Testing of Reactive Systems

Lecture 2: Modelling Reactive Systems

Henrik Bohnenkamp

Lehrstuhl Informatik 2 (MOVES)
RWTH Aachen

Summer Semester 2009

1. Actions
2. Labelled Transition Systems
3. A language to describe LTS

Actions

Actions are

- express activity of the modelled system
- are executed
- are atomic (execution is indivisible)

Actions are used to observe or to influence a system.
Actions might be

- triggered
- or inhibited

Actions

Actions are

- express activity of the modelled system
- are executed
- are atomic (execution is indivisible)

Actions are used to observe or to influence a system.

Actions might be

- triggered
- or inhibited

Actions

Actions are

- express activity of the modelled system
- are executed
- are atomic (execution is indivisible)

Actions are used to observe or to influence a system.
Actions might be

- triggered
- or inhibited

Example

Triggered or Inhibited

Consider action $a \hat{=}$ "Receiving a message over some channel"

Triggered: a triggered, if somebody actually sends a message over the channel

Inhibited: a inhibited, if there is no message

Actions

Definition

Let *Act* be the set of actions.

Note

Actions similar to symbols in an Alphabet (*cf.* Automata Theory).

- Act^*: the set of finite words over Act
- ε: the empty word
- $Act^+ = Act^* \setminus \varepsilon$
- $v \cdot w$: concatenation of words $v, w \in Act^*$

Actions

Definition

Let *Act* be the set of actions.

Note

Actions similar to symbols in an Alphabet (*cf.* Automata Theory).

- Act^*: the set of finite words over Act
- ε: the empty word
- $Act^+ = Act^* \setminus \varepsilon$
- $v \cdot w$: concatenation of words $v, w \in Act^*$

Observability

- Actions $a \in Act$ are considered observable
- Let $\tau \notin Act$: τ is the
  - silent or
  - unobservable

  action.

Why only one unobservable action?

Definition

$Act_\tau = Act \cup \{\tau\}$.

Observability

- Actions $a \in Act$ are considered observable
- Let $\tau \notin Act$: τ is the
  - silent or
  - unobservable

  action.

Why only one unobservable action?

Definition

$Act_\tau = Act \cup \{\tau\}$.

Observability

- Actions $a \in Act$ are considered observable
- Let $\tau \notin Act$: τ is the
  - silent or
  - unobservable

  action.

Why only one unobservable action?

Definition

$Act_\tau = Act \cup \{\tau\}$.

1. Actions
2. Labelled Transition Systems
3. A language to describe LTS

Labelled Transition Systems

Labelled Transition System

- One of the most fundamental models in theoretical computer science
- Ingredients: States, Transitions, Actions

Definition

A Labelled Transition System L is a tuple $L = (S, Act, \rightarrow)$, with:

- S is a set of states
- Act is a set of actions
- $\rightarrow \subseteq S \times (Act \cup \{\tau\}) \times S$ is the transition relation

Labelled Transition Systems

Labelled Transition System

- One of the most fundamental models in theoretical computer science
- Ingredients: States, Transitions, Actions

Definition

A Labelled Transition System L is a tuple $L = (S, Act, \rightarrow)$, with:

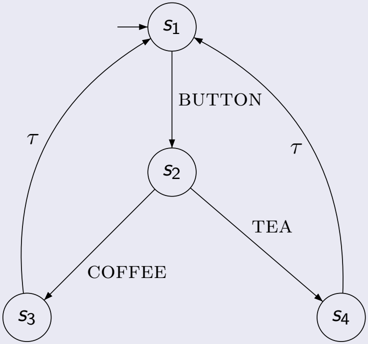
- S is a set of states
- Act is a set of actions
- $\rightarrow \subseteq S \times (Act \cup \{\tau\}) \times S$ is the transition relation

Example

Coffee 1

Let $L = (S, Act, \rightarrow)$
$S = \{s_1, s_2, s_3, s_4\}$
$Act = \{\text{COFFEE}, \text{TEA}, \text{BUTTON}\}$.

- $(s, a, s') \in \rightarrow$: s source state, s' target state
- We write $s \xrightarrow{a} s'$ if $(s, a, s') \in \rightarrow$.

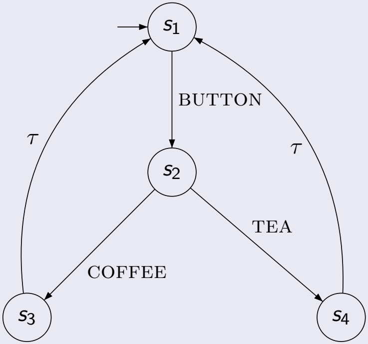


Example

Coffee 1

Let $L = (S, Act, \rightarrow)$
$S = \{s_1, s_2, s_3, s_4\}$
$Act = \{\text{COFFEE}, \text{TEA}, \text{BUTTON}\}$.

- $(s, a, s') \in \rightarrow$: s source state, s' target state
- We write $s \xrightarrow{a} s'$ if $(s, a, s') \in \rightarrow$.

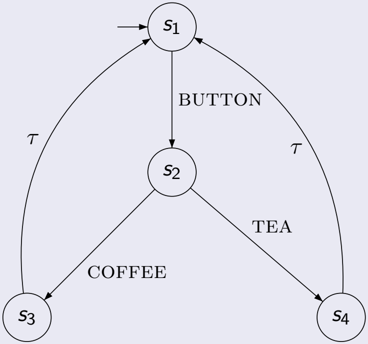


Example

Coffee 1

Let $L = (S, Act, \longrightarrow)$
$S = \{s_1, s_2, s_3, s_4\}$
$Act = \{\text{COFFEE}, \text{TEA}, \text{BUTTON}\}$.

- $(s, a, s') \in \longrightarrow$: s source state, s' target state
- We write $s \xrightarrow{a} s'$ if $(s, a, s') \in \longrightarrow$.



Deadlock

Deadlock

- $s \in S$ absorbing: $s \stackrel{a}{\not\rightarrow} \forall a \in Act_\tau$
- Alternatively: s is deadlocked
- Abbreviation: $s \not\rightarrow$ iff s is deadlocked.

Derived Transition Relations

Definition: Generalising $\rightarrow$

For $\sigma = a_1 \cdot a_2 \cdot \cdots \cdot a_n \in Act_\tau^+$:

$$p \xrightarrow{\sigma} p' \quad \text{iff} \quad \exists p_0, \ldots, p_n \in S : p_0 \xrightarrow{a_1} p_1, p_1 \xrightarrow{a_2} p_2, \cdots, p_{n-1} \xrightarrow{a_n} p_n$$

where $p = p_0$ and $p' = p_n$.

Note that τ is allowed in σ.

Abbreviations

$$p \xrightarrow{\sigma} \quad \text{iff} \quad \exists p' \in S : p \xrightarrow{\sigma} p',$$

$$p \not\xrightarrow{\sigma} \quad \text{iff} \quad \neg\exists p' \in S : p \xrightarrow{\sigma} p',$$

Derived Transition Relations

Definition: Generalising $\longrightarrow$

For $\sigma = a_1 \cdot a_2 \cdot \cdots \cdot a_n \in Act_\tau^+$:

$$p \xrightarrow{\sigma} p' \quad \text{iff} \quad \exists p_0, \ldots, p_n \in S : p_0 \xrightarrow{a_1} p_1, p_1 \xrightarrow{a_2} p_2, \cdots, p_{n-1} \xrightarrow{a_n} p_n$$

where $p = p_0$ and $p' = p_n$.

Note that τ is allowed in σ.

Abbreviations

$$p \xrightarrow{\sigma} \quad \text{iff} \quad \exists p' \in S : p \xrightarrow{\sigma} p',$$

$$p \not\xrightarrow{\sigma} \quad \text{iff} \quad \neg\exists p' \in S : p \xrightarrow{\sigma} p',$$

Derived Transition Relations

Definition: Generalising $\rightarrow$

For $\sigma = a_1 \cdot a_2 \cdot \cdots \cdot a_n \in Act_\tau^+$:

$$p \xrightarrow{\sigma} p' \quad \text{iff} \quad \exists p_0, \ldots, p_n \in S : \\ p_0 \xrightarrow{a_1} p_1, p_1 \xrightarrow{a_2} p_2, \cdots, p_{n-1} \xrightarrow{a_n} p_n$$

where $p = p_0$ and $p' = p_n$.

Note that τ is allowed in σ.

Abbreviations

$$p \xrightarrow{\sigma} \quad \text{iff} \quad \exists p' \in S : p \xrightarrow{\sigma} p',$$
$$p \not\xrightarrow{\sigma} \quad \text{iff} \quad \neg\exists p' \in S : p \xrightarrow{\sigma} p',$$

Derived Transition Relations

Definition: Abstracting from τ

$$
\begin{aligned}
p \stackrel{\varepsilon}{\Longrightarrow} p' \quad & \text{iff} \quad p = p' \text{or } p \stackrel{\tau^n}{\longrightarrow} p' \text{ for some } n \\
p \stackrel{a}{\Longrightarrow} p' \quad & \text{iff} \quad \exists p_1, p_2 \in S : p \stackrel{\varepsilon}{\Longrightarrow} p_1 \stackrel{a}{\longrightarrow} p_2 \stackrel{\varepsilon}{\Longrightarrow} p' \\
p \stackrel{\sigma \cdot a}{\Longrightarrow} p' \quad & \text{iff} \quad \exists p'' \in S : p \stackrel{\sigma}{\Longrightarrow} p'' \stackrel{a}{\Longrightarrow} p'
\end{aligned}
$$

- With the $\Longrightarrow$ relation we abstract from τ transitions.
- $p \stackrel{\sigma}{\Longrightarrow}$ and $p \stackrel{\sigma}{\not\Longrightarrow}$ are defined analogous to "$\longrightarrow$" before.

Derived Transition Relations

Definition: Abstracting from τ

$$
\begin{aligned}
p \stackrel{\varepsilon}{\Longrightarrow} p' \quad &\text{iff} \quad p = p' \text{or } p \stackrel{\tau^n}{\longrightarrow} p' \text{ for some } n \\
p \stackrel{a}{\Longrightarrow} p' \quad &\text{iff} \quad \exists p_1, p_2 \in S : p \stackrel{\varepsilon}{\Longrightarrow} p_1 \stackrel{a}{\longrightarrow} p_2 \stackrel{\varepsilon}{\Longrightarrow} p' \\
p \stackrel{\sigma \cdot a}{\Longrightarrow} p' \quad &\text{iff} \quad \exists p'' \in S : p \stackrel{\sigma}{\Longrightarrow} p'' \stackrel{a}{\Longrightarrow} p'
\end{aligned}
$$

- With the $\Longrightarrow$ relation we abstract from τ transitions.
- $p \stackrel{\sigma}{\Longrightarrow}$ and $p \stackrel{\sigma}{\not\Longrightarrow}$ are defined analogous to "$\longrightarrow$" before.

Derived Transition Relations

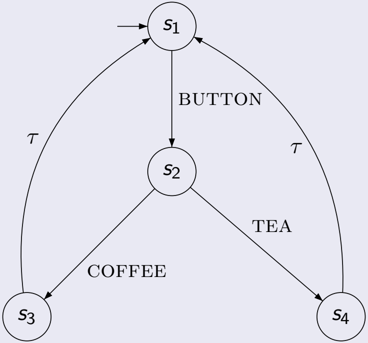
Definition: Abstracting from τ

$$p \stackrel{\varepsilon}{\Longrightarrow} p' \quad \text{iff} \quad p = p' \text{or } p \stackrel{\tau^n}{\longrightarrow} p' \text{ for some } n$$
$$p \stackrel{a}{\Longrightarrow} p' \quad \text{iff} \quad \exists p_1, p_2 \in S : p \stackrel{\varepsilon}{\Longrightarrow} p_1 \stackrel{a}{\longrightarrow} p_2 \stackrel{\varepsilon}{\Longrightarrow} p'$$
$$p \stackrel{\sigma \cdot a}{\Longrightarrow} p' \quad \text{iff} \quad \exists p'' \in S : p \stackrel{\sigma}{\Longrightarrow} p'' \stackrel{a}{\Longrightarrow} p'$$

- With the $\Longrightarrow$ relation we abstract from τ transitions.
- $p \stackrel{\sigma}{\Longrightarrow}$ and $p \stackrel{\sigma}{\not\Longrightarrow}$ are defined analogous to "$\longrightarrow$" before.

Example

Some derived transitions



1. $s_1 \xrightarrow{\text{BUTTON}\cdot\text{COFFEE}} s_3$
2. $s_1 \xrightarrow{\text{BUTTON}\cdot\text{COFFEE}\cdot\tau} s_1$
3. $s_1 \overset{\text{BUTTON}\cdot\text{TEA}}{\Longrightarrow} s_4$, but also
4. $s_1 \overset{\text{BUTTON}\cdot\text{TEA}}{\Longrightarrow} s_1$

Traces

Describing Dynamic Behaviour of LTS

- $\exists$ many different approaches to describe behaviour of LTS
- most basic: traces

Definition: Traces

Let $s \in S$. The set of traces of s, denoted $traces(s)$, is defined as

$$traces(s) = \{\sigma \in Act^* \mid s \stackrel{\sigma}{\Longrightarrow}\}.$$

Traces

Describing Dynamic Behaviour of LTS

- $\exists$ many different approaches to describe behaviour of LTS
- most basic: traces

Definition: Traces

Let $s \in S$. The set of traces of s, denoted $traces(s)$, is defined as

$$traces(s) = \{\sigma \in Act^* \mid s \stackrel{\sigma}{\Longrightarrow}\}.$$

Traces

Note

- traces $traces(s)$ are actually a language
- if we see LTS L as an NFA with
  - start state s
  - all states $s' \in S$ accepting

  then $traces(s)$ is the language accepted by this automaton.

One more note

- We will refer to all words $\sigma \in Act^*$ as traces
- i.e., we will use word and trace synonymously.

Traces

Note

- traces $traces(s)$ are actually a language
- if we see LTS L as an NFA with
  - start state s
  - all states $s' \in S$ accepting

then $traces(s)$ is the language accepted by this automaton.

One more note

- We will refer to all words $\sigma \in Act^*$ as traces
- i.e., we will use word and trace synonymously.

Traces

Note

- traces $traces(s)$ are actually a language
- if we see LTS L as an NFA with
  - start state s
  - all states $s' \in S$ accepting

then $traces(s)$ is the language accepted by this automaton.

One more note

- We will refer to all words $\sigma \in Act^*$ as traces
- i.e., we will use word and trace synonymously.

Traces

Note

- traces $traces(s)$ are actually a language
- if we see LTS L as an NFA with
  - start state s
  - all states $s' \in S$ accepting

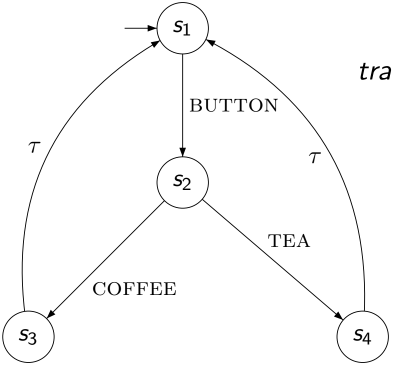
  then $traces(s)$ is the language accepted by this automaton.

One more note

- We will refer to all words $\sigma \in Act^*$ as traces
- i.e., we will use word and trace synonymously.

Example

$$traces(s_3) = \{\varepsilon,$$
$$\text{BUTTON}, \text{BUTTON} \cdot \text{TEA},$$
$$\text{BUTTON} \cdot \text{TEA} \cdot \text{BUTTON},$$
$$\ldots\} = traces(s_1) = traces(s_4)$$



$$traces(s_2) = \{\varepsilon,$$
$$\text{TEA}, \text{COFFEE},$$
$$\text{TEA} \cdot \text{BUTTON}$$
$$\text{TEA} \cdot \text{BUTTON} \cdot \text{TEA}$$
$$\ldots\}$$
$$= \text{COFFEE} \cdot traces(s_1)$$
$$\cup \text{TEA} \cdot traces(s_1)$$
$$\cup \{\varepsilon\}$$

Reachable states

What states can be reached from state s with trace σ?

Definition: after

For $s \in S, \sigma \in Act^*$:

- s **after** $\sigma := \{s' \mid s \overset{\sigma}{\Longrightarrow} s'\}$
- For $S' \subseteq S$: S **after** $\sigma := \bigcup_{s \in S'} s$ **after** σ
- For $A \subseteq Act^*$: s **after** $A := \bigcup_{\sigma \in A} s$ **after** σ

Note

s **after** Act^* are called the derivatives of s, or reachable states from s.

Reachable states

What states can be reached from state s with trace σ?

Definition: after

For $s \in S, \sigma \in Act^*$:

- s **after** σ $:= \{s' \mid s \stackrel{\sigma}{\Longrightarrow} s'\}$
- For $S' \subseteq S$: S **after** σ $:= \bigcup_{s \in S'} s$ **after** σ
- For $A \subseteq Act^*$: s **after** A $:= \bigcup_{\sigma \in A} s$ **after** σ

Note

s **after** Act^* are called the derivatives of s, or reachable states from s.

Reachable states

What states can be reached from state s with trace σ?

Definition: after

For $s \in S, \sigma \in Act^*$:

- s **after** $\sigma := \{s' \mid s \overset{\sigma}{\Longrightarrow} s'\}$
- For $S' \subseteq S$: S **after** $\sigma := \bigcup_{s \in S'} s$ **after** σ
- For $A \subseteq Act^*$: s **after** $A := \bigcup_{\sigma \in A} s$ **after** σ

Note

s **after** Act^* are called the derivatives of s, or reachable states from s.

Reachable states

What states can be reached from state s with trace σ?

Definition: after

For $s \in S, \sigma \in Act^*$:

- s **after** σ $:= \{s' \mid s \stackrel{\sigma}{\Longrightarrow} s'\}$
- For $S' \subseteq S$: S **after** σ $:= \bigcup_{s \in S'} s$ **after** σ
- For $A \subseteq Act^*$: s **after** A $:= \bigcup_{\sigma \in A} s$ **after** σ

Note

s **after** Act^* are called the derivatives of s, or reachable states from s.

Nondeterminism

Definition: Deterministic LTS

A state $s \in S$ is called deterministic iff

$$\forall \sigma \in traces(s) : |\underline{s \textbf{ after } \sigma}| = 1$$

- An LTS is called determinstic, if all its states are deterministic
- An LTS that is not deterministic is non-deterministic

Note:

Equivalent is: $\forall \sigma \in Act^* : |\underline{s \textbf{ after } \sigma}| \leq 1$ (Why?).

Nondeterminism

Definition: Deterministic LTS

A state $s \in S$ is called deterministic iff

$$\forall \sigma \in traces(s) : |\underline{s \textbf{ after } \sigma}| = 1$$

- An LTS is called determinstic, if all its states are deterministic
- An LTS that is not deterministic is non-deterministic

Note:

Equivalent is: $\forall \sigma \in Act^* : |\underline{s \textbf{ after } \sigma}| \leq 1$ (Why?).

Nondeterminism

Definition: Deterministic LTS

A state $s \in S$ is called deterministic iff

$$\forall \sigma \in traces(s) : |\underline{s\ \textbf{after}\ \sigma}| = 1$$

- An LTS is called determinstic, if all its states are deterministic
- An LTS that is not deterministic is non-deterministic

Note:

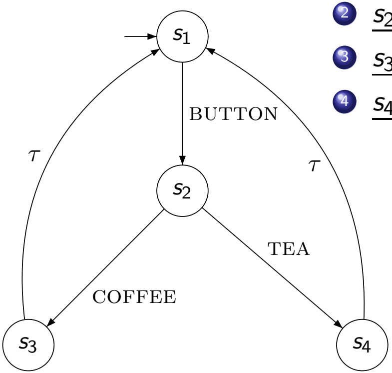
Equivalent is: $\forall \sigma \in Act^* : |\underline{s\ \textbf{after}\ \sigma}| \leq 1$ (Why?).

Nondeterminism

Sources of nondeterminism

1. nondeterministic branching: two outgoing transitions with same action
2. τ-transitions

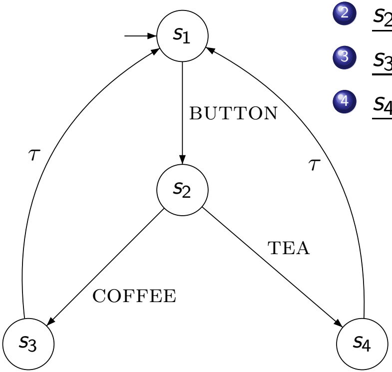
Example



1. s_1 **after** BUTTON · COFFEE =
2. s_2 **after** TEA =
3. s_3 **after** ε =
4. s_4 **after** BUTTON · TEA =

Nondeterminism is here solely caused by the τ-transitions.

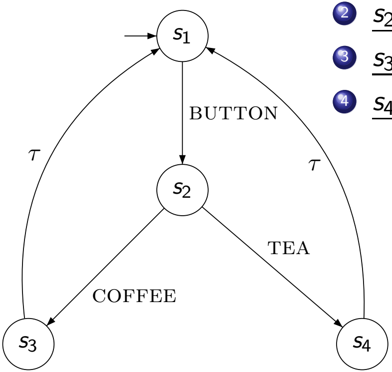
Example



1. $\underline{s_1 \textbf{ after } \text{BUTTON} \cdot \text{COFFEE}} = \{s_3, s_1\}$
2. $\underline{s_2 \textbf{ after } \text{TEA}} =$
3. $\underline{s_3 \textbf{ after } \varepsilon} =$
4. $\underline{s_4 \textbf{ after } \text{BUTTON} \cdot \text{TEA}} =$

Nondeterminism is here solely caused by the τ-transitions.

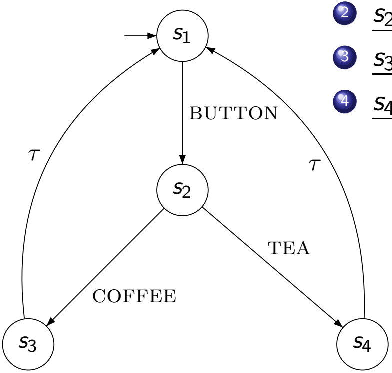
Example



1. s_1 **after** BUTTON · COFFEE $= \{s_3, s_1\}$
2. s_2 **after** TEA $= \{s_4, s_1\}$
3. s_3 **after** $\varepsilon =$
4. s_4 **after** BUTTON · TEA $=$

Nondeterminism is here solely caused by the τ-transitions.

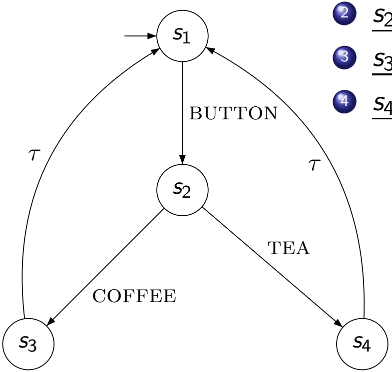
Example



1. s_1 **after** BUTTON · COFFEE $= \{s_3, s_1\}$
2. s_2 **after** TEA $= \{s_4, s_1\}$
3. s_3 **after** $\varepsilon = \{s_3, s_1\}$
4. s_4 **after** BUTTON · TEA $=$

Nondeterminism is here solely caused by the τ-transitions.

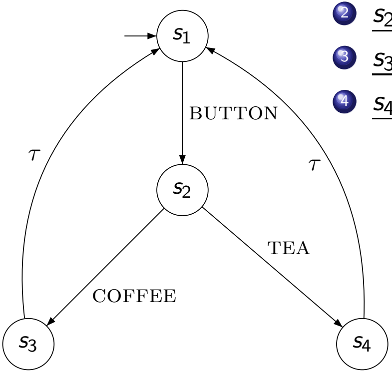
Example



1. $\underline{s_1 \textbf{ after } \text{BUTTON} \cdot \text{COFFEE}} = \{s_3, s_1\}$
2. $\underline{s_2 \textbf{ after } \text{TEA}} = \{s_4, s_1\}$
3. $\underline{s_3 \textbf{ after } \varepsilon} = \{s_3, s_1\}$
4. $\underline{s_4 \textbf{ after } \text{BUTTON} \cdot \text{TEA}} = \{s_4, s_1\}$

Nondeterminism is here solely caused by the τ-transitions.

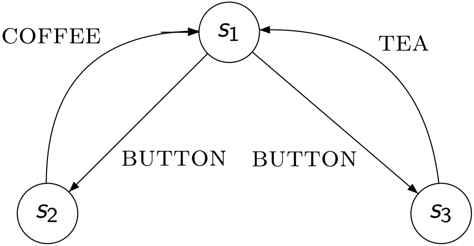
Example



1. $\underline{s_1 \textbf{ after } \text{BUTTON} \cdot \text{COFFEE}} = \{s_3, s_1\}$
2. $\underline{s_2 \textbf{ after } \text{TEA}} = \{s_4, s_1\}$
3. $\underline{s_3 \textbf{ after } \varepsilon} = \{s_3, s_1\}$
4. $\underline{s_4 \textbf{ after } \text{BUTTON} \cdot \text{TEA}} = \{s_4, s_1\}$

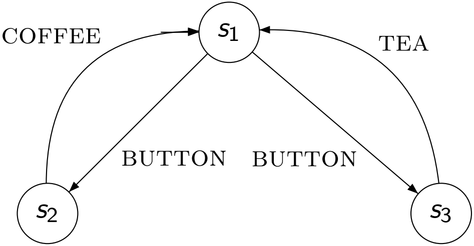
Nondeterminism is here solely caused by the τ-transitions.

Example



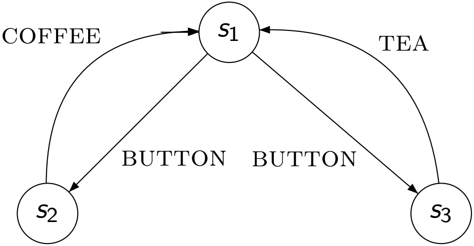
1. $\underline{s_1 \text{ after } \text{BUTTON}} =$
2. $\underline{s_1 \text{ after } \text{BUTTON} \cdot \text{TEA}} =$
3. $\underline{s_2 \text{ after } \text{COFFEE} \cdot \text{BUTTON}} =$

Example



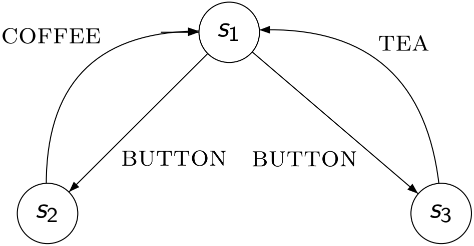
1. $\underline{s_1\ \textbf{after}\ \text{BUTTON}} = \{s_2, s_3\}$
2. $\underline{s_1\ \textbf{after}\ \text{BUTTON} \cdot \text{TEA}} =$
3. $\underline{s_2\ \textbf{after}\ \text{COFFEE} \cdot \text{BUTTON}} =$

Example



1. $\underline{s_1 \ \textbf{after} \ \text{BUTTON}} = \{s_2, s_3\}$
2. $\underline{s_1 \ \textbf{after} \ \text{BUTTON} \cdot \text{TEA}} = \{s_1\}$
3. $\underline{s_2 \ \textbf{after} \ \text{COFFEE} \cdot \text{BUTTON}} =$

Example



1. $\underline{s_1 \textbf{ after } \text{BUTTON}} = \{s_2, s_3\}$
2. $\underline{s_1 \textbf{ after } \text{BUTTON} \cdot \text{TEA}} = \{s_1\}$
3. $\underline{s_2 \textbf{ after } \text{COFFEE} \cdot \text{BUTTON}} = \{s_2, s_3\}$

A language to describe LTS

Reminder

Known from automata theory: regular expressions.

- 0 is a regular expression.
- 1 is a regular expression.
- for $a \in Act$: a is a regular expression.
- for e, e' regular expressions:
  - $e \cdot e'$ is a regular expression.
  - $e|e'$ is a regular expression.
  - e^* is a regular expression.

Regular expressions can be turned into finite automata

We shall now define expressions that describe LTS. We will call these expressions processes.

A language to describe LTS

Reminder

Known from automata theory: regular expressions.

- 0 is a regular expression.
- 1 is a regular expression.
- for $a \in Act$: a is a regular expression.
- for e, e' regular expressions:
  - $e \cdot e'$ is a regular expression.
  - $e|e'$ is a regular expression.
  - e^* is a regular expression.

Regular expressions can be turned into finite automata

We shall now define expressions that describe LTS. We will call these expressions processes.

A language to describe LTS

Reminder

Known from automata theory: regular expressions.

- 0 is a regular expression.
- 1 is a regular expression.
- for $a \in Act$: a is a regular expression.
- for e, e' regular expressions:
  - $e \cdot e'$ is a regular expression.
  - $e|e'$ is a regular expression.
  - e^* is a regular expression.

Regular expressions can be turned into finite automata

We shall now define expressions that describe LTS. We will call these expressions processes.

A language to describe LTS

Reminder

Known from automata theory: regular expressions.

- 0 is a regular expression.
- 1 is a regular expression.
- for $a \in Act$: a is a regular expression.
- for e, e' regular expressions:
  - $e \cdot e'$ is a regular expression.
  - $e|e'$ is a regular expression.
  - e^* is a regular expression.

Regular expressions can be turned into finite automata

We shall now define expressions that describe LTS. We will call these expressions processes.

A language to describe LTS

Reminder

Known from automata theory: regular expressions.

- 0 is a regular expression.
- 1 is a regular expression.
- for $a \in Act$: a is a regular expression.
- for e, e' regular expressions:
  - $e \cdot e'$ is a regular expression.
  - $e|e'$ is a regular expression.
  - e^* is a regular expression.

Regular expressions can be turned into finite automata

We shall now define expressions that describe LTS. We will call these expressions processes.

A language to describe LTS

Reminder

Known from automata theory: regular expressions.

- 0 is a regular expression.
- 1 is a regular expression.
- for $a \in Act$: a is a regular expression.
- for e, e' regular expressions:
  - $e \cdot e'$ is a regular expression.
  - $e|e'$ is a regular expression.
  - e^* is a regular expression.

Regular expressions can be turned into finite automata

We shall now define expressions that describe LTS. We will call these expressions processes.

A language to describe LTS

Reminder

Known from automata theory: regular expressions.

- 0 is a regular expression.
- 1 is a regular expression.
- for $a \in Act$: a is a regular expression.
- for e, e' regular expressions:
  - $e \cdot e'$ is a regular expression.
  - $e|e'$ is a regular expression.
  - e^* is a regular expression.

Regular expressions can be turned into finite automata

We shall now define expressions that describe LTS. We will call these expressions processes.

A language to describe LTS

Reminder

Known from automata theory: regular expressions.

- 0 is a regular expression.
- 1 is a regular expression.
- for $a \in Act$: a is a regular expression.
- for e, e' regular expressions:
  - $e \cdot e'$ is a regular expression.
  - $e|e'$ is a regular expression.
  - e^* is a regular expression.

Regular expressions can be turned into finite automata

We shall now define expressions that describe LTS. We will call these expressions processes.

A language to describe LTS

Reminder

Known from automata theory: regular expressions.

- 0 is a regular expression.
- 1 is a regular expression.
- for $a \in Act$: a is a regular expression.
- for e, e' regular expressions:
  - $e \cdot e'$ is a regular expression.
  - $e|e'$ is a regular expression.
  - e^* is a regular expression.

Regular expressions can be turned into finite automata

We shall now define expressions that describe LTS. We will call these expressions processes.

A language to describe LTS

Reminder

Known from automata theory: regular expressions.

- 0 is a regular expression.
- 1 is a regular expression.
- for $a \in Act$: a is a regular expression.
- for e, e' regular expressions:
  - $e \cdot e'$ is a regular expression.
  - $e|e'$ is a regular expression.
  - e^* is a regular expression.

Regular expressions can be turned into finite automata

We shall now define expressions that describe LTS. We will call these expressions processes.

A language to describe LTS

Definition: Processes $\mathbb{P}$

- Let $\mathcal{P}$ be the set of process variables.
- Let Act be a set of actions.
- The set $\mathbb{P}$ of processes is the language defined by the following grammar:

$$p \quad \rightarrow \quad \mathrm{STOP} \;\Big|\; a.p \;\Big|\; p+p \;\Big|\; p\|_A p \;\Big|\; P$$

where $a \in Act_\tau$, $A \subseteq Act$, and $P \in \mathcal{P}$.

- Process definitions are of the form

$$P \quad \triangleq \quad p$$

with $p \in \mathbb{P}$ and $P \in \mathcal{P}$.

A language to describe LTS

Definition: Processes $\mathbb{P}$

- Let $\mathcal{P}$ be the set of process variables.
- Let Act be a set of actions.
- The set $\mathbb{P}$ of processes is the language defined by the following grammar:

$$p \quad \rightarrow \quad \mathrm{STOP} \;\Big|\; a.p \;\Big|\; p{+}p \;\Big|\; p\|_A p \;\Big|\; P$$

where $a \in Act_\tau$, $A \subseteq Act$, and $P \in \mathcal{P}$.

- Process definitions are of the form

$$P \quad \hat{=} \quad p$$

with $p \in \mathbb{P}$ and $P \in \mathcal{P}$.

A language to describe LTS

Definition: Processes $\mathbb{P}$

- Let $\mathcal{P}$ be the set of process variables.
- Let Act be a set of actions.
- The set $\mathbb{P}$ of processes is the language defined by the following grammar:

$$p \quad \rightarrow \quad \mathrm{STOP} \;\Big|\; a.p \;\Big|\; p+p \;\Big|\; p\|_A p \;\Big|\; P$$

where $a \in Act_\tau$, $A \subseteq Act$, and $P \in \mathcal{P}$.

- Process definitions are of the form

$$P \quad \hat{=} \quad p$$

with $p \in \mathbb{P}$ and $P \in \mathcal{P}$.

A language to describe LTS

Definition: Processes $\mathbb{P}$

- Let $\mathcal{P}$ be the set of process variables.
- Let Act be a set of actions.
- The set $\mathbb{P}$ of processes is the language defined by the following grammar:

$$p \quad \to \quad \mathrm{STOP} \;\Big|\; a.p \;\Big|\; p+p \;\Big|\; p\|_A p \;\Big|\; P$$

  where $a \in Act_\tau$, $A \subseteq Act$, and $P \in \mathcal{P}$.
- Process definitions are of the form

$$P \quad \triangleq \quad p$$

  with $p \in \mathbb{P}$ and $P \in \mathcal{P}$.

A language to describe LTS

$$p \quad \rightarrow \quad \mathrm{STOP} \;\Big|\; a.p \;\Big|\; p{+}p \;\Big|\; p\|_A p \;\Big|\; P$$

Informal Meaning

Let $p, q \in \mathbb{P}$.

STOP: is the process that does nothing, is deadlocked.

$a.p$: executes action $a \in Act_\tau$ and behaves like process p.
the prefix operator

$p + q$: behaves either like process p or q.
the choice operator

$p\|_A q$: behaves like p and q running in parallel, synchronising over synchronisation set A
the parallel operator

P: if $P \triangleq p$, then P behaves exactly like p

A language to describe LTS

$$p \;\rightarrow\; \mathrm{STOP} \;\Big|\; a.p \;\Big|\; p{+}p \;\Big|\; p\|_A p \;\Big|\; P$$

Informal Meaning

Let $p, q \in \mathbb{P}$.

STOP: is the process that does nothing, is deadlocked.

$a.p$: executes action $a \in Act_\tau$ and behaves like process p.

the prefix operator

$p + q$: behaves either like process p or q.

the choice operator

$p\|_A q$: behaves like p and q running in parallel, synchronising over synchronisation set A

the parallel operator

P: if $P \triangleq p$, then P behaves exactly like p

A language to describe LTS

$$p \quad \rightarrow \quad \mathrm{STOP} \;\Big|\; a.p \;\Big|\; p+p \;\Big|\; p\|_A p \;\Big|\; P$$

Informal Meaning

Let $p, q \in \mathbb{P}$.

STOP: is the process that does nothing, is deadlocked.

$a.p$: executes action $a \in Act_\tau$ and behaves like process p.

the prefix operator

$p + q$: behaves either like process p or q.

the choice operator

$p\|_A q$: behaves like p and q running in parallel, synchronising over synchronisation set A

the parallel operator

P: if $P \triangleq p$, then P behaves exactly like p

A language to describe LTS

$$p \quad \rightarrow \quad \mathrm{STOP} \;\Big|\; a.p \;\Big|\; p{+}p \;\Big|\; p\|_A p \;\Big|\; P$$

Informal Meaning

Let $p, q \in \mathbb{P}$.

STOP: is the process that does nothing, is deadlocked.

$a.p$: executes action $a \in Act_\tau$ and behaves like process p.

the prefix operator

$p + q$: behaves either like process p or q.

the choice operator

$p\|_A q$: behaves like p and q running in parallel, synchronising over synchronisation set A

the parallel operator

P: if $P \triangleq p$, then P behaves exactly like p

A language to describe LTS

$$p \quad \rightarrow \quad \mathrm{STOP} \;\Big|\; a.p \;\Big|\; p{+}p \;\Big|\; p\|_A p \;\Big|\; P$$

Informal Meaning

Let $p, q \in \mathbb{P}$.

- STOP: is the process that does nothing, is deadlocked.
- $a.p$: executes action $a \in Act_\tau$ and behaves like process p.

  the prefix operator
- $p + q$: behaves either like process p or q.

  the choice operator
- $p\|_A q$: behaves like p and q running in parallel, synchronising over synchronisation set A

  the parallel operator
- P: if $P \triangleq p$, then P behaves exactly like p

A language to describe LTS

$$p \quad \rightarrow \quad \mathrm{STOP} \;\Big|\; a.p \;\Big|\; p{+}p \;\Big|\; p\|_A p \;\Big|\; P$$

Informal Meaning

Let $p, q \in \mathbb{P}$.

STOP: is the process that does nothing, is deadlocked.

$a.p$: executes action $a \in Act_\tau$ and behaves like process p.

the prefix operator

$p + q$: behaves either like process p or q.

the choice operator

$p\|_A q$: behaves like p and q running in parallel, synchronising over synchronisation set A

the parallel operator

P: if $P \triangleq p$, then P behaves exactly like p

A language to describe LTS

$$p \;\rightarrow\; \mathrm{STOP} \;\Big|\; a.p \;\Big|\; p{+}p \;\Big|\; p\|_A p \;\Big|\; P$$

Informal Meaning

Let $p, q \in \mathbb{P}$.

- STOP: is the process that does nothing, is deadlocked.
- $a.p$: executes action $a \in Act_\tau$ and behaves like process p.
  the prefix operator
- $p + q$: behaves either like process p or q.
  the choice operator
- $p\|_A q$: behaves like p and q running in parallel, synchronising over synchronisation set A
  the parallel operator
- P: if $P \triangleq p$, then P behaves exactly like p

A language to describe LTS

$$p \;\rightarrow\; \mathrm{STOP} \;\Big|\; a.p \;\Big|\; p{+}p \;\Big|\; p\|_A p \;\Big|\; P$$

Informal Meaning

Let $p, q \in \mathbb{P}$.

- STOP: is the process that does nothing, is deadlocked.
- $a.p$: executes action $a \in Act_\tau$ and behaves like process p.
  the prefix operator
- $p + q$: behaves either like process p or q.
  the choice operator
- $p\|_A q$: behaves like p and q running in parallel, synchronising over synchronisation set A
  the parallel operator
- P: if $P \triangleq p$, then P behaves exactly like p

A language to describe LTS

Note

The parallel operator makes this language very powerful:

- componentwise independent specification possible
- combination by parallel composition

Note 2

- Behaviour of a process can be described by LTS.
- processes are also states, *i.e.*, $\mathbb{P}$ is the set of states of the LTS that we will consider.

A language to describe LTS

Note

The parallel operator makes this language very powerful:

- componentwise independent specification possible
- combination by parallel composition

Note 2

- Behaviour of a process can be described by LTS.
- processes are also states, *i.e.*, $\mathbb{P}$ is the set of states of the LTS that we will consider.

A language to describe LTS

Note

The parallel operator makes this language very powerful:

- componentwise independent specification possible
- combination by parallel composition

Note 2

- Behaviour of a process can be described by LTS.
- processes are also states, *i.e.*, $\mathbb{P}$ is the set of states of the LTS that we will consider.

Example 1.3.2: Some simple processes

A language to describe LTS

Recursion

- Up til now only terminating (= deadlocking) processes
- Use process definitions for non-terminating behaviour
- . . . recursive process definitions

A language to describe LTS

Recursion

- Up til now only terminating (= deadlocking) processes
- Use process definitions for non-terminating behaviour
- . . . recursive process definitions

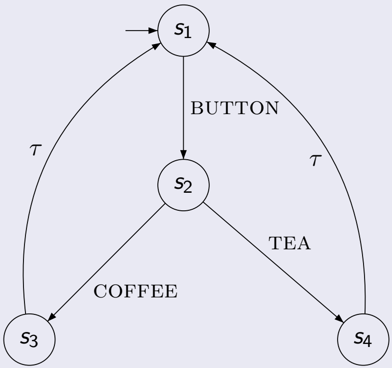
A language to describe LTS

Recursion

- Up til now only terminating (= deadlocking) processes
- Use process definitions for non-terminating behaviour
- . . . recursive process definitions

Example 1.3.3

Coffee 1



Example 1.3.3

Coffee 2