded command $g \Rightarrow \textit{stmt}$ $\longleftarrow$ 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 $\longleftarrow$ 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` :: $g \Rightarrow \mathit{stmt}_1$

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

`FI`

Guarded Command Language (GCL)

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

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

repetitive command/loop:

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

conditional command:

IF :: $g \Rightarrow \textit{stmt}_1$
:: $\neg g \Rightarrow \textit{stmt}_2$
FI

$\longleftarrow$

IF g THEN $\textit{stmt}_1$
ELSE $\textit{stmt}_2$
FI

Guarded Command Language (GCL)

guarded command $g \Rightarrow stmt$ $\longleftarrow$ $\boxed{\text{enabled if } g \text{ is true}}$

repetitive command/loop:

$\mathtt{DO}\ :: g \Rightarrow stmt\ \mathtt{OD}$ $\longleftarrow$ $\boxed{\mathtt{WHILE}\ g\ \mathtt{DO}\ stmt\ \mathtt{OD}}$

conditional command:

$$\begin{array}{lll} \mathtt{IF} & :: & g \Rightarrow stmt_1 \\ & :: & \neg g \Rightarrow stmt_2 \\ \mathtt{FI} & & \end{array} \longleftarrow \boxed{\begin{array}{lll} \mathtt{IF}\ g & \mathtt{THEN} & stmt_1 \\ & \mathtt{ELSE} & stmt_2 \\ \mathtt{FI} & & \end{array}}$$

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

Guarded Command Language (GCL)

modeling language with nondeterministic choice

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

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

Guarded Command Language (GCL)

modeling language with nondeterministic choice

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

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

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

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)	

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

         :: #coke > 0 ⇒ #coke := #coke − 1

         :: #sprite > 0 ⇒ #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

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

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

FI

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

OD

GCL-program for beverage machine

$\mathtt{DO} \;::\; \text{true} \Rightarrow \text{insert\_coin}$; (* user inserts a coin *)

$\mathtt{IF} \;::\; \#sprite = \#coke = 0 \Rightarrow \text{return\_coin}$

(* no beverage available *)

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

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

$\mathtt{FI}$

$:: \;\text{true} \Rightarrow \#sprite := max; \#coke := max$

$\mathtt{OD}$

GCL-program for beverage machine


```
DO :: true ⇒ insert_coin; (* user inserts a coin *)
      IF :: #sprite = #coke = 0 ⇒ return_coin
                                    (* no beverage available *)
         :: #coke > 0 ⇒ #coke := #coke − 1
                                    (* user selects coke *)
         :: #sprite > 0 ⇒ #sprite := #sprite−1
                                    (* user selects sprite *)
      FI
   :: true ⇒ #sprite := max; #coke := 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

$$
\begin{array}{lll}
\texttt{DO} & :: & \text{true} \Rightarrow \text{insert\_coin}; \\
& & \texttt{IF} \ :: \ \#\textit{sprite} = \#\textit{coke} = 0 \Rightarrow \text{return\_coin} \\
& & \phantom{\texttt{IF}} \ :: \ \#\textit{coke} > 0 \Rightarrow \text{get\_coke} \\
& & \phantom{\texttt{IF}} \ :: \ \#\textit{sprite} > 0 \Rightarrow \text{get\_sprite} \\
& & \texttt{FI} \\
& :: & \text{true} \Rightarrow \text{refill} \\
\texttt{OD} & &
\end{array}
$$

```
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

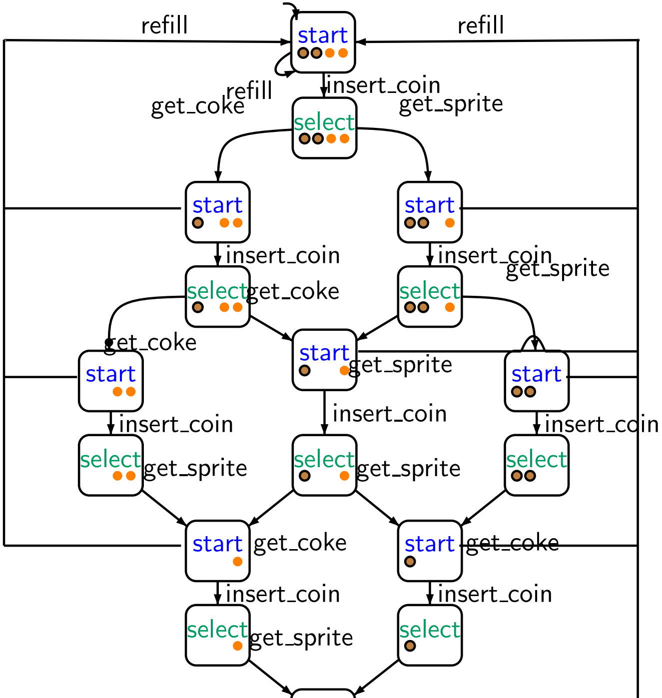
- two variables $\#sprite, \#coke \in \{0, 1, \ldots, max\}$

GCL-program for beverage machine

$$
\begin{array}{lll}
\textit{start} \longrightarrow \texttt{DO} & :: & \text{true} \Rightarrow \text{insert\_coin}; \\
\textit{select} \longrightarrow & & \texttt{IF} \; :: \; \#\textit{sprite} = \#\textit{coke} = 0 \\
& & \qquad\qquad \Rightarrow \text{return\_coin} \\
& & \qquad :: \; \#\textit{coke} \; > 0 \Rightarrow \text{get\_coke} \\
& & \qquad :: \; \#\textit{sprite} > 0 \Rightarrow \text{get\_sprite} \\
& & \texttt{FI} \\
& :: & \text{true} \Rightarrow \text{refill} \\
\quad \texttt{OD} & &
\end{array}
$$

... yields a program graph with

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


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

Summary

- We defined transition system
- We showed how to transform a sequential circuit into a transition system.
- We showed how to transform a program in Dijkstra's guarded command language into a program graph.
- We showed how to convert a program graph into a transition system.