Specification models and their analysis –Background Material–

Kai Lampka

December 10, 2010

Institut für Technische Informatik und Kommunikationsnetze

Part I

Introduction

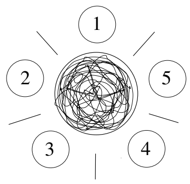
Motivation (1)

- In the following we will develop a concise (mathematical) framework for formally describing systems of interest ($\rightarrow$ formal model).
- 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.
- 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**.

The dining philosophers Dijkstra'65 (en.wikipedia.org/wiki/Dining_philosophers_problem)

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)




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

Properties: Deadlock? Starvation-free? Etc.

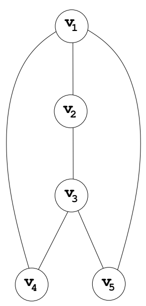
Motivation (2)

- 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.
- These graphs represent all possible behaviors of the specified high-level description.
- 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.
- Don't mind the formal notation, this will be made clear by examples and allows you to understand the resp. literature.

Part II

Basic (formal) Facts about Graphs

Preliminaries – Foundations of Graphs (at a glance)

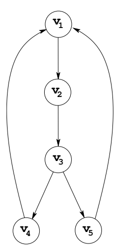


Definition 2.1: Graph

A graph **G** is a pair $(\mathbb{V}, \mathbb{E})$ where

1. $\mathbb{V}$ is a discrete set of vertices (or nodes) $\{v_1, \ldots, v_m\}$
2. $\mathbb{E} \subseteq \mathbb{V} \times \mathbb{V}$ 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.

Preliminaries – Foundations of Graphs (at a glance)



Definition 2.2: Directed Graph (Digraph)

- If the elements of $\mathbb{E}$ are ordered in such a way that $(v, w) \not\equiv (w, v)$ the elements of $\mathbb{E}$ are denoted as directed edges or directed arcs.
- A graph with such a property is denoted directed graph or *digraph* for short.

Preliminaries – Foundations of Graphs (at a glance)

1. In a digraph and for an ordered pair $(u, v) \in \mathbb{E}$ vertex u is denoted as **predecessor or parent** of vertex v and vertex v is denoted as **successor or child** of vertex u.
2. This can be extended to the level of sets of children and parent nodes as follows:
   - $\mathcal{P}ost(u) := \{v \in \mathbb{V} \mid (u, v) \in \mathbb{E}\}$ is the set of all **children or direct successors** of a vertex u.
   - $\mathcal{P}re(v) := \{u \in \mathbb{V} \mid (u, v) \in \mathbb{E}\}$ is the set of all **parents or direct predecessors** of a vertex v.
3. The above sets are very helpful, once we want to operate on graphs.

$\longrightarrow$ Question 2.1: Please write down an algorithm which check if a node n is contained in a graph

Preliminaries — Reachability Check

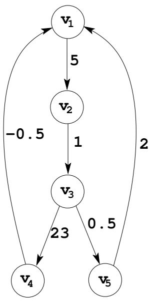
Check reachability of a node n

```
(1)  Nodes2beTraversed := {v0}, Nodes := ∅, Edges := ∅
(2)  While Nodes2beTraversed ≠ ∅
(3)    vs := getElement(Nodes2beTraversed) //get node to be processed next
(4)    Edges := PostSetEdges(v) //get set of outgoing edges of node vs
(5)    While Edges ≠ ∅ //still edges we did not travers so far?
(6)      e := pop(Edges)//pick one of the outgoing edges of node vs
(7)      vt := get2ndElement(e) //extract children node w. r. t. edge e
(8)      if (vt == n)  return(YES)//did we reach node n?
(9)      if (vt ∉ Nodes) // did we reach a new node vt to be tra-
             versed; avoid being trapped in cycles.
(10)       insert(vt, Nodes)// put vt in set of known states
(11)       insert(vt, Nodes2beTraversed)// put vt in set of states to be traversed
(12)     remove(e, Edges) //done with edge e
(13)   remove(vs, Nodes2beTraversed) //done with node vs
(14) return(NO)
```

Please clarify the functions `PostSetEdges`.

Order of traversal: depth-first-search or breadth first search?

Preliminaries – Foundations of Graphs (at a glance)

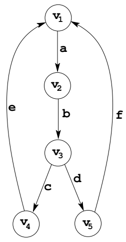


- $\mathcal{W}(v_3, v_5) =$
- $\mathcal{W}(v_3, v_1) =$

Definition 2.3: Weighted Graphs

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

Preliminaries – Foundations of Graphs (at a glance)



Definition 2.4: Labelled Graphs

- If the edges of a graph are labelled with elements from a finite set, e. g. $l \in \mathcal{A}ct$ one speaks commonly of a *labelled graph*.
- In fact the set of edges of a labelled digraph is once again a ternary relation: $\mathbb{E} \subseteq \mathbb{V} \times \mathcal{A}ct \times \mathbb{V}$.

Note:

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

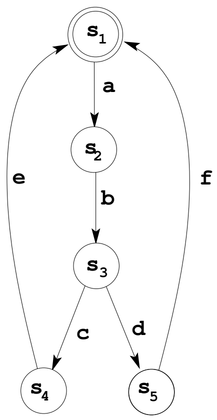
Preliminaries – Labelled Transition System

Definition 2.5: Labelled Transition System (LTS)

A LTS is a quadruple $\mathcal{T} := (\mathbb{S}, \mathbb{S}_0, \mathcal{A}ct, \mathbb{E})$, where

1. $\mathbb{S} := \{\vec{s}_1, \ldots, \vec{s}_n\}$ is an ordered (indexed) set of states with
2. $\mathbb{S}_0$ as the set of initial states.
3. $\mathcal{A}ct$ is the discrete set of transition labels,
4. $\mathbb{E} \subseteq \mathbb{S} \times \mathcal{A}ct \times \mathbb{S}$ is an ordered (indexed) set of labelled state-to-state transitions.

Preliminaries – Labelled Transition system



1. $\mathbb{S} := \{$
2. $\mathbb{S}_0 :=$
3. $\mathcal{A}ct := \{$
4. $\mathbb{E} := \{$

Preliminaries – Termination and Determinism of LTS

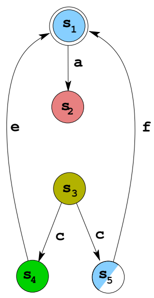
Note:

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.

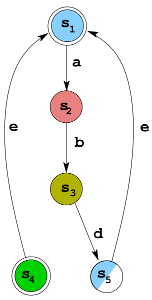
- A LTS $\mathcal{T}$ is defined *non-terminal* if each state has at least one out-going edge otherwise $\mathcal{T}$ is called *terminal*.
- A LTS $\mathcal{T}$ is defined deterministic if each state has at most one out-going edge with the same edge label otherwise $\mathcal{T}$ is denoted as *non-deterministic*.

Are non-deterministic finite LTS more expressive than deterministic ones?

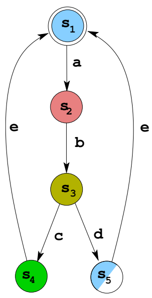
Preliminaries – Termination and Determinism (Examples)



non-terminal,deterministic?



non-terminal,deterministic?



non-terminal,deterministic?

Part III

Backus-Naur-Form

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).

$$A ::= \mathit{true} \mid \alpha \in \mathcal{AP} \mid \neg A \mid A \wedge A \mid (A)$$

is read as follows: each occurence of variable A can be replaced by the constant *true*, a terminal symbol of set $\mathcal{AP}$ or $\neg A$ or $A \wedge A$ 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.

$\longrightarrow$ Question 3.1: What does we above rule define?