Chapter 9
Labelled Transition Systems

Systems and Processes

Remember: Abstractly, what is a Process?
• Processes are subsets of the events occurring in a system.
• In a sequential process, the events are fully ordered in time.

Therefore:
• A system specification is decomposed into process specifications.
• A system implementation is composed from process implementations.

Processes, Actions, Events

• A process is a subset of the events occurring in a system.
• The simplest possible process: empty set of events, called STOP.
• More interesting processes have events, which can also be interpreted as actions.
• We assume that all actions can be decomposed into atomic actions.
• In a system, each event belongs to at least one process.
• Events can be shared between processes — several processes can together engage in a single action.

System Composition

• A system specification is decomposed into process specifications.
• A system implementation is composed from process implementations.

• Sequential composition: every event in $P_1$ occurs before every event in $P_2$.
• Concurrent composition: No such clear ordering imposed a priori.
• Sequential processes are basic building blocks.
**Processes and State**

- Processes perform **state transitions** — in different states, a process will be able to engage in different sets of actions.
  
  — After some action, the set of possible continuing actions may be different from before.

- Atomic actions induce **indivisible state changes**.

- A system composed of several processes has a state that is composed from the states of the individual processes.

---

**Labelled Transition Systems (LTSs)**

**Definition:** A labelled transition system \((S, s_0, L, \delta)\) consists of

- a set \(S\) of **states**
- an initial state \(s_0 : S\)
- a set \(L\) of **action labels**
- a transition relation \(\delta : \mathcal{P}(S \times L \times S)\).

**Example:**

\[
\text{LightSwitch}_1 = (\{\text{dark}, \text{light}\}, \text{dark}, \{\text{on, off}\}, \\
\{(\text{dark, on, light}), (\text{light, off, dark})\})
\]

---

**Another LTS …**

\[
\text{LightSwitch}_1 = (\{\text{dark, light}\}, \text{dark}, \{\text{on, off}\}, \\
\{(\text{dark, on, light}), (\text{light, off, dark})\})
\]

Different, but **isomorphic**, where the isomorphism preserves action labels and the transition relation.

— The identity of the states does not matter.

---

**Traces**

**Definition:** A **trace** of an LTS is a sequence (finite or infinite) of action labels that results from a maximal path (with respect to the prefix ordering) starting at the initial state.

**Example:**

- Sequences of action labels that result from finite paths starting at the initial state:
  - on
  - on, off
  - on, off, on
  - on, off, on, off

- \(\text{LightSwitch}_1\) has only one infinite trace:
  - on, off, on, off, on, off, ...

- \(\text{LightSwitch}_2\) has the same set of traces as \(\text{LightSwitch}_1\) — they are **behaviourally equivalent**.
Concurrent Composition

- A system composed of several processes has a state that is composed from the states of the individual processes.

Converse =

\[ \begin{array}{c}
0 \\
\text{think} \\
\downarrow \\
1 \\
\text{talk} \\
\downarrow \\
2
\end{array} \]

Itch =

\[ \begin{array}{c}
a \\
\text{scratch} \\
\downarrow \\
b
\end{array} \]

Converse \parallel Itch =

While Converse and Itch have only one trace each, their composition has three, representing arbitrary interleaving.

Shared Actions

Bill =

\[ \begin{array}{c}
0 \\
\text{play} \\
\downarrow \\
1 \\
\text{meet} \\
\downarrow \\
2
\end{array} \]

Ben =

\[ \begin{array}{c}
a \\
\text{work} \\
\downarrow \\
b \\
\text{meet} \\
\downarrow \\
c
\end{array} \]

In the composition Bill \parallel Ben,

- play and work are concurrent actions — the order in which they are observed does not matter.
- The shared action meet synchronizes the execution of the two constituent processes.
- Traces of the composition: play, work, meet
  work, play, meet

Concurrent Composition of LTSs

Definition: For \( P_1 = (S_1, s_1, L_1, \delta_1) \) and \( P_2 = (S_2, s_2, L_2, \delta_2) \), the concurrent composition \( P_1 \parallel P_2 \) is the LTS

\[ (S_1 \times S_2, (s_1, s_2), L_1 \cup L_2, \delta) \]

where

\[ ((x_1, x_2), (y_1, y_2)) \in \delta \]

\[ \iff \begin{cases} (x_1, a, y_1) \in \delta_1 & \land \ x_2 = y_2 & \land \ a \in L_1 - L_2 \\ x_1 = y_1 & \land \ (x_2, a, y_2) \in \delta_2 & \land \ a \in L_2 - L_1 \\ (x_1, a, y_1) \in \delta_1 & \land \ (x_2, a, y_2) \in \delta_2 & \land \ a \in L_1 \cap L_2 \end{cases} \]
**Composition with Shared Actions**

Bill = 

Ben = 

Bill || Ben = 

Unreachable states do not influence the behaviour!

**Maker — User 2**

Maker = 

User = 

**Factory**

Factory = MakerA || MakerB || Assemble
Liveness and Safety Properties

A safety property asserts:
“something bad will never happen”

A liveness property asserts:
“something good will eventually happen”

How many traces do these processes have?

Deadlock

- **Deadlock** occurs in a system when all its constituent processes are blocked.
- A system is **deadlocked** if there are no actions it can perform.
- A **deadlock state** in an LTS is a reachable state with no outgoing transitions.
- An LTS has a deadlock state iff it has a **finite trace**.

- A terminating constituent process introduces “atypical” deadlock.
- “**Typical**” deadlocks occur in concurrent compositions of processes that individually are deadlock-free.

Safety

A safety property asserts:
“something bad will never happen”

Important safety conditions:

- **Partial correctness**
  - State predicate: If in a proper termination state, then postcondition is satisfied.
- **Invariants**
  - If in a certain kind of state, or before or after a certain kind of action, then the invariant holds for the current state.
- **Safe access sequences to resources**
  - Certain actions happen only conforming to a fixed pattern.

Such properties are often formulated using temporal logic.
Safe Access Sequences

- Given a system modelled as an LTS \( P = (S, s, L, \delta) \), accesses to some resource (set) involve actions of a subset \( A \subseteq L \).
- For every trace \( t \) of \( P \), only its projection on \( A \) is considered, i.e., the sequence of those elements of \( t \) that are in \( A \).
- These projections need to satisfy some predicate.
- **Conveniently:** These projections have to be traces of some (simpler) LTS

**Example:** \( SAFE = \text{command} \rightarrow \text{response} \rightarrow \text{SAFE} \)

Checking Safe Access Sequences using ||

\( SAFE = \text{command} \rightarrow \text{response} \rightarrow \text{SAFE} \)

Add catch-all error state:

\( SAFEcheck = \)

**Safety**

Ideally, a software system will be safe if it satisfies its specification.

— However, the specification may not guarantee safety.

Safety is a greater concern in a concurrent software system because the order of events is harder to control

**Fundamental Safety Failure:** An action by a process or thread that is intended to be atomic is breached by another process or thread.

- The code that implements the atomic action is called a critical section
- The breach of the atomic action may be unpredictable due to race conditions

**Liveness**

A liveness property asserts:

“no matter when we start to look, something good will eventually happen”

**Example:** “Philosopher \( i \) cannot starve at the table.”

- **No matter when we start to look, if philosopher \( i \) is at the table, he will eventually be eating**
- **This can be expressed in terms of traces:**

Philosopher \( \text{phil}.i \) “cannot starve at the table” \( \text{iff} \) for every trace \( t \) and every position \( m \) such that \( t_m = \text{phil}.i\.sitdown \) there is a position \( n \) with \( n > m \) such that \( t_n = \text{phil}.i\.eat \).
Liveness

Ideally, a software system will be live if it satisfies its specification.
— However, the specification may not guarantee liveness.

Fundamental Liveness Failure:

A process (thread) waits for an event that will never happen.

Examples:

- Deadlock
- Missed signals
- Nested monitor lockouts
- Livelock
- Starvation
- Resource exhaustion
- Distributed failure

Branching Transitions

A state of a process from which several transitions exist usually models one of the following:

- In this state, the process is prepared to react to different environmental stimuli
- In this state, the process acts by making a (non-deterministic) choice
  - non-determinism could be intended
  - non-determinism could be the result of abstraction

LTSs do not differentiate between action and reaction!

Reactive Choice

Sema3 = 

Lookup =

Server =

Active Non-Deterministic Choice

Client1 = request 1 → service 1 → sleep → Client1
Client2 = request 2 → service 2 → work → Client2
Clients = Client1 || Client2
System = Clients || Server

Concurrency is a good source of non-determinism!

Distribution is one of the best sources of non-determinism!
Modelling Real Non-Deterministic Choice

How should we model a process that repeatedly tosses a coin?

How should we model a process that bets on alternating outcomes?

\[ Bet = \begin{array}{c}
\text{heads} \\
H \\
\text{tails} \\
T
\end{array} \]

\[ Coin1 = \begin{array}{c}
\text{heads} \\
0 \\
\text{tails} \\
0
\end{array} \]

\[ Coin2 = \begin{array}{c}
\text{heads} \\
0 \\
\text{tails} \\
0
\end{array} \]

\[ Coin3 = \begin{array}{c}
\text{heads} \\
1 \\
\text{tails} \\
2
\end{array} \]

Consider the compositions with \( Bet \)!

Betting Introduces Deadlock

\[ Bet = \begin{array}{c}
\text{heads} \\
H \\
\text{tails} \\
T
\end{array} \]

\[ Coin3 = \begin{array}{c}
\text{heads} \\
1 \\
\text{tails} \\
2
\end{array} \]

Non-Deterministic Choice, Traces, and Composition

\[ Coin2 \parallel Bet \]

\[ Coin3 \parallel Bet = \]

\[ Coin2 = \begin{array}{c}
\text{heads} \\
0 \\
\text{tails} \\
0
\end{array} \]

\[ Coin3 = \begin{array}{c}
\text{heads} \\
1 \\
\text{tails} \\
2
\end{array} \]

\[ \Rightarrow \text{Two LTSs } P_1 \text{ and } P_2 \text{ are equivalent iff for every LTS } Q, \text{ the compositions } P_1 \parallel Q \text{ and } P_2 \parallel Q \text{ have the same trace set.} \]

This is a black-box view: “No context enables distinction.”
How Not to Model Signal Handling

0  M.acquire  1 block  2 M.use  3 unblock  4

M.release

block → a
deliver → c  H.release → d

unblock

finish

Modelling Signal Handling

0  M.acquire  1 block  2 M.use  3 unblock  4

deliver  M.release

finish

0s  deliver (finish)

4s  deliver (finish)

0a  M.acquire  1a  deliver  2a  M.use  3a  deliver  4a  M.release

0b  M.acquire  1b  deliver  2b  M.use  3b  deliver  4b  M.release

0c  M.acquire  1c  deliver  2c  M.use  3c  deliver  4c  M.release

0d  M.acquire  1d  deliver  2d  M.use  3d  deliver  4d  M.release
**Labelling and Sharing**

**Definition:** For an action label set \( L \) and a label set \( A \), we let \( A::L \) denote the following set of **labelled actions**:

\[
F::L = \{ f : F; q : L \cdot f.q \}
\]

For an LTSs \( P = (S, s_0, L, \delta) \), we define:

- The LTS \( P \) **labelled** with a label \( f \) is \( f:P = (S, s_0, \{f\}::L, \delta_f) \), where
  \[
  (x, a, y) \in \delta_f \iff \exists a_0 : L \cdot a = f.a_0 \land (x, a_0, y) \in \delta.
  \]

- The LTS \( P \) **shared** among a label set \( F \) is \( F::P = (S, s_0, F::L, \delta_F) \), where
  \[
  (x, a, y) \in \delta_F \iff \exists f : F; a_0 : L \cdot a = f.a_0 \land (x, a_0, y) \in \delta.
  \]

**Labelling: Switches**

\[
\text{Switch} = \begin{cases} 
0 & \text{on} \\
1 & \text{off}
\end{cases}
\]

\[
x:\text{Switch} = \begin{cases} 
0 & x.\text{on} \\
1 & x.\text{off}
\end{cases}
\]

\[
a:\text{Switch} || b:\text{Switch} = \begin{cases} 
00 & a.\text{on} \\
10 & a.\text{off} \\
01 & b.\text{on} \\
11 & b.\text{off}
\end{cases}
\]

**Sharing: Resources**

\[
\text{Resource} = \begin{cases} 
0 & \text{acquire} \\
1 & \text{release}
\end{cases}
\]

\[
\{a, b\}::\text{Resource} = \begin{cases} 
01 & a.\text{on} \\
11 & a.\text{off} \\
01 & b.\text{on} \\
11 & b.\text{off}
\end{cases}
\]
Sharing: Resources

\[ \text{Resource} = \]

\[ \{a, b\}::\text{Resource} = \]

Blocking Resources

\[ \text{ResBlocking} = \text{a:Abuser} \ || \ b:User \ || \ \{a, b\}::\text{Resource} \]

Sharing Resources

\[ \text{ResSharing} = \text{a:User} \ || \ b:User \ || \ \{a, b\}::\text{Resource} \]

Sharing a Labelled Resource

\[ \text{Resource} = \]

\[ \{a, b\}::\text{printer:Resource} = \]
An Alternative Way of Defining Primitive Processes

\[
\text{Resource} = \begin{array}{c}
0 \\
\text{acq} & \rightarrow & 1 \\
\text{printer.acq} & \rightarrow & 2 \\
\text{scanner.acq} & \rightarrow & 3 \\
\text{scanner.rel} & \rightarrow & 4 \\
\text{printer.rel} & \rightarrow & 1
\end{array}
\]

\[A = 0 \begin{array}{c}
1 \\
\text{printer.acq} & \rightarrow & 2 \\
\text{scanner.acq} & \rightarrow & 3 \\
\text{copy} & \rightarrow & 1
\end{array}
\]

Process Calculus Notation:

\[
\text{Resource} = \text{acq} \rightarrow \text{rel} \rightarrow \text{Resource}
\]

\[A = \text{printer.acq} \rightarrow \text{scanner.acq} \rightarrow \text{copy} \rightarrow \text{printer.rel} \rightarrow \text{scanner.rel} \rightarrow A
\]

Sharing Two Resources

\[
\text{Resource} = \text{acq} \rightarrow \text{rel} \rightarrow \text{Resource}
\]

\[A = \text{pr.acq} \rightarrow \text{sc.acq} \rightarrow \text{copy} \rightarrow \text{pr.rel} \rightarrow \text{sc.rel} \rightarrow A
\]

\[B = \text{sc.acq} \rightarrow \text{pr.acq} \rightarrow \text{copy} \rightarrow \text{sc.rel} \rightarrow \text{pr.rel} \rightarrow B
\]

\[S\text{ys} = a:A \parallel \{a, b\}::pr:\text{Resource} \parallel \{a, b\}::sc:\text{Resource} \parallel b:B
\]

The Dining Philosophers

- Five philosophers live together in a house.
- The live of a philosopher essentially consists of alternating phases of thinking and eating.
- For eating, there is a round table with five seats and a large bowl of spaghetti on it; between adjacent seats there is always one fork.
- Each philosopher needs two forks in order to be able to eat.
- When hungry, each philosopher will sit down on a free chair, take up the fork to his left, take up the fork to his right, eat, put down the forks, and leave for more thinking.
- Is it possible that the philosophers all starve to death?

Let \( N = 5 \)

Let \( \text{succ}_N(i) = (i + 1)\%N \)

\[
\text{Diners} = \bigparallel_{i=0}^{N-1} (\text{phil}.i:\text{Phil} \parallel \{\text{phil}.i.\text{right}, \text{phil}.\text{succ}_N(i).\text{left}\}::\text{Fork})
\]
Model-Checking the Dining Philosophers Using LTSA

\[ \text{PHIL} = (\text{sitdown} \rightarrow \text{right.get} \rightarrow \text{left.get} \rightarrow \text{eat} \rightarrow \text{left.put} \rightarrow \text{right.put} \rightarrow \text{arise} \rightarrow \text{PHIL}). \]

\[ \text{FORK} = (\text{get} \rightarrow \text{put} \rightarrow \text{FORK}). \]

\[ \text{DINERS (N=5)} = \forall [i:0..N-1] \]
\[ (\text{phil}[i]:\text{PHIL} \lor (\text{phil}[i].\text{right}, \text{phil}[(i+1)\%N].\text{left})::\text{FORK}). \]

Trace to DEADLOCK:
\[ \text{phil}.0.\text{sitdown} \]
\[ \text{phil}.0.\text{right.get} \]
\[ \text{phil}.1.\text{sitdown} \]
\[ \text{phil}.1.\text{right.get} \]
\[ \text{phil}.2.\text{sitdown} \]
\[ \text{phil}.2.\text{right.get} \]
\[ \text{phil}.3.\text{sitdown} \]
\[ \text{phil}.3.\text{right.get} \]
\[ \text{phil}.4.\text{sitdown} \]
\[ \text{phil}.4.\text{right.get} \]

Solutions to the Dining Philosophers Problem

Original solution: Introduce a butler who restricts the maximum number of sitting philosophers to 4.

\[ \text{Butler} = \begin{array}{c}
\text{sitdown} & \text{sitdown} & \text{sitdown} & \text{sitdown} \\
4 & 3 & 2 & 1 & 0
\end{array} \]

The butler is a counting semaphore!

Some other solutions:
- Have some philosophers pick up the left fork first.
- Make picking up both forks atomic.
- Have all philosophers decide randomly which fork to pick up, and give priority to “hungrier” neighbours.