M&K

Chapter 1 - Introduction

- Execution of a sequential program is called a *process* (1.1, 2.1).
- **Interleaved** execution: Two or more processes *taking turns* in getting time slices.
- **Parallel** execution: Two or more processes running *simultaneously*.

Chapter 2 - Processes and Threads

- LTS models are always finite, but transition traces can be infinite (2.1)
- FSP (documentation: https://cs.au.dk/~amoeller/dConc/fsp.pdf)
    - **Transition-/action-based**. *State names have no meaning*.
    - `x->P` means **executing action** `x`, and ending up in *process* `P` (2.1.1)
    - A sequence of actions is a **trace** (2.1.1)
    - `(x->P|y->Q)` denotes a **choice** between executing `x` and ending up in `P`, or executing `y` and ending up in `Q` (2.1.2).
        - This type of choice does not make LTS non-deterministic, as the "input string" is (sometimes) partially reliant on human interaction (drink dispenser).
    - `(x->P|x->Q)` denotes **non-deterministic choice** between ending up in process `P` or process `Q` after executing `x` (2.1.2, coin toss).
        - LTSA picks a process at random during animation (test).
    - Actions and processes can be **indexed** using brackets (2.1.3).
        - `(in[i:0..3]->out[i])` denotes 4 *actions*: `in[0]`, `in[1]`, `in[2]`, `in[3]`, followed by similar `out` actions (one-slot buffer, BUFF). The `i` in the `out` actions is determined by the chosen `in` action; `in[2]->out[2]`
        - Similarly, in the *process* `STORE[i:0..3] = (out[i]->...)`, the `out` action is determined by which value the `STORE` process is called with.
    - Processes can be **parameterized** using parenthesis (2.1.4).
    - Actions can be **guarded**, using `when`. `(when B x->P|y->Q)` means that *when* the condition/boolean `B` is true, execute `x`; `y` otherwise (2.1.5).
    - A **process alphabet** can be extended using `+{x,y,...}`. Extending the alphabet allows the process to "see" more actions, even though it may never engage in them (2.1.6).

Chapter 3 - Concurrent Execution

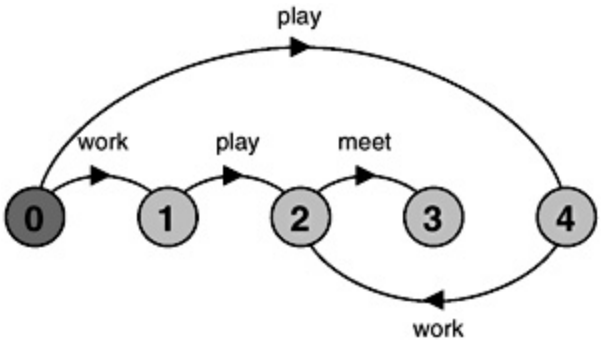
- **Interleaving** is when the processor switches between processes to “simulate” concurrent execution

- Processes can be **composed to execute concurrently**. `(P||Q)` means `P` and `Q` are executed concurrently (3.1.1). `||R = (P||Q)` means `R` is a *process composition*, and is composed of `P` and `Q`.

    - The resulting state machine is formed by the **Cartesian product** of the states in the composed processes, i.e. $States_P \times States_Q$ (3.1.1).

    - If composed processes have **shared actions**, these actions (unless relabelled) must occur *at the same time* during execution (3.1.2).

```
BILL = (play -> meet -> STOP).
BEN  = (work -> meet -> STOP).
||BILL_BEN = (BILL || BEN).
```



    - **Shared actions** are used to **synchronize** concurrent processes (3.1.2).

    - Process actions can be **prefixed with labels**. `a:P` prepends all actions `x` in `P` with `a`, i.e. `a.x`, which is useful if two instances of the same process need to be executed concurrently, but *independently of each other* (3.1.3).

```
||TWO_SWITCH = (a:SWITCH || b:SWITCH).
```

- Two switches operating independently of each other because of labelling.

    - If such two instances **share a resource** (mutual exclusion), the resource must be labelled with the set `{a,b}`. `{a,b}::P` (notice double-colon!) will make an action `a.x` and `b.x` for all actions $x \in P$ (3.1.3).

```
USER = (acquire->use->release->USER).
RESOURCE = (acquire->release->RESOURCE).
||RESOURCE_SHARE = (a:USER || b:USER || {a,b}::RESOURCE).
```

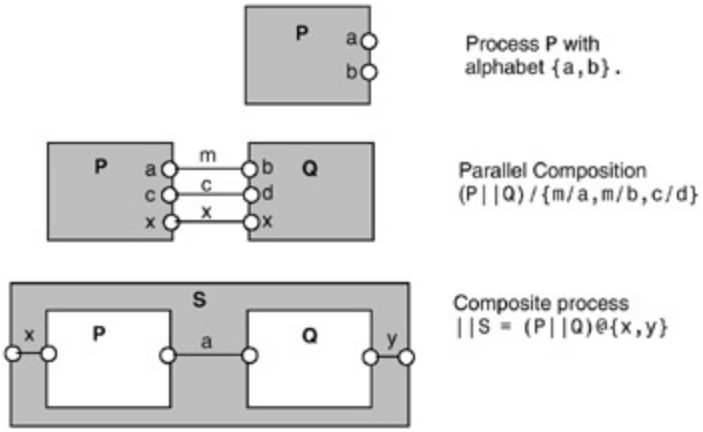
- The users can take turns acquiring, using and releasing the resource, but they can’t do it at the same time. The resource is mutually exclusive.

    - Proess actions can be **relabelled**, meaning the action labels change (3.1.4). `(P||Q)/{y/x,b/a}` (*forward* slash) changes all action labels `x` to `y`, and `a` to `b`,

i.e. `new_label/old_label`.
    - Relabelling also works on *prefix labels*.

- Actions can be **hidden**, meaning they will become *silent* or *internal actions* (replaced by `tau`). Actions can be hidden with either the hiding operator `\{...}` (*backslash*), or by excluding them with the interface operator `@{...}` (3.1.5).
    - If alphabet of `(P||Q)` is `{x,y,z}`, `(P||Q)\{y}` and `(P||Q)@{x,z}` are *equivalent*!

- **Structure diagrams** can be used to visualize processes.



Chapter 4 - Shared Objects and Mutual Exclusion

- **Inteference** is when two or more processes share an object, and their arbitrary interleaving causes the state of the shared object to be *incorrect* (4.1). It is also called a *destructive update* (4.1.2).
- Prefixing a method with `synchronized` in Java makes access to the object **mutually exclusive**, meaning only one process/thread may call it at any one time (4.2). A thread will block if the object lock is not released by the time the thread calls the method.

Chapter 5 - Monitors and Condition Synchronization

- A **monitor** is a (typically shared) object that doesn't do anything by itself (it is passive), and is accessed by other objects/processes. The monitor provides an interface through which the access to a shared resource is made mutually exclusive (5).
    - In Java, a monitor is simply a class with private fields that can only be accessed and/or mutated through `synchronized` methods (5).

- The `synchronized` keyword can be used for **condition synchronization**, which means blocking other threads until a condition holds.
    - In FSP: `when cond act -> NEWSTAT`
    - In Java, this is usually realized with this pattern inside a `synchronized` method

or code block:

```
while (!cond) wait();
...
notifyAll();
```

- A **semaphore** is an integer *s*, that, once given an initial value, only allows two operations; *up* and *down*, i.e. increments and decrements. The semaphore can always be incremented, but only decremented when greater than 0 (5.2). A *binary semaphore* is a semaphore that only ever holds the values 0 and 1 (5.2.1).

  down: `when s > 0 decrement s;`

  up: `increment s;`

  - Semaphores can be used as **locks** for *mutual exclusion*, by having an initial value of 1. When a process wants the lock, it executes the `down` action on the semaphore, which is only possible if no other process has done so without also executing `up` to *release* the lock (5.2.1).
  - In Java, semaphores can be modelled as passive classes with synchronized `up()` and `down()` methods, i.e. as a *monitor* (5.2.2). The guard on the `down()` method can be modelled with *condition synchronization*.
  -

- If a monitor is a type of **buffer** that doesn't alter or check the data it handles, the system is *data-independent* (5.3.1).

- A **monitor invariant** is an assertion about the monitor's variables. The assertion must be true whenever no processes are accessing the shared resources (5.5).

Chapter 6 - Deadlock

- A **deadlock** is a situation in which two or more processes are waiting for resources held by the others, such that none of them can proceed until they get the resources they are waiting for. These four conditions are necessary and sufficient for a deadlock to occur:
  - **Serially reusable resources** - a shared resource
  - **Incremental acquisition** - processes hold resources while waiting for more
  - **No preemtion** - processes can't force other processes to give up resources
  - **Wait-for cycle** - a cycle exists such that one process holds a resource its successor waits to acquire

Chapter 7 - Safety and Liveness

- A **safety property** asserts that nothing bad ever happens
- A **liveness property** asserts that something good eventually happens
- **Progress properties** is a subset of liveness properties. They assert that a specified action will always eventually be executed (opposite of **starvation**).
- **Fair choice/scheduling** asserts that if a set of transitions is available infinately often

then every transition in the set will be executed infinately often

Chapter 8 - Model-based design

- **Safety properties** should be composed with the system or subsystem it refers to. It is thus not necessary to compose them with a full/complete system, if they only refer to a small module/subsystem (8.1.4).
- **Progress properties** should be composed with the complete system, and not only with a module/subsystem (8.1.6).

Chapter 13 - Program Verification

- **Sequential composition** is simply executing one process after another (given that the first process successfully ends) (13.1.2).
    - In FSP, `P;Q` denotes sequential composition, and it means that when `P` has executed and successfully ended, `Q` is executed (13.1.2).
    - In Java, using the `Thread.join()` method can be used to achieve the same effect (no ref).

Chapter 14 - Logical Properties

- FSP is *action-based*, but using **fluents**, we can map an action trace to a sequence of abstract states (14.1).
    - `fluent FL = <{`s_1`,...,`s_n`},{`e_1`,...,`e_n`}> initially B` is a fluent that is initially true if `B` is true, and false otherwise (omitting `initially B` means the fluent is also initially false) (14.1.1).
    - A fluent holds/is true, if it is either initially true, or one of the actions $s_1, ..., s_n$ has occurred, and false if it is either initially false, or one of the actions $e_1, ..., e_n$ has occurred (14.1.1). That is, the `s`-actions make the fluent true, and `e`-actions make the fluent false.
- Fluents vs. LTL
    - `[]` = □ - *Always* (14.2.1)
    - `<>` = ◇ - *Eventually* (14.2.2)

B&K

Chapter 1 - System Verification

Chapter 2 - Modelling Concurrent Systems

- A **transition system** *TS* is a tuple *(S, Act, →, I, AP, L)* where
    - *S* is a set of states
    - *Act* is a set of actions

    - $\rightarrow \subseteq S \times Act \times S$ is a transition relation
    - $I \subseteq S$ is a set of initial states
    - AP is a set of atomic propositions
    - $L : S \rightarrow 2^{AP}$ is a labeling function
- *TS* is called finite if *S*, *Act*, and *AP* are finite

- TS is **action-deterministic** if $|I| \leq 1 \wedge |Post(s, \alpha)| \leq 1$ for all states s and actions α
- TS is **AP-deterministic** if $|I| \leq 1 \wedge |Post(s) \cap \{s' \in S \mid L(s') = A\}| \leq 1, \forall$ states $s \wedge A \in 2^{AP}$

- A ***maximal execution fragment*** is either a finite execution fragement that ends in a terminal state or an infinite execution fragment
- An ***initial execution fragment*** is an execution fragment that starts in an initial state i.e. $s \in I$
- An ***execution*** is an initial, maximal execution fragment

- A **Program Graph** *PG* over set *Var* of typed variables is a tuple $(Loc, Act, Effect, \hookrightarrow, Loc_0, g_0)$
    - Loc is a set of locations and Act is a set of actions
    - Effect: Act x Eval(Var) → Eval(Var) is the effect function
    - $\hookrightarrow \subseteq$ Loc x Cond(Car) x Act x Loc is the conditional transition relation
    - $Loc_0 \subseteq$ Loc is a set of initial locations
    - $g_0 \in$ Cond(Var) is the initial contition

- A **transition system** is obtained from a **program graph** by **unfolding**

Chapter 3 - Linear-Time Properties

- **Unconditional fairness**: Every process gets its turn infinitely often
- **Strong fairness**: Every process that is enabled infinitely often gets its turn infinitely often
- **Weak fairness**: Every process that is continuously enabled from a certain time instatn on gets its turn infinitely often

Chapter 5 - Linear Temporal Logic

- $TS \models \varphi \quad \Leftrightarrow \quad Traces(TS) \subseteq Words(\varphi) \quad \Leftrightarrow \quad TS \models Words(\varphi)$

Spørgetime spørgsmål:

- På s. 68 i M&K (kap 4.3) bruges +VarAlpha til at udvidde processens alfabet, på trods af, at alfabetet (tilsyneladende) allerede indeholder alle actions fra VarAlpha (men ikke write[0] ← that's why!). Er det et levn/en fejl, eller er der en egentlig årsag?
  "The alphabet for the TURNSTILE process is extended with this set using the alphabet

extension construct +{...}. This is to ensure that there are no unintended free actions. For example, if a VAR write of a particular value is not shared with another process then it can occur autonomously. A TURNSTILE process never engages in the action value.write[0] since it always increments the value it reads. However, since this action is included in the alphabet extension of TURNSTILE, although it is not used in the process definition, it is prevented from occurring autonomously."

- Hvad er forskellen på action based fluents og explicitly defined fluents? (side 348 midt)