Motivation

1. In the following we will develop a concise (mathematical) framework for formally describing systems of interest (→ formal model).

2. This framework allows one to formally, i.e., mathematically reason about a model’s and hence a system’s correctness w.r.t. dedicated properties, e.g., deadlock-freeness etc.

3. In principle we could start with any programming language. However, their interpretation is very complicated (address arithmetic, arbitrary data types, ...). Also only certain aspects of a system matter, where one may abstract away many details. Hence it appears useful to follow a more abstract view and speak here only about very simple “languages” for describing systems. Such methods are commonly denoted as high-level model description methods.

Motivation

1. Even though the high-level model description methods appear very simple, they possess clearly defined (execution or operational) semantics. These semantics allow us to map them to graphs.

2. These graphs represent all possible behaviors of the specified high-level description.

3. Hence the basic objects which represent the entities to be studied are graphs. Therefore we will briefly re-visit some basic definitions, which you probably have already seen before.

4. Don’t mind the formal notation, this will be made clear by examples and allows you to understand the resp. literature.

THE DINING PHILOSOPHERS

There are N philosophers sitting around a circular table either thinking or eating pasta. Each philosopher needs his left and right fork to eat, but there is only one fork between each 2 philosophers. Design an algorithm that the philosophers can follow.

Consider the following protocol (= sequence of interaction)

```plaintext
void philosopher()
while(1)
{
  think();
  get_left_fork();
  get_right_fork();
  eat();
  put_left_fork();
  put_right_fork();
}
```


Part II

Basic (formal) Facts about Graphs

**Definition 2.1: Graph**

A graph G is a pair (V, E) where

1. V is a discrete set of vertices (or nodes)
   \{v_1, \ldots, v_n\}

2. E ⊆ V × V is a discrete set of pairs
   \{(v_i, v_j) \mid \text{for some } i, j \in \{1, \ldots, m\}\}. One commonly denotes these pairs as edges or arcs.

**Definition 2.2: Directed Graph (Digraph)**

- If the elements of E are ordered in such a way that (v, w) ≠ (w, v) the elements of E are denoted as directed edges or directed arcs.
- A graph with such a property is denoted directed graph or digraph for short.
In a digraph and for an ordered pair \((u, v) \in E\) vertex \(u\) is denoted as predecessor or parent of vertex \(v\) and vertex \(v\) is denoted as successor or child of vertex \(u\).

This can be extended to the level of sets of children and parent nodes as follows:

- \(Post(u) := \{v \in V \mid (u, v) \in E\}\) is the set of all children or direct successors of a vertex \(u\).
- \(Pre(v) := \{u \in V \mid (u, v) \in E\}\) is the set of all parents or direct predecessors of a vertex \(v\).

The above sets are very helpful, once we want to operate on graphs.

--- 

Question 2.1: Please write down an algorithm which check if a node \(n\) is contained in a graph.

Definition 2.3: Weighted Graphs

- If the edges of a graph are labelled with elements from \(R\) one speaks of a weighted graph.
- In fact the set of edges of a weighted digraph is a ternary relation (set of triples):
  \[E \subseteq V \times R \times V\].
- Function \(W : V \times V \rightarrow R\) gives one the weight associated with the respective edge \((u, x, v)\).
- \(W(u, v) := 0\) for all triples not contained in \(E\).

Definition 2.4: Labelled Graphs

- If the edges of a graph are labelled with elements from a finite set, e.g. \(I \in \text{Act}\) one speaks commonly of a labelled graph.
- In fact the set of edges of a labelled digraph is once again a ternary relation: \(E \subseteq V \times \text{Act} \times V\).

Note:

A labelled graph is also often referred to as labelled transition system (LTS), where instead of vertices one speaks of states.

Definition 2.5: Labelled Transition System (LTS)

A LTS is a quadruple \(T := (S, S_0, \text{Act}, E)\), where

- \(S := \{s_1, \ldots, s_n\}\) is an ordered (indexed) set of states with \(s_0\) as the set of initial states.
- \(\text{Act}\) is the discrete set of transition labels,
- \(E \subseteq S \times \text{Act} \times S\) is an ordered (indexed) set of labelled state-to-state transitions.

Note:

LTS are essentially the semantics of the here discussed high-level modelling techniques, where the techniques of model checking allow us to reason about their properties. In the following we briefly strive some important definitions.

- A LTS \(T\) is defined non-terminal if each state has at least one out-going edge otherwise \(T\) is called terminal.
- A LTS \(T\) is defined deterministic if each state has at most one out-going edge with the same edge label otherwise \(T\) is denoted as non-deterministic.

Are non-deterministic finite LTS more expressive than deterministic ones?
BNF is a method for compactly writing down production rules (of a context-free grammar). The production rules employ variables (capital letters) and terminal symbols (lower case letters).

\[ A ::= \text{true} | \alpha \in \mathcal{AP} | \neg A | A \land A | (A) \]

is read as follows: each occurrence of variable \( A \) can be replaced by the constant \( \text{true} \), a terminal symbol of set \( \mathcal{AP} \) or \( \neg A \) or \( A \land A \) or \( (A) \). One may note that \( A \) may not only be a non-terminal symbol (variable), it can also be a word produced by the above grammar. This is, because it also appears on the left-hand side.

--- Question 3.1: What does we above rule define?