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
- Semantic model
- Implementation refinement
Transition systems

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**
The semantic model yields a formal representation of:

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

control component + information on “relevant” data
Transition systems $\cong$ extended digraphs

The semantic model yields a formal representation of:

- the **states** of the system $\leftarrow$ nodes
- the **stepwise behaviour** $\leftarrow$ edges
- the **initial states**

**real system**

**semantic model**

- semantics abstraction
- implementation refinement

control component $\oplus$ information on “relevant” data
The semantic model yields a formal representation of:

- the **states** of the system \(\rightarrow\) **nodes**
- the **stepwise behaviour** \(\rightarrow\) **transitions**
- the **initial states**
Transition systems $\cong$ extended digraphs

The semantic model yields a formal representation of:

- the **states** of the system $\leftarrow$ **nodes**
- the **stepwise behaviour** $\leftarrow$ **transitions**
- the **initial states**
- additional information on **communication**
- **state properties**
Transition systems $\cong$ extended digraphs

The semantic model yields a formal representation of:

- **the states** of the system $\leftarrow$ **nodes**
- **the stepwise behaviour** $\leftarrow$ **transitions**
- **the initial states**
- **additional information on** communication $\leftarrow$ **actions**
  - state properties $\leftarrow$ **atomic proposition**
A transition system is a tuple

\[ \mathcal{T} = (S, Act, \rightarrow, S_0, AP, L) \]
Transition system (TS)

A transition system is a tuple

\[ \mathcal{T} = (S, \text{Act}, \rightarrow, S_0, AP, L) \]

- \( S \) is the state space, i.e., set of states,
A transition system is a tuple

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

• $S$ is the state space, i.e., set of states,
• $Act$ is a set of actions,
A transition system is a tuple

\[ \mathcal{T} = (S, Act, \rightarrow, S_0, AP, L) \]

- \( S \) is the state space, i.e., set of states,
- \( Act \) is a set of actions,
- \( \rightarrow \subseteq S \times Act \times S \) is the transition relation,
A transition system is a tuple

\[ \mathcal{T} = (S, Act, \rightarrow, S_0, AP, L) \]

- \( S \) is the state space, i.e., set of states,
- \( Act \) is a set of actions,
- \( \rightarrow \subseteq S \times Act \times S \) 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

\[ \mathcal{T} = (S, \text{Act}, \rightarrow, S_0, \text{AP}, L) \]

- \( S \) is the state space, i.e., set of states,
- \( \text{Act} \) is a set of actions,
- \( \rightarrow \subseteq S \times \text{Act} \times S \) is the transition relation,
- i.e., transitions have the form \( s \xrightarrow{\alpha} s' \) where \( s, s' \in S \) and \( \alpha \in \text{Act} \)
- \( S_0 \subseteq S \) the set of initial states,
A transition system is a tuple

\[ \mathcal{T} = (S, \text{Act}, \rightarrow, S_0, \text{AP}, L) \]

- \( S \) is the state space, i.e., set of states,
- \( \text{Act} \) is a set of actions,
- \( \rightarrow \subseteq S \times \text{Act} \times S \) is the transition relation,
- \( S_0 \subseteq S \) the set of initial states,
- \( \text{AP} \) a set of atomic propositions,
- \( L : S \rightarrow 2^{\text{AP}} \) the labeling function

i.e., transitions have the form \( s \xrightarrow{\alpha} s' \) where \( s, s' \in S \) and \( \alpha \in \text{Act} \)
Transition system for beverage machine
Transition system for beverage machine

State space \( S = \{ \text{pay, select, coke, sprite} \} \)

Set of initial states: \( S_0 = \{ \text{pay} \} \)
Transition system for beverage machine

state space $S = \{ \text{pay, select, coke, sprite} \}$
set of initial states: $S_0 = \{ \text{pay} \}$
Transition system for beverage machine

state space \( S = \{ \text{pay}, \text{select}, \text{coke}, \text{sprite} \} \)

set of initial states: \( S_0 = \{ \text{pay} \} \)

set of atomic propositions: \( AP = \{ \text{pay}, \text{drink} \} \)

labeling function:
\[
L(\text{coke}) = L(\text{sprite}) = \{ \text{drink} \} \\
L(\text{pay}) = \{ \text{pay} \}, \quad L(\text{select}) = \emptyset
\]
Transition system for beverage machine

state space \( S = \{\text{pay}, \text{select}, \text{coke}, \text{sprite}\} \)

set of initial states: \( S_0 = \{\text{pay}\} \)

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
possible behaviours of a TS result from:

```plaintext
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"

\[
\begin{align*}
    s_0 & \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} s_2 \xrightarrow{\alpha_3} \ldots \text{ with } s_0 \in S_0
\end{align*}
\]
“Behaviour” of transition systems

possible behaviours of a TS result from:

select nondeterministically an initial state \( s \in S_0 \)

\[
\text{WHILE } s \text{ is non-terminal DO}
\]

select nondeterministically a transition \( s \xrightarrow{\alpha} s' \)

execute the \textit{action} \( \alpha \) and put \( s := s' \)

\[
\text{OD}
\]

\textit{executions:} maximal “transition sequences”

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

\textit{reachable fragment:}

\[
\text{Reach}(T) = \text{ set of all states that are reachable from an initial state through some execution}
\]
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. \[ x := x + 1 \quad || \quad y := y - 3 \]

\( \alpha \), \( \beta \) independent

parallel execution of dependent actions
Transition system for parallel actions

parallel execution of independent actions

\[ x := x + 1 \ ||\ || \ y := y - 3 \]

\[ \text{action } \alpha \ ||\ || \ \text{action } \beta \]

\(\alpha, \beta\) independent

parallel execution of dependent actions

\[ x := x + 1 \ ||\ || \ y := 2 \times x \]

\[ \text{action } \alpha \ ||\ || \ \text{action } \beta \]

\(\alpha, \beta\) dependent
### Transition system for parallel actions

**Parallel execution of independent actions** ← *interleaving*

- Example:
  \[
  \begin{align*}
    &x \leftarrow x + 1 \quad || \quad y \leftarrow y - 3 \\
    &\text{action } \alpha \quad || \quad \text{action } \beta
  \end{align*}
  \]

**Parallel execution of dependent actions** ← *competition*

- Example:
  \[
  \begin{align*}
    &x \leftarrow x + 1 \quad || \quad y \leftarrow 2 \times x \\
    &\text{action } \alpha \quad || \quad \text{action } \beta
  \end{align*}
  \]
parallel execution of independent actions

\[
\begin{align*}
\alpha: & \quad x = 1, \quad y = 5 \\
\beta: & \quad x = 0, \quad y = 2 \\
\end{align*}
\]

action \( \alpha \)

\[
\begin{align*}
\alpha: & \quad x = 1, \quad y = 2 \\
\beta: & \quad x = 0, \quad y = 2 \\
\end{align*}
\]

action \( \beta \)

\[
\begin{align*}
x & := x + 1 \\
y & := y - 3
\end{align*}
\]
parallel execution of independent actions ← interleaving

parallel execution of dependent actions ← competition

\[
x = 1 \\
y = 5
\]

\[
x = 0 \\
y = 5
\]

\[
x = 0 \\
y = 2
\]

\[
x = 1 \\
y = 2
\]

\[
x := x + 1 \quad \text{action } \alpha
\]

\[
y := y - 3 \quad \text{action } \beta
\]
parallel execution of independent actions ← interleaving

parallel execution of dependent actions ← competition
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:

- Generate message
- Send fax
- Send email
Implementation freedom

Realization by a TS:

- **sender**
- **unknown receiver**

- *fax* connection
- *email* connection

**At a future refinement step the nondeterminism is replaced with one of the alternatives**
Implementation freedom

Realization by a TS:

- Generate message
- ... (other alternatives)
- Send fax
- Send email

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

produce message

try to send

lost
delivered
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

menu  name

1  2  3
4  5  6
7  8  9
0  

off

on

0  1  ...  9

...  ...  ...  ...  ...  menu
Incomplete information on the environment

mobile phone

resolution of the nondeterministic choices by a human user
Possible meanings of nondeterminism in TS

concurrency (interleaving)

\[ \alpha \parallel \parallel \beta \] 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
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 \( P_1 \parallel \ldots \parallel P_n \)

requirements

transition system \( \mathcal{T} \)

specification \( spec \)

model checker

does \( \mathcal{T} \) satisfy \( spec \)?

yes

no + error indication
Model checking

system $P_1 \parallel \ldots \parallel P_n$

requirements

transition system $\mathcal{T}$

model checker

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

yes

no + error indication

specification $\text{spec}$
Model checking

syntactic description of $P_1 \parallel \ldots \parallel P_n$

requirements

specification $spec$

transition system $\mathcal{T}$

model checker

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

semantics

yes

no + error indication

yes no +++ error indication
Modelling of sequential circuits by TS

input bits $x_1, \ldots, x_n$ → circuit → register $r_1, \ldots, r_k$ → output bits $y_1, \ldots, y_m$
Modelling of sequential circuits by TS

input bits \( x_1, \ldots, x_n \) \( \to \) circuit \( \to \) output functions \( \lambda_1, \ldots, \lambda_m \)

register \( r_1, \ldots, r_k \)

transition functions \( \delta_1, \ldots, \delta_k \)
Modelling of sequential circuits by TS

**input bits** $x_1, \ldots, x_n$ → **circuit** → **output functions** $y_1, \ldots, y_m$

**register** $r_1, \ldots, r_k$

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

$\delta_j, \lambda_i \equiv$ switching functions $\{0, 1\}^n \times \{0, 1\}^k \rightarrow \{0, 1\}$
Modelling of sequential circuits by TS

input bits $x_1, \ldots, x_n$ → circuit → output functions $y_1, \ldots, y_m$

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

register $r_1, \ldots, r_k$

$\delta_j, \lambda_i \equiv$ switching functions $\{0, 1\}^n \times \{0, 1\}^k \rightarrow \{0, 1\}$

input values $a_1, \ldots, a_n$

for the input variables

$\uparrow$ current values $c_1, \ldots, c_k$

of the registers

output value $\lambda_i(\ldots)$

for output variable $y_i$

$\rightarrow$ next value $\delta_j(\ldots)$

for register $r_j$
Modelling of sequential circuits by TS

Input bits $x_1, \ldots, x_n$ → Circuit → Output functions $y_1, \ldots, y_m$

Register $r_1, \ldots, r_k$

Transition functions $\delta_1, \ldots, \delta_k$

Initial register evaluation $[r_1 = c_{01}, \ldots, r_k = c_{0k}]$
Modelling of sequential circuits by TS

Input bits \( x_1, \ldots, x_n \) → circuit → output functions \( y_1, \ldots, y_m \)

Register \( r_1, \ldots, r_k \)

Transition functions \( \delta_1, \ldots, \delta_k \)

Initial register evaluation \([r_1 = c_{01}, \ldots, r_k = c_{0k}]\)

Transition system:
- States: evaluations of \( x_1, \ldots, x_n, r_1, \ldots, r_k \)
Modelling of sequential circuits by TS

input bits \( x_1, \ldots, x_n \) → circuit → output functions \( \lambda_1, \ldots, \lambda_m \)

memory cells \( r_1, \ldots, r_k \)

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

transition functions \( \delta_1, \ldots, \delta_k \)
Modelling of sequential circuits by TS

input bits $x_1, \ldots, x_n$ \rightarrow circuit \rightarrow output functions $y_1, \ldots, y_m$

register $r_1, \ldots, r_k$

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

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
- values of input bits change nondeterministically
Modelling of sequential circuits by TS

input bits \( x_1, \ldots, x_n \) → circuit → output functions \( \lambda_1, \ldots, \lambda_m \)

- register \( r_1, \ldots, r_k \)
- transition functions \( \delta_1, \ldots, \delta_k \)
- 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
- 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: \( \delta_r = x \lor r \)
Example: TS for sequential circuit

output function
\[ \lambda_y = \neg(x \oplus r) \]

transition function
\[ \delta_r = x \lor r \]
Example: TS for sequential circuit

Output function
\[ \lambda_y = \neg(x \oplus r) \]

Transition function
\[ \delta_r = x \lor r \]

Transition system
Example: TS for sequential circuit

output function
\[ \lambda_y = \neg (x \oplus r) \]

transition function
\[ \delta_r = x \lor r \]

transition system

<table>
<thead>
<tr>
<th>x</th>
<th>r</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Example: TS for sequential circuit

output function
\[ \lambda_y = \neg(x \oplus r) \]
transition function
\[ \delta_r = x \lor r \]

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

![Diagram of a sequential circuit with XOR, NOT, and OR gates.]

**Output function**

\[ \lambda_y = \neg(x \oplus r) \]

**Transition function**

\[ \delta_r = x \lor r \]

**Transition system**

- \( x=0 \), \( r=0 \) → \( x=1 \), \( r=0 \)
- \( x=0 \), \( r=1 \) → \( x=0 \), \( r=1 \)
- \( x=1 \), \( r=0 \) → \( x=1 \), \( r=1 \)

**Initial register evaluation:** \( r=0 \)
Example: TS for sequential circuit

Transition system:

\[ \begin{align*}
\text{output function} & \quad \lambda_y = \neg(x \oplus r) \\
\text{transition function} & \quad \delta_r = x \lor r
\end{align*} \]

Initial register evaluation: \( r=0 \)
Example: TS for sequential circuit

output function
\[ \lambda_y = \neg(x \oplus r) \]

transition function
\[ \delta_r = x \lor r \]

transition system

initial register evaluation: \( r=0 \)
Example: TS for sequential circuit

Output function:
\[ \lambda_y = \neg (x \oplus r) \]

Transition function:
\[ \delta_r = x \lor r \]

Transition system:

\[
\begin{align*}
\{y\} & \quad x=0, r=0 \\
& \quad x=1, r=0 \\
& \quad x=0, r=1 \\
& \quad x=1, r=1 \\
\end{align*}
\]

Initial register evaluation: \( r=0 \)
Example: TS for sequential circuit

output function
\[ \lambda_y = \neg(x \oplus r) \]

transition function
\[ \delta_r = x \lor r \]

transition system

initial register evaluation: \( r = 0 \)
Example: TS for sequential circuit

Output function:
\[ \lambda_y = \neg(x \oplus r) \]

Transition function:
\[ \delta_r = x \lor r \]

Transition system:

Initial register evaluation: \( r = 0 \)
Example: TS for sequential circuit

output function
\[ \lambda_y = \neg(x \oplus r) \]

transition function
\[ \delta_r = x \lor r \]

transition system

\[ \{y\} \]
\[ x=0 \ r=0 \]
\[ \{r\} \]
\[ x=0 \ r=1 \]
\[ \{x, r, y\} \]
\[ x=1 \ r=0 \]
\[ x=1 \ r=1 \]
\[ \{x\} \]

initial register evaluation: \[ r=0 \]
How many states...

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

1000 gates

$y$

$r_1, \ldots, r_{100}$

1 output bit
no input
100 registers
How many states . . .

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

1000 gates

\[ r_1, \ldots, r_{100} \]

\( y \)

1 output bit
no input
100 registers

answer: \( 2^{100} \)
How many states . . .

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

answer: $2^{100}$
How many states . . .

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

\[ \begin{align*}
1000 \text{ gates} & \quad \rightarrow \quad y \\
r_1, \ldots, r_{100} & \quad \rightarrow \quad \ldots \\
\rightarrow \quad x & \quad \rightarrow \quad \ldots \\
r_1, \ldots, r_{100} & \quad \rightarrow 
\end{align*} \]

1 output bit
no input
100 registers

answer: \(2^{100}\)

1 input bit
no output
100 registers

answer: \(2^{100} \times 2^1 = 2^{101}\)
Data-dependent systems

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

\[
\text{if } x > 0 \quad \text{if } x \leq 0
\]

\[
\ldots \quad \ldots
\]
Data-dependent systems

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

\[
\begin{align*}
\text{if } x > 0 & \quad \text{if } x \leq 0 \\
\ldots & \quad \ldots
\end{align*}
\]

**example:** sequential program

\[
\begin{align*}
\text{WHILE } x > 0 \text{ DO} \\
\quad x & := x - 1; \\
\quad y & := y + 1 \\
\text{OD} \\
\ldots
\end{align*}
\]
Data-dependent systems

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

\[
\begin{align*}
\text{if } x > 0 & \quad \text{if } x \leq 0 \\
\text{...} & \quad \text{...}
\end{align*}
\]

**Example:** Sequential program

\[
\begin{align*}
\text{WHILE } x > 0 \text{ DO} \\
& \quad x := x - 1; \\
& \quad y := y + 1 \\
\text{OD} \\
\text{...}
\end{align*}
\]

\[
\begin{align*}
& \quad \text{if } x \leq 0 \quad \text{if } x > 0 \text{ then} \\
& \quad y := y + 1 \quad x := x - 1 \\
\end{align*}
\]
Data-dependent systems

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

if \( x > 0 \) \( \rightarrow \) if \( x \leq 0 \)

... ... ...

e**xample:** sequential program

\[
\begin{align*}
\text{WHILE } & x > 0 \text{ DO} \\
& x := x - 1; \\
& y := y + 1 \\
\text{OD} \\
& ... \\
\end{align*}
\]

program graph

\[
\begin{align*}
& l_1 \quad \text{if } x \leq 0 \quad y := y + 1 \\
& l_2 \quad \text{if } x > 0 \text{ then} \\
& l_3 \quad x := x - 1 \\
\end{align*}
\]
**Data-dependent systems**

*problem:* TS-representation of conditional branchings?

\[
\text{if } x > 0 \quad \text{if } x \leq 0
\]

\[
\ldots \quad \ldots
\]

*example:* sequential program

\[
\ell_1 \rightarrow \text{WHILE } x > 0 \text{ DO } \quad y := y + 1
\]

\[
x := x - 1;
\]

\[
\ell_2 \rightarrow \text{OD}
\]

\[
\ell_3 \rightarrow \ldots
\]

\[
\ell_1, \ell_2, \ell_3 \text{ are locations, i.e., control states}
\]
**Data-dependent systems**

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

\[ \text{if } x > 0 \quad \bullet \quad \text{if } x \leq 0 \]

\[ \ldots \quad \ldots \]

**Example:** Sequential program

\[ l_1 \rightarrow \text{WHILE } x > 0 \text{ DO} \]

\[ x := x - 1; \]

\[ l_2 \rightarrow \text{Od} \]

\[ y := y + 1 \]

\[ l_3 \rightarrow \ldots \]

**Program graph**

\[ \text{if } x \leq 0 \]

\[ y := y + 1 \]

\[ l_1 \quad \rightarrow \quad l_3 \]

\[ \text{if } x > 0 \quad \text{then} \]

\[ x := x - 1 \]

\[ l_2 \quad \rightarrow \quad l_1 \]

**States of the transition system:**

- **Locations + relevant data** *(here: values for x and y)*
Example: TS for sequential program

Initially: \( x = 2, y = 0 \)

\( \ell_1 \rightarrow \) WHILE \( x > 0 \) DO
  \( x := x - 1 \)
  \( \ell_2 \rightarrow \) DO
    \( y := y + 1 \)
    \( \ell_3 \rightarrow \) ...

program graph

\( y := y + 1 \) \( \ell_1 \) if \( x \leq 0 \)

\( \ell_2 \) if \( x > 0 \) then \( x := x - 1 \)

\( \ell_3 \)
Example: TS for sequential program

initially: $x = 2, y = 0$

$l_1 \rightarrow$ WHILE $x > 0$ DO
  $x := x - 1$

$l_2 \rightarrow$ y := y + 1

OD

$l_3 \rightarrow$ ...

program graph

$x := x - 1$ if $x \leq 0$

$l_1 \rightarrow$ y := y + 1 if $x > 0$ then

$l_2 \rightarrow$ if $x > 0$ then

$l_3 \rightarrow$ ...

$l_1 x = 2 \ y = 0$

$l_2 x = 1 \ y = 0$

$l_1 x = 1 \ y = 1$

$l_2 x = 0 \ y = 1$

$l_1 x = 0 \ y = 2$

$l_3 x = 0 \ y = 2$
Example: TS for sequential program

Initially: $x = 2, y = 0$

1 $\rightarrow$ WHILE $x > 0$ DO
   2 $\rightarrow$ action $\alpha$
   3 $\rightarrow$ OD
3 $\rightarrow$ ...

Program graph

If $x \leq 0$ then loop_exit

If $x > 0$ then $\alpha$
Typed variables

**typed variable**: variable $x$ + data domain $\text{Dom}(x)$
Typed variables

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

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

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

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

**evaluation** for a set $\text{Var}$ of typed variables:

*type-consistent function* $\eta : \text{Var} \rightarrow \text{Values}$
Typed variables

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

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

**evaluation** for a set $\text{Var}$ of typed variables:

- type-consistent function $\eta : \text{Var} \rightarrow \text{Values}$

\[\eta(x) \in \text{Dom}(x)\]

for all $x \in \text{Var}$

\[\text{Values} = \bigcup_{x \in \text{Var}} \text{Dom}(x)\]
Typed variables

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

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

evaluation for a set $\text{Var}$ of typed variables:

type-consistent function $\eta : \text{Var} \rightarrow \text{Values}$

$$\eta(x) \in \text{Dom}(x)$$

for all $x \in \text{Var}$

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

Notation: $\text{Eval} (\text{Var}) =$ set of evaluations for $\text{Var}$
Conditions on typed variables

If $\textbf{Var}$ is a set of typed variables then

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

If \( \textbf{Var} \) is a set of typed variables then

\[
\text{Cond}(\textbf{Var}) = \text{set of Boolean conditions on the variables in } \textbf{Var}
\]

Example: \((\neg x \land y < z + 3) \lor w = \text{red}\)

where \(\text{Dom}(x) = \{0, 1\}, \text{Dom}(y) = \text{Dom}(z) = \mathbb{N},\)
\(\text{Dom}(w) = \{\text{yellow, red, blue}\}\)
Conditions on typed variables

If \( \text{Var} \) is a set of typed variables then

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

Example: \( (\neg x \land y < z + 3) \lor w = \text{red} \)

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

satisfaction relation \( \models \) for evaluations and conditions
Conditions on typed variables

If \textbf{Var} is a set of typed variables then

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

Example: \((\neg x \land y < z + 3) \lor w = \text{red}\)

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

\textit{satisfaction relation} \models \text{for evaluations and conditions}

Example:

\([x=0, y=3, z=6] \models \neg x \land y < z\)

\([x=0, y=3, z=6] \not\models x \lor y = z\)
Effect-function for actions

Given a set $\text{Act}$ of actions that operate on the variables in $\text{Var}$, the effect of the actions is formalized by:
Effect-function for actions

Given a set $\text{Act}$ of actions that operate on the variables in $\text{Var}$, the effect of the actions is formalized by:

$$\text{Effect} : \text{Act} \times \text{Eval}(\text{Var}) \rightarrow \text{Eval}(\text{Var})$$
Effect-function for actions

Given a set $\text{Act}$ of actions that operate on the variables in $\text{Var}$, the effect of the actions is formalized by:

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

if $\alpha$ is “$x:=2x+y$” then:

$$\text{Effect}(\alpha, [x=1, y=3, \ldots]) = [x=5, y=3, \ldots]$$
Effect-function for actions

Given a set $\text{Act}$ of actions that operate on the variables in $\text{Var}$, the effect of the actions is formalized by:

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

if $\alpha$ is “$x:=2x+y$” then:

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

if $\beta$ is “$x:=2x+y$; $y:=1-x$” then:

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

Given a set $\textit{Act}$ of actions that operate on the variables in $\textit{Var}$, the effect of the actions is formalized by:

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

If $\alpha$ is "$x:=2x+y$" then:

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

If $\beta$ is "$x:=2x+y; y:=1-x$" then:

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

If $\gamma$ is "$(x, y):=(2x+y, 1-x)$" then:

$$\textit{Effect}(\gamma, [x=1, y=3, \ldots]) = [x=5, y=0, \ldots]$$
Program graph (PG)
Let $\textbf{Var}$ be a set of typed variables.

A program graph over $\textbf{Var}$ is a tuple

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

where
Let $\mathbf{Var}$ be a set of typed variables.

A program graph over $\mathbf{Var}$ is a tuple

$$\mathcal{P} = (\mathbf{Loc}, \mathbf{Act}, \mathbf{Effect}, \rightarrow, \mathbf{Loc}_0, g_0)$$

where

- $\mathbf{Loc}$ is a (finite) set of locations, i.e., control states,
Let \( \text{Var} \) be a set of typed variables.

A program graph over \( \text{Var} \) is a tuple

\[
\mathcal{P} = (\text{Loc}, \text{Act}, \text{Effect}, \rightarrow, \text{Loc}_0, g_0)
\]

where

- \( \text{Loc} \) is a (finite) set of locations, i.e., control states,
- \( \text{Act} \) a set of actions,
Let $\text{Var}$ be a set of typed variables.

A program graph over $\text{Var}$ is a tuple

$$P = (\text{Loc}, \text{Act}, \text{Effect}, \rightarrow, \text{Loc}_0, g_0)$$

where

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

A program graph over $\mathbf{Var}$ is a tuple

$$\mathcal{P} = (\mathbf{Loc}, \mathbf{Act}, \mathbf{Effect}, \rightarrow, \mathbf{Loc}_0, g_0)$$

where

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

function that formalizes the effect of the actions
Let \( \textbf{Var} \) be a set of typed variables.

A program graph over \( \textbf{Var} \) is a tuple

\[
\mathcal{P} = (\textbf{Loc}, \textbf{Act}, \text{Effect}, \rightarrow, \textbf{Loc}_0, g_0)
\]

where

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

function that formalizes the effect of the actions

example: if \( \alpha \) is the assignment \( x:=x+y \) then

\[
\text{Effect}(\alpha, [x=1, y=7]) = [x=8, y=7]
\]
Let $\text{Var}$ be a set of typed variables.

A program graph over $\text{Var}$ is a tuple

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

- $\text{Loc}$ is a (finite) set of locations, i.e., control states,
- $\text{Act}$ a set of actions,
- $\text{Effect} : \text{Act} \times \text{Eval(Var)} \rightarrow \text{Eval(Var)}$
- $\rightarrow \subseteq \text{Loc} \times \text{Cond(Var)} \times \text{Act} \times \text{Loc}$
Let \( \text{Var} \) be a set of typed variables.

A program graph over \( \text{Var} \) is a tuple

\[
\mathcal{P} = (\text{Loc}, \text{Act}, \text{Effect}, \rightarrow, \text{Loc}_0, g_0)
\]

where

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

specifies conditional transitions of the form \( \ell \xrightarrow{g : \alpha} \ell' \)

\( \ell, \ell' \) are locations, \( g \in \text{Cond}(\text{Var}) \), \( \alpha \in \text{Act} \)
Program graph (PG)

Let \( \text{Var} \) be a set of typed variables.

A program graph over \( \text{Var} \) is a tuple

\[
\mathcal{P} = (\text{Loc}, \text{Act}, \text{Effect}, \rightarrow, \text{Loc}_0, g_0)
\]

where

- \( \text{Loc} \) is a (finite) set of locations, i.e., control states,
- \( \text{Act} \) a set of actions,
- \( \text{Effect} : \text{Act} \times \text{Eval}(\text{Var}) \rightarrow \text{Eval}(\text{Var}) \)
- \( \rightarrow \subseteq \text{Loc} \times \text{Cond}(\text{Var}) \times \text{Act} \times \text{Loc} \)
  specifies conditional transitions of the form \( \ell \xrightarrow{g: \alpha} \ell' \)
- \( \text{Loc}_0 \subseteq \text{Loc} \) is the set of initial locations,
Program graph (PG)

Let $\textbf{Var}$ be a set of typed variables.

A program graph over $\textbf{Var}$ is a tuple

\[ \mathcal{P} = (\text{Loc}, \text{Act}, \text{Effect}, \rightarrow, \text{Loc}_0, g_0) \]

where

- $\text{Loc}$ is a (finite) set of locations, i.e., control states,
- $\text{Act}$ a set of actions,
- $\text{Effect} : \text{Act} \times \text{Eval(Var)} \rightarrow \text{Eval(Var)}$
- $\rightarrow \subseteq \text{Loc} \times \text{Cond(Var)} \times \text{Act} \times \text{Loc}$ specifies conditional transitions of the form $\ell \xrightarrow{g:\alpha} \ell'$
- $\text{Loc}_0 \subseteq \text{Loc}$ is the set of initial locations,
- $g_0 \in \text{Cond(Var)}$ initial condition on the variables
Let $\textbf{Var}$ be a set of typed variables.

A program graph over $\textbf{Var}$ is a tuple

$$\mathcal{P} = (\textbf{Loc}, \textbf{Act}, \textbf{Effect}, \rightarrow, \textbf{Loc}_0, g_0)$$

where

- $\textbf{Loc}$ is a (finite) set of locations, i.e., control states,
- $\textbf{Act}$ a set of actions,
- $\textbf{Effect}: \textbf{Act} \times \text{Eval}(\textbf{Var}) \rightarrow \text{Eval}(\textbf{Var})$
- $\rightarrow \subseteq \textbf{Loc} \times \text{Cond}(\textbf{Var}) \times \textbf{Act} \times \textbf{Loc}$
- specifies conditional transitions of the form $\ell \xleftarrow{\alpha} g: \rightarrow \ell'$
- $\textbf{Loc}_0 \subseteq \textbf{Loc}$ is the set of initial locations,
- $g_0 \in \text{Cond}(\textbf{Var})$ initial condition on the variables.
TS-semantics of a program graph
TS-semantics of a program graph

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

$\Downarrow$

transition system $\mathcal{T}_\mathcal{P}$
TS-semantics of a program graph

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

transition system $\mathcal{I}_P$

states in $\mathcal{I}_P$ have the form

$\langle l, \eta \rangle$

location variable evaluation
Let $\mathcal{P} = (\text{Loc}, \text{Act}, \text{Effect}, \rightarrow, \text{Loc}_0, g_0)$ be a PG.

The transition system of $\mathcal{P}$ is:

$$\mathcal{T}_\mathcal{P} = (S, \text{Act}, \rightarrow, S_0, AP, L)$$
TS-semantics of a program graph

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

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

- state space: $S = \text{Loc} \times \text{Eval}(\text{Var})$
TS-semantics of a program graph

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

\[
\mathcal{T}_\mathcal{P} = (S, \text{Act}, \rightarrow, S_0, \text{AP}, L)
\]

- state space: \( S = \text{Loc} \times \text{Eval(Var)} \)
- initial states: \( S_0 = \{ \langle l, \eta \rangle : l \in \text{Loc}_0, \eta \models g_0 \} \)
TS-semantics of a program graph

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

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

- state space: $S = \text{Loc} \times \text{Eval(Var)}$
- initial states: $S_0 = \{ \langle \ell, \eta \rangle : \ell \in \text{Loc}_0, \eta \models g_0 \}$

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

$$\ell \xrightarrow{g; \alpha} \ell' \land \eta \models g \quad \Rightarrow \quad \langle \ell, \eta \rangle \xrightarrow{\alpha} \langle \ell', \text{Effect}(\alpha, \eta) \rangle$$
The transition system of a program graph $\mathcal{P}$ is

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

where the transition relation $\rightarrow$ is given by the following rule

$$\ell \xrightarrow{g:\alpha} \ell' \land \eta \models g$$

$$\langle \ell, \eta \rangle \xrightarrow{\alpha} \langle \ell', \text{Effect}(\alpha, \eta) \rangle$$

is a shortform notation in SOS-style.
Structured operational semantics (SOS)

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

\[
\mathcal{T}_\mathcal{P} = (S, Act, \rightarrow, S_0, AP, L)
\]

where

the transition relation \( \rightarrow \) is given by the following rule

\[
\begin{align*}
\ell & \xleftarrow{g: \alpha} \ell' \land \eta \models g \\
\langle \ell, \eta \rangle & \xrightarrow{\alpha} \langle \ell', \text{Effect}(\alpha, \eta) \rangle
\end{align*}
\]

is a shortform notation in SOS-style.

It means that \( \rightarrow \) is the smallest relation such that:

\[
\text{if } \ell \xrightarrow{g: \alpha} \ell' \land \eta \models g \text{ then } \langle \ell, \eta \rangle \xrightarrow{\alpha} \langle \ell', \text{Effect}(\alpha, \eta) \rangle
\]
Let $\mathcal{P} = (\text{Loc}, \text{Act}, \text{Effect}, \mathcal{G}, \text{Loc}_0, g_0)$ be a PG.

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

- state space: $S = \text{Loc} \times \text{Eval(Var)}$
- initial states: $S_0 = \{ \langle \ell, \eta \rangle : \ell \in \text{Loc}_0, \eta \models g_0 \}$
- $\mathcal{G}$ is given by the following rule:

\[
\ell \xrightarrow{g : \alpha} \ell' \land \eta \models g \quad \alpha \xrightarrow{\ell, \eta} \langle \ell', \text{Effect}(\alpha, \eta) \rangle
\]
Labeling of the states

Let $\mathcal{P} = (\text{Loc}, \text{Act}, \text{Effect}, \rightarrow, \text{Loc}_0, g_0)$ be a PG. The transition system $\mathcal{T}_\mathcal{P} = (S, \text{Act}, \rightarrow, S_0, AP, L)$ has the following components:

- **state space**: $S = \text{Loc} \times \text{Eval(Var)}$
- **initial states**: $S_0 = \{ \langle \ell, \eta \rangle : \ell \in \text{Loc}_0, \eta \models g_0 \}$
- **transitions**: is given by the following rule:
  \[
  \ell \xleftarrow{g: \alpha} \ell' \land \eta \models g \\
  \langle \ell, \eta \rangle \xrightarrow{\alpha} \langle \ell', \text{Effect}(\alpha, \eta) \rangle
  \]
- **atomic propositions**: $AP = \text{Loc} \cup \text{Cond(Var)}$
Labeling of the states

Let $\mathcal{P} = (\text{Loc}, \text{Act}, \text{Effect}, \rightarrow, \text{Loc}_0, g_0)$ be a PG.

transition system $\mathcal{I}_\mathcal{P} = (S, \text{Act}, \rightarrow, S_0, AP, L)$

- state space: $S = \text{Loc} \times \text{Eval(Var)}$
- initial states: $S_0 = \{ \langle \ell, \eta \rangle : \ell \in \text{Loc}_0, \eta \models g_0 \}$
- $\rightarrow$ is given by the following rule:
  \[
  \ell \xleftarrow{g: \alpha} \ell' \land \eta \models g \\
  \langle \ell, \eta \rangle \xrightarrow{\alpha} \langle \ell', \text{Effect}(\alpha, \eta) \rangle
  \]
- atomic propositions: $AP = \text{Loc} \cup \text{Cond(Var)}$
- labeling function:
  \[
  L(\langle \ell, \eta \rangle) = \{ \ell \} \cup \{ g \in \text{Cond(Var)} : \eta \models g \}
  \]
Let \( \mathcal{P} = (\text{Loc}, \text{Act}, \text{Effect}, \rightarrow, \text{Loc}_0, g_0) \) be a PG. transition system \( \mathcal{T}_\mathcal{P} = (S, \text{Act}, \rightarrow, S_0, \text{AP}, \text{L}) \)

- state space: \( S = \text{Loc} \times \text{Eval}(\text{Var}) \)
- initial states: \( S_0 = \{ \langle \ell, \eta \rangle : \ell \in \text{Loc}_0, \eta \models g_0 \} \)
- \( \rightarrow \) is given by the following rule:
  \[
  \begin{array}{c}
  \ell \xrightarrow[g: \alpha]{\text{g:}\alpha} \ell' \wedge \eta \models g \\
  \langle \ell, \eta \rangle \xrightarrow{\alpha} \langle \ell', \text{Effect}(\eta, \alpha) \rangle
  \end{array}
  \]
- atomic propositions: \( \text{AP} = \text{Loc} \cup \text{Cond}(\text{Var}) \)
- labeling function:
  \[
  L(\langle \ell, \eta \rangle) = \{ \ell \} \cup \{ g \in \text{Cond}(\text{Var}) : \eta \models g \}
  \]
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:

\[
\text{GCL-program} \downarrow \rightarrow \text{program graph} \downarrow \rightarrow \text{transition system}
\]
guarded command $g \Rightarrow stmt$

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

$stmt$ : statement
Guarded Command Language (GCL)

guarded command \( g \Rightarrow stmt \)  \[ \iff \] enabled if \( g \) is true

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

\( stmt \) : statement
Guarded Command Language (GCL)

guarded command $g \Rightarrow stmt$ \hspace{1cm} \text{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 \( g \Rightarrow stmt \) \( \leftarrow \) enabled if \( g \) is true

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

repetitive command/loop:

\[
\text{DO } :: g \Rightarrow stmt \text{ OD } \leftarrow \text{WHILE } g \text{ DO } stmt \text{ OD}
\]
Guarded Command Language (GCL)

guarded command \( g \Rightarrow stmt \) \( \leftarrow \) enabled if \( g \) is true

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

\( stmt \) : statement

repetitive command/loop:

\[
\text{DO} \quad :: \quad g \Rightarrow stmt \quad \text{OD} \quad \leftarrow \quad \text{WHILE} \quad g \quad \text{DO} \quad stmt \quad \text{OD}
\]

conditional command:

\[
\text{IF} \quad :: \quad g \Rightarrow stmt_1 \\
:: \quad \neg g \Rightarrow stmt_2 \\
\text{FI}
\]
Guarded Command Language (GCL)

guarded command $g \Rightarrow stmt$  \hspace{1cm} \text{enabled if } g \text{ is true}

\begin{align*}
    g & : \text{guard, i.e., Boolean condition on the program variables} \\
    stmt & : \text{statement}
\end{align*}

repetitive command/loop:

\[ \text{DO } :: g \Rightarrow stmt \text{ OD} \hspace{1cm} \text{WHILE } g \text{ DO } stmt \text{ OD} \]

conditional command:

\[ \text{IF } :: g \Rightarrow stmt_1 \]
\[ :: \neg g \Rightarrow stmt_2 \]
\[ \text{FI} \]

\[ \text{IF } g \text{ THEN } stmt_1 \]
\[ \text{ELSE } stmt_2 \]
\[ \text{FI} \]
Guarded Command Language (GCL)

guarded command $g \Rightarrow stmt$ \hspace{1cm} \text{enabled if $g$ is true}

repetitive command/loop:

DO :: $g \Rightarrow stmt$ OD \hspace{1cm} \text{WHILE $g$ DO $stmt$ OD}

conditional command:

IF :: $g \Rightarrow stmt_1$
\hspace{1cm} $\neg g \Rightarrow stmt_2$
FI

IF $g$ THEN $stmt_1$
ELSE $stmt_2$
FI

symbol :: stands for the nondeterministic choice between enabled guarded commands
Guarded Command Language (GCL)

modeling language with nondeterministic choice

\[
\text{stmt} \quad \text{def} \quad x := \text{expr} \quad | \quad \text{stmt}_1; \text{stmt}_2 \\
\text{DO} \quad \implies g_1 \Rightarrow \text{stmt}_1 \quad \ldots \quad \implies g_n \Rightarrow \text{stmt}_n \quad \text{OD}
\]

\[
\text{IF} \quad \implies g_1 \Rightarrow \text{stmt}_1 \quad \ldots \quad \implies g_n \Rightarrow \text{stmt}_n \quad \text{FI}
\]

\[
; \quad ;
\]

where \( x \) is a typed variable and \( \text{expr} \) an expression of the same type
Guarded Command Language (GCL)

modeling language with nondeterministic choice

\[
\begin{align*}
\text{stmt} & \quad \textbf{def} \quad x \ := \ expr \quad \mid \quad \text{stmt}_1 ; \text{stmt}_2 \quad \mid \\
\text{DO} & \quad :: g_1 \ \Rightarrow \ \text{stmt}_1 \quad \ldots \quad :: g_n \ \Rightarrow \ \text{stmt}_n \quad \textbf{OD} \\
\text{IF} & \quad :: g_1 \ \Rightarrow \ \text{stmt}_1 \quad \ldots \quad :: g_n \ \Rightarrow \ \text{stmt}_n \quad \textbf{FI} \\
\vdots
\end{align*}
\]

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

for the number of available drinks (sprite or coke)

uses the following actions:

<table>
<thead>
<tr>
<th>Action</th>
<th>enabled</th>
<th>effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_coke</td>
<td>if $#\text{coke} &gt; 0$</td>
<td>$#\text{coke} := #\text{coke} - 1$</td>
</tr>
<tr>
<td>get_sprite</td>
<td>if $#\text{sprite} &gt; 0$</td>
<td>$#\text{sprite} := #\text{sprite} - 1$</td>
</tr>
</tbody>
</table>
GCL-program for beverage machine

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

uses the following actions:

<table>
<thead>
<tr>
<th></th>
<th>enabled</th>
<th>effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_coke</td>
<td>if $#\text{coke} &gt; 0$</td>
<td>$#\text{coke} := #\text{coke} - 1$</td>
</tr>
<tr>
<td>get_sprite</td>
<td>if $#\text{sprite} &gt; 0$</td>
<td>$#\text{sprite} := #\text{sprite} - 1$</td>
</tr>
<tr>
<td>refill</td>
<td>any time</td>
<td>$#\text{sprite} := \max$ $#\text{coke} := \max$</td>
</tr>
</tbody>
</table>
GCL-program for beverage machine

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

uses the following actions:

<table>
<thead>
<tr>
<th>action</th>
<th>enabled</th>
<th>effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_coke</td>
<td>if $#coke &gt; 0$</td>
<td>$#coke := #coke - 1$</td>
</tr>
<tr>
<td>get_sprite</td>
<td>if $#sprite &gt; 0$</td>
<td>$#sprite := #sprite - 1$</td>
</tr>
<tr>
<td>refill</td>
<td>any time</td>
<td>$#sprite := \text{max}$, $#coke := \text{max}$</td>
</tr>
<tr>
<td>insert_coin</td>
<td>any time</td>
<td>no effect on variables</td>
</tr>
</tbody>
</table>
GCL-program for beverage machine

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

uses the following actions:

<table>
<thead>
<tr>
<th>action</th>
<th>enabled</th>
<th>effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>get_coke</td>
<td>if $#coke &gt; 0$</td>
<td>$#coke ::= #coke - 1$</td>
</tr>
<tr>
<td>get_sprite</td>
<td>if $#sprite &gt; 0$</td>
<td>$#sprite ::= #sprite - 1$</td>
</tr>
<tr>
<td>refill</td>
<td>any time</td>
<td>$#sprite ::= \text{max}$</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$#coke ::= \text{max}$</td>
</tr>
<tr>
<td>insert_coin</td>
<td>any time</td>
<td>no effect on variables</td>
</tr>
<tr>
<td>return_coin</td>
<td>if machine is empty and user has entered a coin (no effect on variables)</td>
<td></td>
</tr>
</tbody>
</table>
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 ⇒ insert_coin; (* user inserts a 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

(* 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 :: 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

```
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

\[
\text{DO :: true } \Rightarrow \text{ insert \_coin;}
\]

\[
\text{IF :: } \#\text{sprite} = \#\text{coke} = 0
\]

\[
\Rightarrow \text{ return \_coin}
\]

\[
\Rightarrow \text{ get \_coke}
\]

\[
\Rightarrow \text{ get \_sprite}
\]

\[
\text{FI}
\]

\[
\text{:: true } \Rightarrow \text{ refill}
\]

\[
\text{OD}
\]

... yields a program graph with

- two variables \#sprite, \#coke \in \{0, 1, \ldots, max\}
GCL-program for beverage machine

\[ \text{start} \rightarrow \text{DO} :: \text{true} \Rightarrow \text{insert \_ coin}; \]
\[ \text{select} \rightarrow \text{IF} :: \#\text{sprite} = \#\text{coke} = 0 \Rightarrow \text{return \_ coin} \]
\[ :: \#\text{coke} > 0 \Rightarrow \text{get \_ coke} \]
\[ :: \#\text{sprite} > 0 \Rightarrow \text{get \_ sprite} \]
\[ \text{FI} \]
\[ :: \text{true} \Rightarrow \text{refill} \]
\[ \text{OD} \]

... yields a program graph with

- two variables \#\text{sprite}, \#\text{coke} ∈ \{0, 1, \ldots, \text{max}\}
- two locations \text{start} and \text{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.