Testing of Reactive Systems

—Course Notes—

Henrik Bohnenkamp

Summer Semester 2008

Chapter 1

Modelling Reactive Systems

The basic ingredients to model reactive systems:

Actions, **States**, and **Transitions**

1.1 Actions

Actions

- model 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* by the environment.

1.1.1 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

■ End Example

Let Act be the set of actions.

Actions are much similar to symbols in an Alphabet (*cf.* Automata Theory). We use

- 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 $a \in Act$ are considered *observable*: you can see if and when they are executed, and they can influence the environment, or be influenced by it. However, sometimes it is necessary to express activity that happens without being observable. For this we introduce one extra action.

1.1.2 Definition (τ) With $\tau \notin Act$ we denote a special action: the *silent*, or *unobservable* action

The reason why we need only one unobservable action is simple. If there were several unobservable actions, there would be no way to distinguish them by observation, them being unobservable. unobservable implies indistinguishable

For the sake of easier reference, we define

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

1.2 Labelled Transition Systems

One of the most fundamental models in theoretical computer science is the labelled transition system (abbreviated LTS). In its different varieties, it can be used to describe the behaviour of complicated reactive systems with very simple mechanisms. States are one basic ingredient of a LTS.

- States denote *current state* of modelled system
- They also represent the potential *future behaviour*

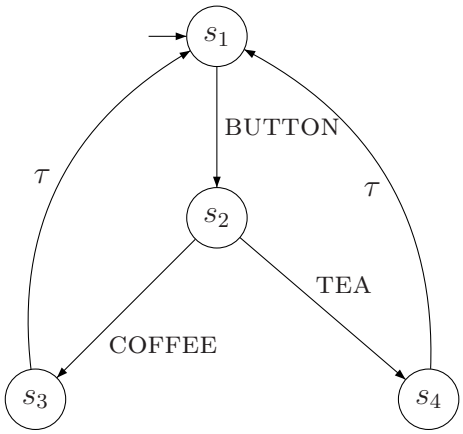
The other ingredient, apart from actions, are the *labelled transitions*. These describe the potential state changes of the LTS than can occur due to action execution.

1.2.1 Definition (Labelled Transition System (LTS)) 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
- $\rightarrow \subseteq S \times (Act \cup \{\tau\}) \times S$

1.2.2 Example (Coffee 1, slide #1)
Let $L = (S, Act, \rightarrow)$ with $S = \{s_1, s_2, s_3, s_4\}$ and $Act = \{\text{COFFEE}, \text{TEA}, \text{BUTTON}\}$. In the figure, we see the LTS represented as graph, with the nodes representing the states, and the labelled arrows the transitions.

Note: All material on the slides used in the lecture is in the script.



Not that an LTS is similar to a (finite) automaton. However, the difference here is that we usually do not have a starting state, and also no accepting states.

■ END EXAMPLE

When $(s, a, s') \in \rightarrow$, we write $s \xrightarrow{a} s'$.

In the following Definitions, Lemmas etc., we assume an LTS $L = (S, Act, \rightarrow)$ as given. In the following definition, we introduce two generalisations of the transition relation $\rightarrow$: first we extend $\rightarrow$ such that it can be labelled with words over Act_τ. Then we define a so-called *weak* transition relation which is labelled with words over Act, *i.e.,* we abstract from τ-transitions.

1.2.3 Definition (Derived Transition Relations)

1. We lift $\rightarrow$ to Act_τ^+ as follows: for $\sigma = a_1 \cdot a_2 \cdot \cdots \cdot a_n \in Act_\tau^+$ we define

$$\begin{array}{ll} & p \xrightarrow{\sigma} p' \\ \text{iff} & \exists p_0, \ldots, p_n \in S : p_0 \xrightarrow{a_1} p_1, p_1 \xrightarrow{a_2} p_2, \cdots, p_{n-1} \xrightarrow{a_n} p_n \\ & \text{where } p = p_0, p' = p_n. \end{array}$$

   Later, the following abbreviations will be useful:

$$\begin{array}{rll} p \xrightarrow{\sigma} & \text{iff} & \exists p' \in S : p \xrightarrow{\sigma} p', \\ p \not\xrightarrow{\sigma} & \text{iff} & \neg\exists p' \in S : p \xrightarrow{\sigma} p', \end{array}$$

   i.e., if, for $\sigma \in Act_\tau^+$, there is a state p' such that $p \xrightarrow{\sigma} p'$, then we write $p \xrightarrow{\sigma}$. If there is no such p', then we write $p \not\xrightarrow{\sigma}$

Note: All material on the slides used in the lecture is in the script.

2. With the $\Longrightarrow$ relation we abstract from τ transitions.

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

   $p \stackrel{\sigma}{\Longrightarrow}$ and $p \stackrel{\sigma}{\not\Longrightarrow}$ are defined analogous to "$\rightarrow$" above.

3. We call states s with $s \not\rightarrow \forall a \in Act_\tau$ *absorbing*, or *deadlock states*. If s is deadlocked, we abbreviate this as $s \not\rightarrow$.

1.2.4 Example (Some derived transitions)
See LTS in Example 1.2.2: we have

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

■ END EXAMPLE

An LTS describes the complete behaviour of a reactive system. However, in order to understand this behaviour, it is important to also look at the actual executions of the LTS. There are many different approaches to do that (and we will see some of them later), but the most basic approach is to look at the sequences of observable actions that the LTS can produce: the so-called *traces*.

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

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

It might be interesting to realize that the traces $traces(s)$ are actually a *language*: if we see LTS L as an automaton with start state s with all states $s' \in S$ being accepting, then $traces(s)$ is the language accepted by this automaton.

Although traces are actually only defined for the states of the LTS, we will not put too fine a point on this and refer later to all words $\sigma \in Act^*$ as traces, *i.e.,* we will use *word* and *trace* synonymously.

Note: All material on the slides used in the lecture is in the script.

1.2.6 Example (Traces for Example 1.2.2, slide #2)
See Example 1.2.2.

$$\begin{aligned} traces(s_3) &= \{\varepsilon, \\ &\quad \text{BUTTON}, \text{BUTTON} \cdot \text{TEA}, \\ &\quad \text{BUTTON} \cdot \text{TEA} \cdot \text{BUTTON}, \\ &\quad \ldots\} = traces(s_1) = traces(s_4) \end{aligned}$$

$$\begin{aligned} traces(s_2) &= \{\varepsilon, \\ &\quad \text{TEA}, \text{COFFEE}, \\ &\quad \text{TEA} \cdot \text{BUTTON} \\ &\quad \text{TEA} \cdot \text{BUTTON} \cdot \text{TEA} \\ &\quad \ldots\} \\ &= \text{COFFEE} \cdot traces(s_1) \\ &\quad \cup \text{TEA} \cdot traces(s_1) \\ &\quad \cup \{\varepsilon\} \end{aligned}$$

■ END EXAMPLE

It will become later very important to know which states can be reached with a trace σ, starting from a state s. For this we introduce $\underline{\cdot \textbf{ after } \cdot}$, which is a function in infix notation.

1.2.7 Definition ($\underline{\cdot \textbf{ after } \cdot}$) For $s \in S, \sigma \in Act^*$:

- $\underline{s \textbf{ after } \sigma} := \{s' \mid s \stackrel{\sigma}{\Longrightarrow} s'\}$
- For $S' \subseteq S$: $\underline{S \textbf{ after } \sigma} := \bigcup_{s \in S'} \underline{s \textbf{ after } \sigma}$
- For $A \subseteq Act^*$: $\underline{s \textbf{ after } A} := \bigcup_{\sigma \in A} \underline{s \textbf{ after } \sigma}$ (the *derivatives* of s)

The special set of derivatives $\underline{s \textbf{ after } Act^*}$ is the set of all states that are *reachable* from s in LTS L.
A very important phenomenon is that of nondeterminism.

1.2.8 Definition (Deterministic LTS) A state $s \in S$ is called *deterministic* iff

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

i.e., from s with trace σ only *one* state can be reached. An equivalent characterisation is: $\forall \sigma \in Act^* : |\underline{s \textbf{ after } \sigma}| \leq 1$ (Why?).
In LTS, there are essentially two sources of nondeterminism: one is *nondeterministic branching*, *i.e.,* when a state has two or more outgoing transitions labelled with the same action. The other one is nondeterminism caused by silent transitions. The effect is essentially the same: when a state is nondeterministic, it is not possible anymore to say which state will be reached with a given trace.

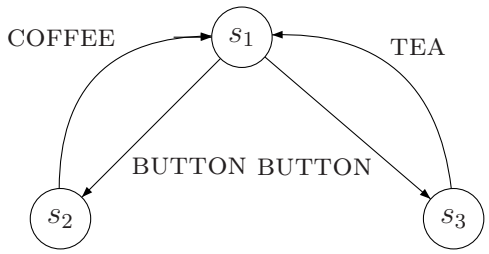
Note: All material on the slides used in the lecture is in the script.

1.2.9 Example (Coffee 2, slides #3, #4)
Here are two examples which demonstrate the effect of nondeterminism.

1. See Example 1.2.2: no state is deterministic:
   (a) $\underline{s_1 \textbf{ after } \text{BUTTON} \cdot \text{COFFEE}} = \{s_3, s_1\}$
   (b) $\underline{s_2 \textbf{ after } \text{TEA}} = \{s_4, s_1\}$
   (c) $\underline{s_3 \textbf{ after } \varepsilon} = \{s_3, s_1\}$
   (d) $\underline{s_4 \textbf{ after } \text{BUTTON} \cdot \text{TEA}} = \{s_4, s_1\}$

   Nondeterminism is here solely caused by the τ-transitions.

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



Again, no state is deterministic:

1. $\underline{s_1 \textbf{ after } \text{BUTTON}} = \{s_2, s_3\}$
2. $\underline{s_1 \textbf{ after } \text{BUTTON} \cdot \text{TEA}} = \{s_1\}$. But this does not mean that s_1 is is deterministic, because of 1.
3. $\underline{s_2 \textbf{ after } \text{COFFEE} \cdot \text{BUTTON}} = \{s_2, s_3\}$

■ END EXAMPLE

There is an extension of this classification of nondeterminism, which should be mentioned here: *internal* nondeterminism, and *external* nondeterminism. Internal nondeterminism covers exactly the two types of nondeterminism that we have mentioned before, caused by nondeterministic branching and internal actions. It is called internal because the cause of the nondeterminism is caused entirely by the internal structure of the considered LTS.
External nondeterminism occurs, if the LTS is considered a reactive system and the observable actions as being triggered from the environment. If a state has two outgoing transitions with different observable actions, it is from the point-of-view of the LTS not clear which of the two actions will be chosen by the environment to be executed.
Definition 1.2.8 is describing only internal nondeterminism, and this is what we shall be concerned with for most of this lecture.

Note: All material on the slides used in the lecture is in the script.

1.3 A Language to describe LTS

Labelled transition systems, despite their undisputed usefulness, are a bit difficult to handle. Especially if an LTS is supposed to describe the behaviour of concurrent processes, it is near impossible to define it manually. In this section we will introduce a small language, accompanied by an LTS semantics, which will greatly ease this task.

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*.

1.3.1 Definition (Processes)

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

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

   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 $P \in \mathcal{P}$.

Process variables are used as placeholders for process expressions. The process definitions determine, what process a process variable stands for. The other types of processes have the following meaning.

STOP**:** is the process that does nothing. This process shows no observable behaviour. It is deadlocked.

Note: All material on the slides used in the lecture is in the script.

$a.p$: is the process that can execute action a and then behaves like process p. a is either observable or $a = \tau$. $a.p$ is called the prefix operator (actually: for every $a \in Act_\tau$ there is one distinct prefix operator, $a.\cdot$).

$p + q$: if $p, q \in \mathbb{P}$, then $p + q$ behaves either like process p or like process q. $p + q$ describes thus an nondeterministic choice (internal or external). $+$ is called the *choice operator*.

$p\|_A q$: if $p, q \in \mathbb{P}$, then $p\|_A q$ behaves like the parallel execution of p and q, where actions in A can only be executed in synchrony. $\|_A$ is called the parallel (composition) operator, where A is the *synchronisation set*.

The parallel operator is the reason why this small language is actually very powerful: is is possible to specify concurrent processes independently from each other, and then combine them using the parallel operator.
The behaviour of a process can be described by an LTS. The next example shows how this works. Note that the processes are now used also as states, *i.e.,* $\mathbb{P}$ is the set of states of the LTS that we consider.

1.3.2 Example (Some simple processes)
Let $Act = \{a, b, c, d\}$.

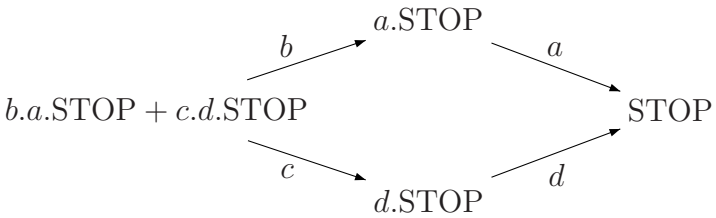
- The process a.STOP can execute action a and then nothing anymore. LTS:

$$a.\text{STOP} \xrightarrow{a} \text{STOP}$$

- The process $b.a$.STOP can execute action b, then a and then nothing anymore.
  LTS:

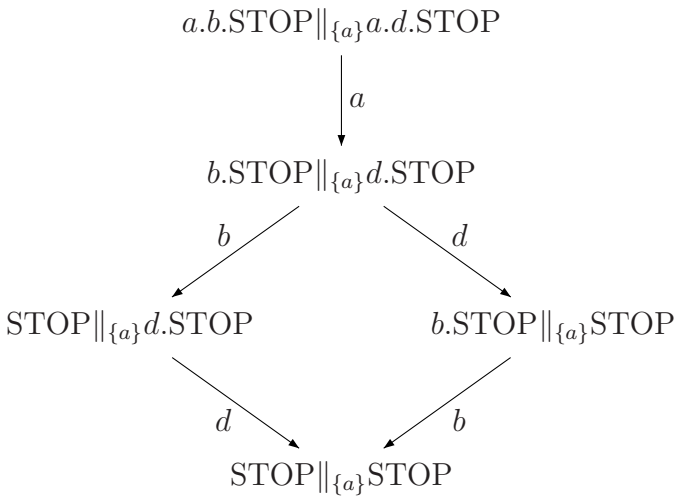
$$b.a.\text{STOP} \xrightarrow{b} a.\text{STOP} \xrightarrow{a} \text{STOP}$$

- The process $b.a.\text{STOP} + c.d.\text{STOP}$ can either execute action b and then a, or action c and then d.
  LTS:



- The process $a.b.\text{STOP}\|_{\{a\}} a.d.\text{STOP}$ can execute action a, and then action b and d, or action d and then b.

Note: All material on the slides used in the lecture is in the script.



■ END EXAMPLE

The previous example shows only states which eventually end up in a deadlock state. However, we would like to be able to define processes which describe non-terminating behaviour. For this purposes the process variables are introduced: they allow us to define recursive processes, which show generally nonterminating behaviour.

1.3.3 Example (Recursion)

1. Reconsider Example 1.2.2. We can describe this LTS as process $X \in \mathcal{P}$ with

$$\begin{aligned} X &\hat{=} \text{BUTTON}.Y \\ Y &\hat{=} \text{COFFEE}.\tau.X + \text{TEA}.\tau.X \end{aligned}$$

2. Reconsider Example 1.2.9. We can describe this coffee machine as process X with

$$X \hat{=} \text{BUTTON.COFFEE}.X + \text{BUTTON.TEA}.X$$

■ END EXAMPLE

End of Lecture #1

Note: All material on the slides used in the lecture is in the script.