Modelling

Lecture 2

27-01-2026

VALIDATION

Are we building the right (correct) program?

⇕

Does the design faithfully capture the requirements?

VERIFICATION

Are we building the program right (correctly)?

⇕

Does the design satisfy the

Labeled Transition Systems

STATE

snapshot of the system "at a given time"

TRANSITION

evolution step of the system "in time"

$$s \xrightarrow{\alpha} s'$$

s: state; α: label (transition step); s': state

LTS — Labeled transition system

Kripke structure

Transition System $TS = \langle S, Act, \to, I, AP, L \rangle$

where:

- S is a set of states
- Act is a set of actions;
- $\to \subseteq S \times Act \times S$ is a transition relation
- $I \subseteq S$ a set of initial states
- AP is a set of atomic propositions
- $L : S \to 2^{AP}$ a labeling function

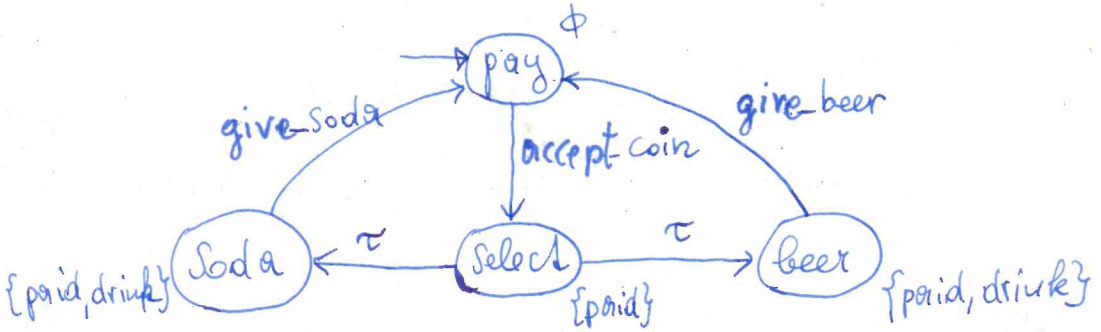
(2^{AP}: powerset of AP)

$2^{AP} = \{A \mid A \subseteq AP\}$

If Φ is a formula of propositional logic over at. prop' in AP, then we write

$$s \models \Phi \;:\Leftrightarrow\; L(s) \models \Phi.$$

Example. Beverage Vending Machine



$S = \{pay, select, soda, beer\}$

$I = \{pay\}$

$Act = \{accept\_coin, \tau, get\_soda, get\_beer\}$

$\rightarrow = \{\langle pay, accept\_coin, select\rangle, \langle select, \tau, soda\rangle, \langle select, \tau, beer\rangle,$
$\langle soda, give\_soda, pay\rangle, \langle beer, give\_beer, pay\rangle\}$

$pay \xrightarrow{accept\_coin} select$, $select \xrightarrow{\tau} soda$, $select \xrightarrow{\tau} beer$, ...

$AP = \{paid, drink\}$

$L: S \to 2^{AP}$ $\quad pay \mapsto \Phi$

$soda, beer \mapsto \{paid, drink\}$

$select \mapsto \{paid\}$

In general $L(s) := \{s\}$ is a simple reasonable choice for a Labelling

But e.g. if we only want to check properties that do not refer to the drink in question, like:

"The vending machine only delivers a drink after a coin is inserted"

then the above labeling is fine.

Therefore in general: $L(s) := \{s\} \cap AP$.

Later in LTL:
$\square$ (drink $\rightarrow$ paid)

Direct predecessors and Successors

$TS = (S, Act, \to, I, AP, L)$

For all $s \in S$ and $\alpha \in Act$:

$Post(s, \alpha) := \{s' \in S \mid s \xrightarrow{\alpha} s'\}$, ($\alpha$-successors of s)

$Post(s) := \bigcup_{\alpha \in Act} Post(s, \alpha)$, (successors of s)

$Pre(s, \alpha) := \{\tilde{s} \in S \mid \tilde{s} \xrightarrow{\alpha} s\}$, ($\alpha$-predecessors of s)

$Pre(s) := \bigcup_{\alpha \in Act} Pre(s, \alpha)$. (predecessors of s)

Extensions to sets C of states:

For all $C \subseteq S$ and $\alpha \in Act$:

$Post(C, \alpha) := \bigcup_{s \in C} Post(s, \alpha)$,

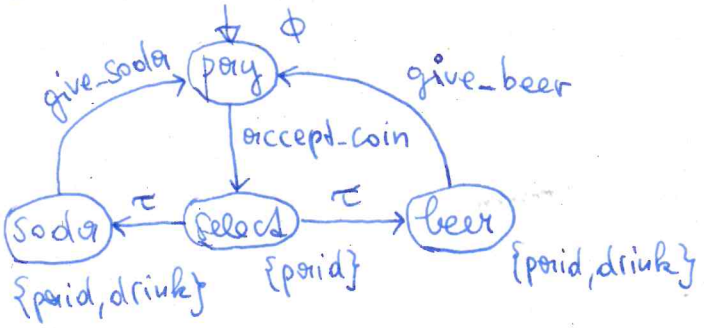
$Post(C) := \bigcup_{\alpha \in Act} Post(C, \alpha)$,

$Pre(C, \alpha) := \bigcup_{s \in C} Pre(s, \alpha)$,

$Pre(C) := \bigcup_{\alpha \in Act} Pre(C, \alpha)$.

Terminal State:

$s \in S$ is called terminal : $\Leftrightarrow$ $Post(s) = \emptyset$.



No terminal states

$Post(select) = \{soda, beer\}$

$= Pre(pay)$

$Pre(pay, give\_beer) = \{beer\}$

neither action- nor AP-deterministic

$TS = (S, Act, \rightarrow, I, AP, L)$ where $\rightarrow \subseteq S \times Act \times S$,
$I \subseteq S$ (usually $I \neq \emptyset$)
$L: S \rightarrow 2^{AP}$ (usually $AP \subseteq S$ and often $L(s) = \{s\}$)

Executions and Traces

Execution fragments are sequences $\rho \in S(Act\,S)^* \cup S(Act\,S)^\omega$ such that

if $\rho = s_0 a_1 s_1 a_2 s_2 a_3 s_3 \ldots a_n s_n \ldots \Rightarrow s_i \xrightarrow{a_{i+1}} s_i$ for all i

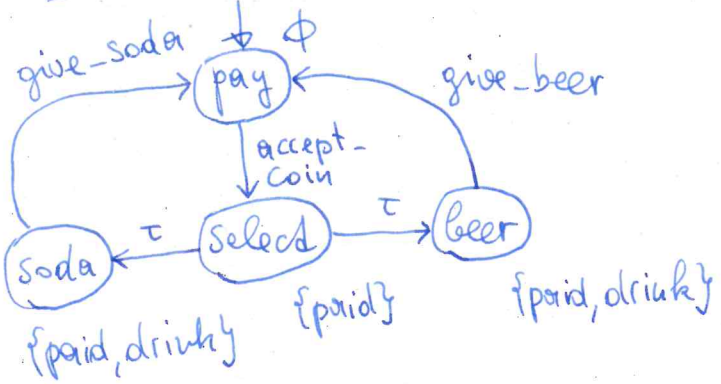
ρ is initial if $s_0 \in I$

ρ is maximal if { ρ is infinite or $\rho = q_0 a_1 q_1 a_2 q_3 \ldots a_n q_n$ and $Post(q_n) = \emptyset$

An execution is an initial and maximal execution fragment.

$Reach(TS) = \{ s \in S \mid \exists s_0 \in I \; \exists \rho$ (initial) execution fragment (ρ starts in s_0 and ends in s) $\}$

reachable states of TS



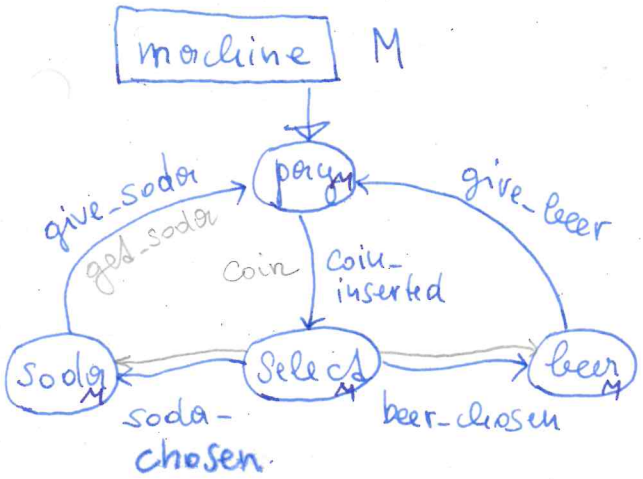
A trace of an execution fragment ρ
$\rho = q_0 a_1 q_1 a_2 q_2 a_3 q_3 \ldots$
is the sequence
$\tau = L(q_0) L(q_1) L(q_2) \ldots$

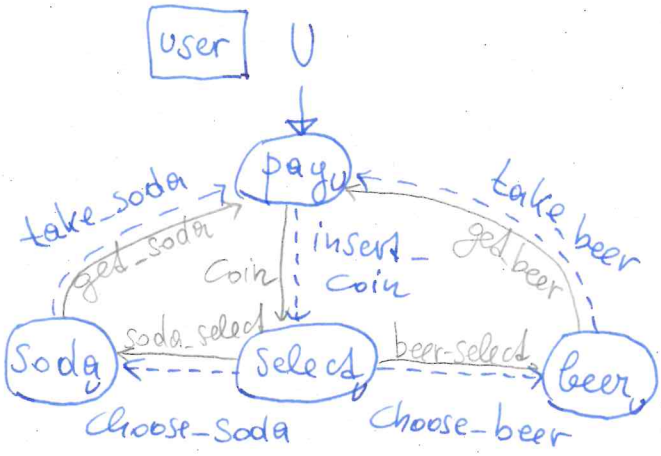
initial execution fragment π

$\langle pay, a\_c, select, \tau, beer, g\_b, pay \rangle$

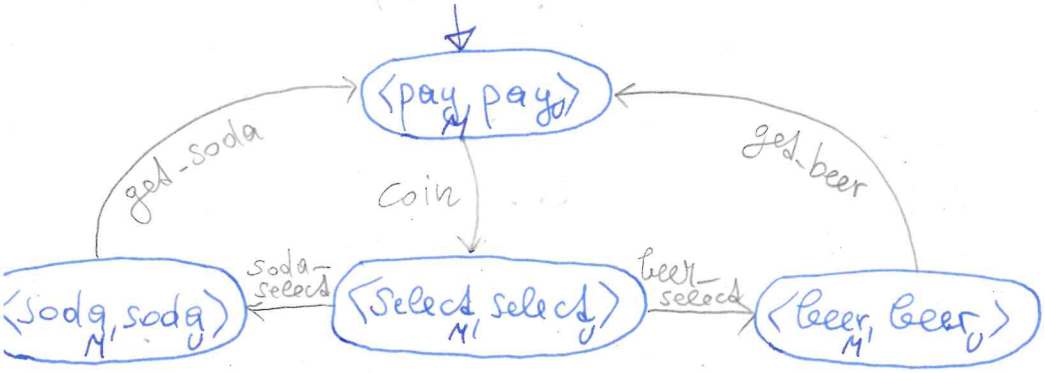
trace of π

$\langle \emptyset, \{paid\}, \{paid, drink\}, \emptyset \rangle$





$M \|_H U$ where $H = \{coin, soda\_select, beer\_select, get\_soda, get\_beer\}$



via handshaking Communication

(if $a \in H$)
$$\frac{S_1 \xrightarrow{a}_M S_1' \qquad S_2 \xrightarrow{a}_U S_2'}{\langle S_1, S_1 \rangle \xrightarrow[M \| U]{\#} \langle S_1', S_2' \rangle}$$

"Why do we need both actions and Labelling to express properties?"

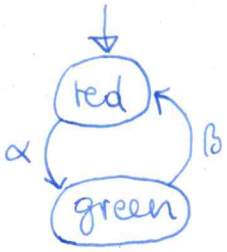
Verification can be { action-based / state-based / action + state-based (most-complex, often too complex)

Execution-fragments are used for action-based verification, which is essential for modelling communication.

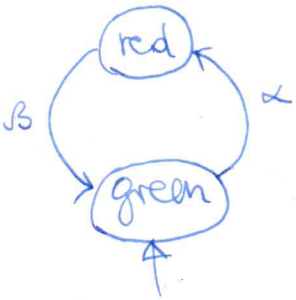
Notation Given a sequence $\sigma = \sigma_0 \sigma_1 \sigma_2 \ldots \in A^* \cup A^\omega$

- the length of σ is $|\sigma|$ (if σ is infinite, then $|\sigma| = \infty$)
- the i-th element of σ is $\sigma[i]$
- the last element of σ is $last(\sigma)$, provided that $\sigma \in A^*$ and hence is finite

TrLight1



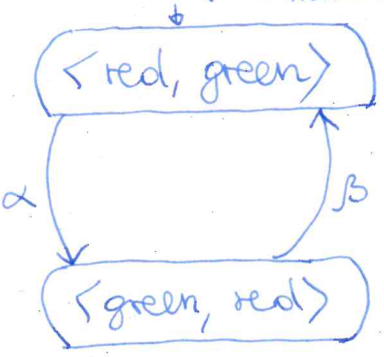
TrLight2



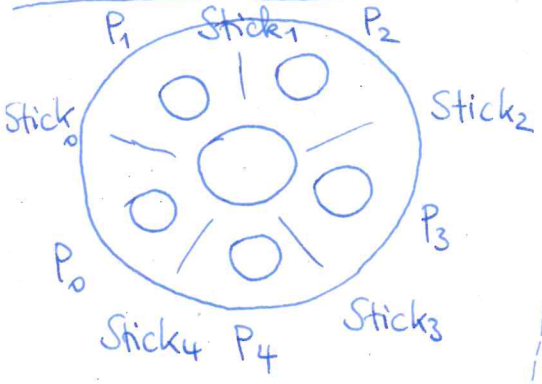
Traffic Lights

4

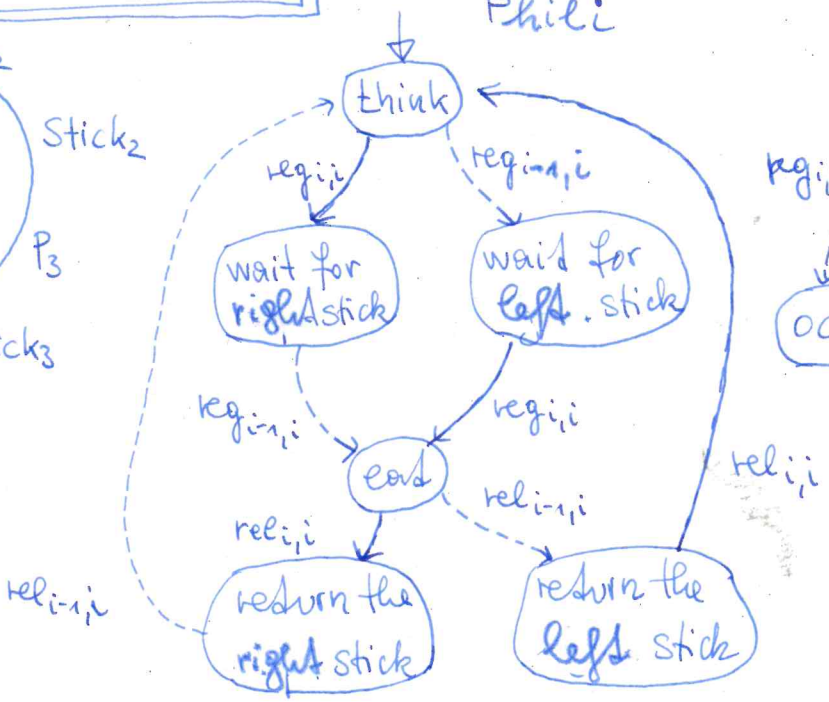
TrLight1 || TrLight2
H
with H = {α, β} for hand-shaking



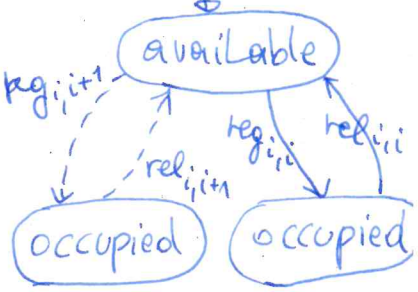
Dining Philosophers



Phil i



Stick i

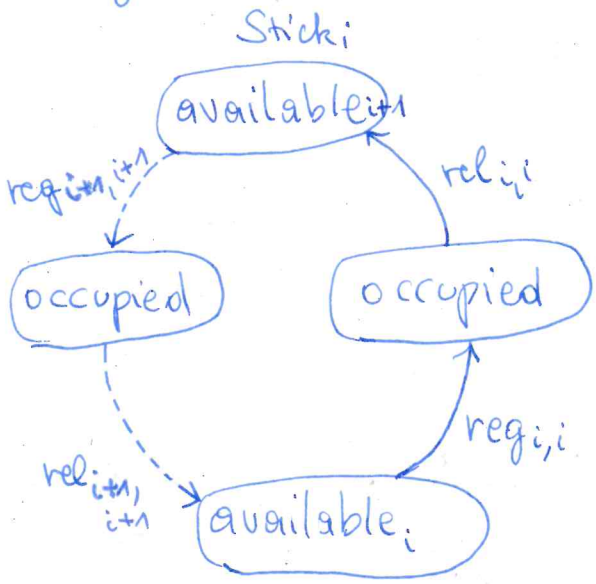


$Phil_4 \,||\, Stick_3 \,||\, Phil_3 \,||\, Stick_2 \,||\, Phil_2 \,||\, Stick_1 \,||\, Phil_1 \,||\, Stick_0 \,||\, Phil_0 \,||\, Stick_4$

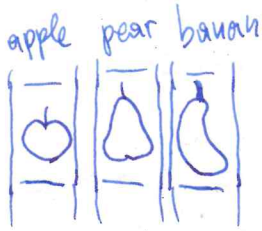
permits deadlock (e.g. if every philosopher picks up her right stick immediately)

Possibility to avoid deadlock:

- make a stick only available for one philosopher at a time
- change the stick-initial state for the 0, 2,



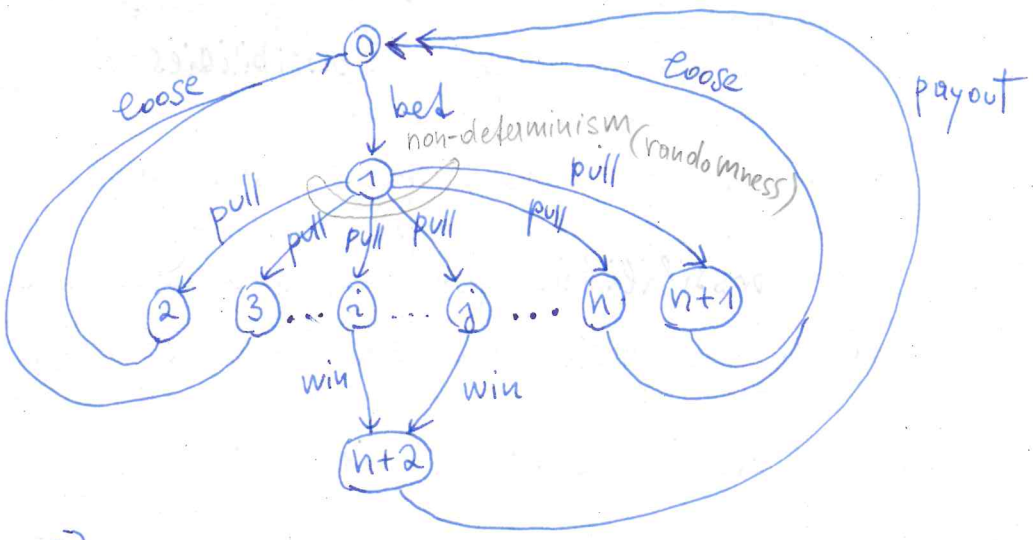
Example. A (simplified) 3-wheel slot machine



3 wheels

set of fruits

Fruits



$S = \{0, 1, 2, \dots, n+2\}$

$Act = \{bet, pull, loose, win, payout\}$

For an interval $W = [i, j]$ with $2 \le i \le j \le n+1$

$\longrightarrow = \{\langle 0, bet, 1\rangle\} \cup \{1\} \times \{pull\} \times [2, n+1] \cup \{\langle n+2, payout, 0\rangle\}$

$\cup\ W \times \{win\} \times \{n+2\} \cup (S \setminus W) \times \{loose\} \times \{0\}$

3-wheels machine

$n \le \max\{p(s) / s \in W\}$

$AP = \bigcup_{i=1}^{3} \{w_i = f \mid f \in Fruits\} \cup \bigcup_{n \in W} \{price = n / n \in \mathbb{N}\}$

where $Fruits = \{apple, pear, banana, \dots\}$

$L : S \to 2^{AP}$

$$s \longmapsto \begin{cases} \bigcup_{k=1}^{3} \{w_k = c_k(s)\} & \dots\ s \in \{2, \dots, n+1\} \setminus W \\ \{price = p(s)\} \cup \bigcup_{k=1}^{3} \{w_k = c_k(s)\} & \dots\ s \in W \\ \emptyset & \dots\ s \in \{0, n+2\} \end{cases}$$

where $c_1, c_2, c_3 : \{2, \dots, n+1\} \to Fruits$ — functions that assign to states $2, \dots, n+1$ the 3 fruits that identify the state

$p : W \subseteq S \to \mathbb{N}$ — prize function

such that

$\{\langle c_1(s), c_2(s), c_3(s)\rangle / s \in \{2, \dots, n+1\}\} = Fruits \times Fruits \times Fruits = Fruits^3$

Non-Determinism

crucial modeling concept

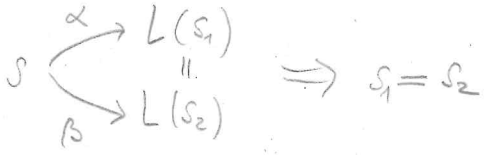
- for modeling interleaving of processes (wanting to study all possibilities how parallel processes can execute in time)
- for abstraction purposes, e.g. underspecification (leaving room for how implementations can resolve different possibilities for intended executions)
- for interaction with an unknown environment

$TS = (S, Act, \to, I, AP, L)$ is

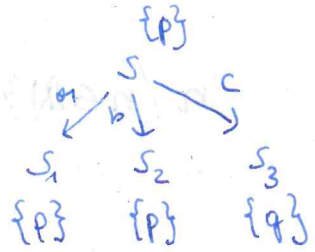
action-deterministic $:\Leftrightarrow |I| \leq 1$, and $\forall s \in S, \alpha \in Act: |Post(s,\alpha)| \leq 1$

AP-deterministic: $\Leftrightarrow$ $|I| \leq 1$, and

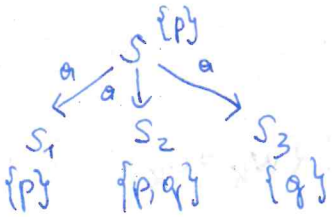
$\forall s \in S \; \forall A \in 2^{AP}: |\{s' \in Post(s) \mid L(s') = A\}| \leq 1$



Example.



action-deterministic ✓
but not AP-deterministic



not action-deterministic
but AP-deterministic ✓

- interleaving for $\alpha \notin H$:

$$\frac{s_1 \xrightarrow{\alpha}_1 s_1'}{\langle s_1, s_2\rangle \xrightarrow{\alpha} \langle s_1', s_2\rangle} \qquad \frac{s_2 \xrightarrow{\alpha}_2 s_2'}{\langle s_1, s_2\rangle \xrightarrow{\alpha} \langle s_1, s_2'\rangle}$$

- handshaking for $\alpha \in H$:

$$\frac{s_1 \xrightarrow{\alpha}_1 s_1' \quad \wedge \quad s_2 \xrightarrow{\alpha}_2 s_2'}{\langle s_1, s_2\rangle \xrightarrow{\alpha} \langle s_1', s_2'\rangle}$$

Figure 2.11: Rules for handshaking.

whereas according to the latter type processes "communicate" via shared variables. In this subsection, we consider a mechanism by which concurrent processes interact via handshaking. The term "handshaking" means that concurrent processes that want to interact have to do this in a *synchronous* fashion. That is to say, processes can interact only if they are both participating in this interaction at the same time—they "shake hands".

Information that is exchanged during handshaking can be of various nature, ranging from the value of a simple integer, to complex data structures such as arrays or records. In the sequel, we do not dwell upon the content of the exchanged messages. Instead, an abstract view is adopted and only communication (also called synchronization) actions are considered that represent the occurrence of a handshake and not the content.

To do so, a set H of *handshake actions* is distinguished with $\tau \notin H$. Only if both participating processes are ready to execute the same handshake action, can message passing take place. All actions outside H (i.e., actions in $Act \setminus H$) are independent and therefore can be executed autonomously in an interleaved fashion.

Definition 2.26. Handshaking (Synchronous Message Passing)

Let $TS_i = (S_i, Act_i, \rightarrow_i, I_i, AP_i, L_i)$, i=1, 2 be transition systems and $H \subseteq Act_1 \cap Act_2$ with $\tau \notin H$. The transition system $TS_1 \parallel_H TS_2$ is defined as follows:

$$TS_1 \parallel_H TS_2 \;=\; (S_1 \times S_2, Act_1 \cup Act_2, \rightarrow, I_1 \times I_2, AP_1 \cup AP_2, L)$$

where $L(\langle s_1, s_2\rangle) \;=\; L_1(s_1) \cup L_2(s_2)$, and where the transition relation $\rightarrow$ is defined by the rules shown in Figure 2.11. ■

Notation: $TS_1 \parallel TS_2$ abbreviates $TS_1 \parallel_H TS_2$ for $H = Act_1 \cap Act_2$.

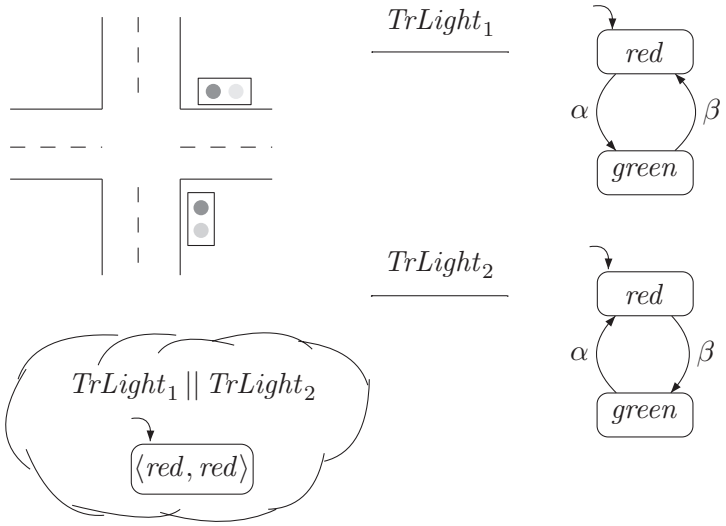


Figure 3.1: An example of a deadlock situation.

Example 3.1. Deadlock for Fault Designed Traffic Lights

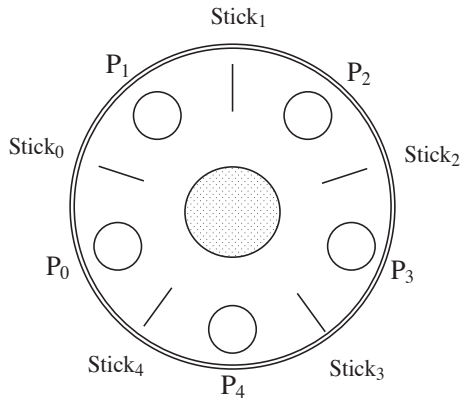
Consider the parallel composition of two transition systems

$$TrLight_1 \parallel TrLight_2$$

modeling the traffic lights of two intersecting roads. Both traffic lights synchronize by means of the actions α and β that indicate the change of light (see Figure 3.1). The apparently trivial error to let both traffic lights start with a red light results in a deadlock. While the first traffic light is waiting to be synchronized on action α, the second traffic light is blocked, since it is waiting to be synchronized with action β. ■

Example 3.2. Dining Philosophers

This example, originated by Dijkstra, is one of the most prominent examples in the field of concurrent systems.



Five philosophers are sitting at a round table with a bowl of rice in the middle. For the philosophers (being a little unworldly) life consists of thinking and eating (and waiting, as we will see). To take some rice out of the bowl, a philosopher needs two chopsticks. In between two neighboring philosophers, however, there is only a single chopstick. Thus, at any time only one of two neighboring philosophers can eat. Of course, the use of the chopsticks is exclusive and eating with hands is forbidden.

Note that a deadlock scenario occurs when all philosophers possess a single chopstick. The problem is to design a protocol for the philosophers, such that the complete system is deadlock-free, i.e., at least one philosopher can eat and think infinitely often. Additionally, a fair solution may be required with each philosopher being able to think and eat infinitely often. The latter characteristic is called freedom of *individual starvation*.

The following obvious design cannot ensure deadlock freedom. Assume the philosophers and the chopsticks are numbered from 0 to 4. Furthermore, assume all following calculations be "modulo 5", e.g., chopstick $i-1$ for i=0 denotes chopstick 4, and so on.

Philosopher i has stick i on his left and stick $i-1$ on his right side. The action $request_{i,i}$ express that stick i is picked up by philosopher i. Accordingly, $request_{i-1,i}$ denotes the action by means of which philosopher i picks up the $(i-1)$th stick. The actions $release_{i,i}$ and $release_{i-1,i}$ have a corresponding meaning.

The behavior of philosopher i (called process $Phil_i$) is specified by the transition system depicted in the left part of Figure 3.2. Solid arrows depict the synchronizations with the i-th stick, dashed arrows refer to communications with the $i-1$th stick. The sticks are modeled as independent processes (called $Stick_i$) with which the philosophers synchronize via actions *request* and *release*; see the right part of Figure 3.2 that represents the process of stick i. A stick process prevents philosopher i from picking up the ith stick when philosopher $i+1$ is using it.

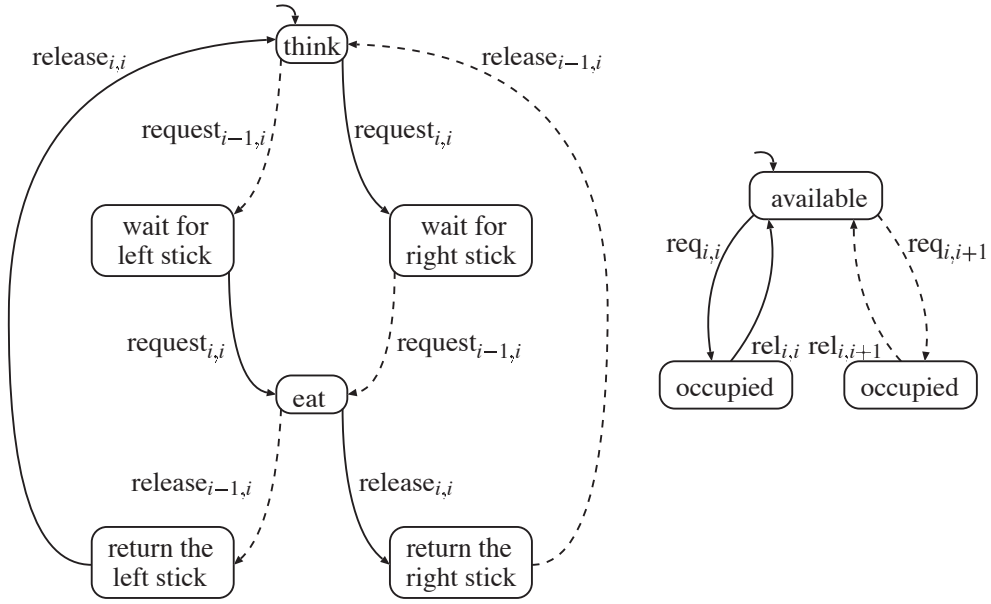


Figure 3.2: Transition systems for the ith philosopher and the ith stick.

The complete system is of the form:

$$\mathit{Phil}_4 \,\|\, \mathit{Stick}_3 \,\|\, \mathit{Phil}_3 \,\|\, \mathit{Stick}_2 \,\|\, \mathit{Phil}_2 \,\|\, \mathit{Stick}_1 \,\|\, \mathit{Phil}_1 \,\|\, \mathit{Stick}_0 \,\|\, \mathit{Phil}_0 \,\|\, \mathit{Stick}_4$$

This (initially obvious) design leads to a deadlock situation, e.g., if all philosophers pick up their left stick at the same time. A corresponding execution leads from the initial state

$$\langle \mathit{think}_4, \mathit{avail}_3, \mathit{think}_3, \mathit{avail}_2, \mathit{think}_2, \mathit{avail}_1, \mathit{think}_1, \mathit{avail}_0, \mathit{think}_0, \mathit{avail}_4 \rangle$$

by means of the action sequence $\mathit{request}_4$, $\mathit{request}_3$, $\mathit{request}_2$, $\mathit{request}_1$, $\mathit{request}_0$ (or any other permutation of these 5 request actions) to the terminal state

$$\langle \mathit{wait}_{4,0}, \mathit{occ}_{4,4}, \mathit{wait}_{3,4}, \mathit{occ}_{3,3}, \mathit{wait}_{2,3}, \mathit{occ}_{2,2}, \mathit{wait}_{1,2}, \mathit{occ}_{1,1}, \mathit{wait}_{0,1}, \mathit{occ}_{0,0} \rangle.$$

This terminal state represents a deadlock with each philosopher waiting for the needed stick to be released.

A possible solution to this problem is to make the sticks available for only one philosopher at a time. The corresponding chopstick process is depicted in the right part of Figure 3.3. In state $\mathit{available}_{i,j}$ only philosopher j is allowed to pick up the ith stick. The above-mentioned deadlock situation can be avoided by the fact that some sticks (e.g., the first, the third, and the fifth stick) start in state $\mathit{available}_{i,i}$, while the remaining sticks start in state $\mathit{available}_{i,i+1}$. It can be verified that this solution is deadlock- and starvation-free.

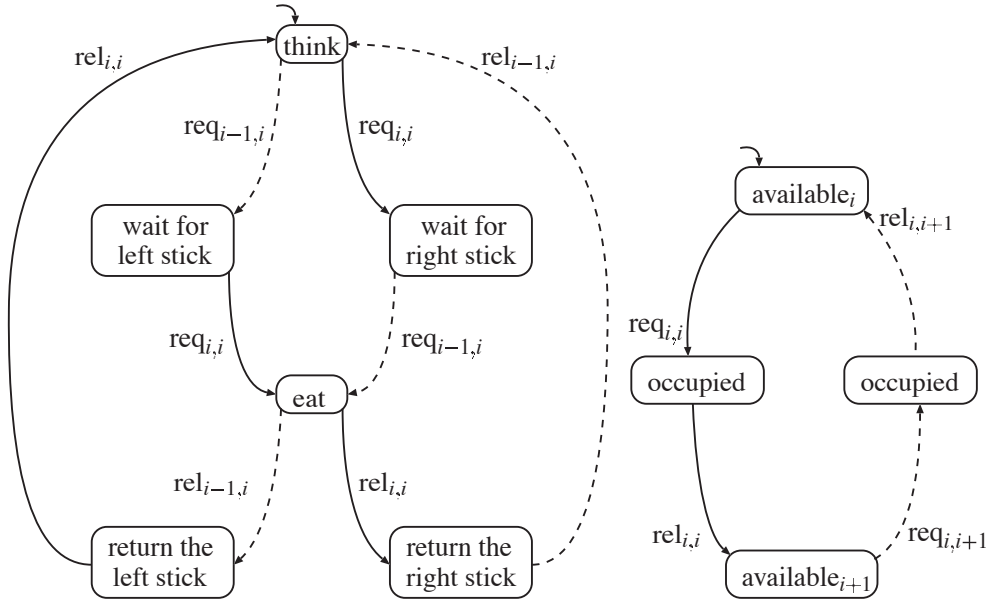


Figure 3.3: Improved variant of the ith philosopher and the ith stick.

A further characteristic often required for concurrent systems is robustness against failure of their components. In the case of the dining philosophers, robustness can be formulated in a way that ensures deadlock and starvation freedom even if one of the philosophers is "defective" (i.e., does not leave the think phase anymore).[1] The above-sketched deadlock- and starvation-free solution can be modified to a fault-tolerant solution by changing the transition systems of philosophers and sticks such that philosopher $i{+}1$ can pick up the ith stick even if philosopher i is thinking (i.e., does not need stick i) independent of whether stick i is in state $\mathit{available}_{i,i}$ or $\mathit{available}_{i,i+1}$. The corresponding is also true when the roles of philosopher i and $i{+}1$ are reversed. This can be established by adding a single Boolean variable x_i to philosopher i (see Figure 3.4). The variable x_i informs the neighboring philosophers about the current location of philosopher i. In the indicated sketch, x_i is a Boolean variable which is true if and only if the ith philosopher is thinking. Stick i is made available to philosopher i if stick i is in location $\mathit{available}_i$ (as before), or if stick i is in location $\mathit{available}_{i+1}$ while philosopher $i{+}1$ is thinking.

Note that the above description is at the level of program graphs. The complete system is a channel system with *request* and *release* actions standing for handshaking over a channel of capacity 0. ■

[1] Formally, we add a loop to the transition system of a defective philosopher at state think_i.

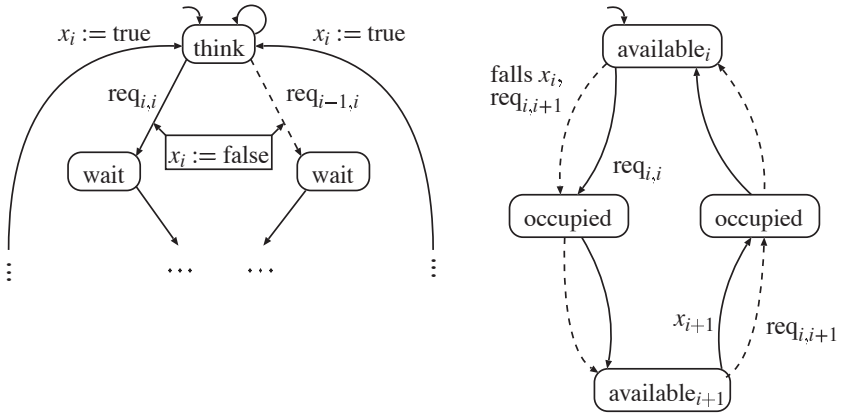


Figure 3.4: Fault-tolerant variant of the dining philosophers.

3.2 Linear-Time Behavior

To analyze a computer system represented by a transition system, either an action-based or a state-based approach can be followed. The state-based approach abstracts from actions; instead, only labels in the state sequences are taken into consideration. In contrast, the action-based view abstracts from states and refers only to the action labels of the transitions. (A combined action- and state-based view is possible, but leads to more involved definitions and concepts. For this reason it is common practice to abstract from either action or state labels.) Most of the existing specification formalisms and associated verification methods can be formulated in a corresponding way for both perspectives.

In this chapter, we mainly focus on the state-based approach. Action labels of transitions are only necessary for modeling communication; thus, they are of no relevance in the following chapters. Instead, we use the atomic propositions of the states to formulate system properties. Therefore, the verification algorithms operate on the *state graph* of a transition system, the digraph originating from a transition system by abstracting from action labels.