Design and Analysis of Distributed Interacting Systems

Organization
Prof. Dr. Joel Greenyer

April 11, 2013
Organization

• Lecture: Thursdays, 10:15 – 11:45, F 128
• Tutorial: Thursdays, 13:00 – 13:45, G 323
  – first tutorial: April 11 (today)
• No lecture on May 23. (Widsun Break, Pfingsten)

• Script: Lecture Slides, will be available after each lecture on the SE webpage (Stud.ip later)

• Grades:
  – Oral Exam, in English or German
  – Dates: t.b.d., probably some time in August
  – Prerequisite: 50% of total points in exercises and have presented an exercise solution during tutorial / lecture
  – Grade +0.3 with 80% of points in exercises and have presented a solution of an exercise in the tutorial
Design and Analysis of Distributed Interacting Systems

Brief Excursion – Hannover Messe Impressions

Prof. Dr. Joel Greenyer

April 11, 2013
Visit of Hannover Messe

- ABB: Demonstrator and Development with the Control Builder Plus
Visit of Hannover Messe

• Siemens: Design and Simulation of a production system with Tecnomatix
Design and Analysis of Distributed Interacting Systems

Lecture 2 – Modeling with Automata and Statecharts

Prof. Dr. Joel Greenyer

April 11, 2013
Automata

• There are different kinds of automata which we will get to know in the course of this lecture
  – for different purposes, with different extensions

• Recapitulation: Finite-State Automata
• Kripke Structures and Labeled Transition Systems
  – simple automata, used in model-checking algorithms
  – used as a semantic model for other, higher-level formalisms

• Automata extended with variables, conditions and assignments
• UML Statecharts
• Later: Büchi automata, Timed Automata
Automata

- Used for modeling the discrete behavior of systems

- Original use: describing hardware switching circuits
  - states are the bit-values of registers

- A simple example: a light switch
Finite-State Automata

A finite-state automaton is a tuple $A = (Q, \Sigma, T, I, F)$ in which

- $Q$ is a finite, non-empty set of states
- $\Sigma$ is a finite alphabet, an element in $\Sigma$ is called a symbol (a symbol is also called an input)
- $T \subseteq Q \times \Sigma \times Q$ is a transition relation
- $I \subseteq Q$ is a set of initial state
- $F \subseteq Q$ is a set of final states

$Q = \{\text{off, low, high}\}$
$\Sigma = \{\text{press, hold}\}$
$T = \{(\text{off, press, low}), (\text{low, hold, high}), (\text{low, press, off}), (\text{high, press, off})\}$
$I = \{\text{off}\}$
$F = \{\text{off}\}$
A finite-state automaton $A = (Q, \Sigma, T, I, F)$ accepts a (finite) word $w = a_0, a_1, a_2, \ldots, a_n$ over the alphabet $\Sigma$ if there exists a sequence of states $q_0, q_1, \ldots, q_n$ in $A$ with

- $q_0 \in I$
- $(q_i, a_i, q_{i+1}) \in T$ for all $i = 1, \ldots, n-1$
- $q_n \in F$

The set of all words accepted by an automaton $A$ is called the language accepted by $A$, written $L(A)$

- $L(A) \subseteq \Sigma^*$
Finite-State Automata and Regular Languages

- The languages accepted by finite-state automata is the set of all regular languages
  - languages that can be described by regular expressions
  - or regular grammars
  - Type-3 in the Chomsky Hierarchy

Remember:

- recursively enumerable
- context sensitive
- context free
- regular

Type-0
Type-1
Type-2
Type-3
Regular languages over $\Sigma$

- The empty language is a regular language
- For every $a \in \Sigma$ the language \{a\} is regular
- Let $L_1$ and $L_2$ be regular languages, so are
  - $L_1 \cup L_2$ (union)
  - $L_1 \cdot L_2$ (concatenation, i.e., all strings $xy$, such that $x \in L_1$, $y \in L_2$)
  - $L_1^*$ ("Kleene closure", all sequences of elements in a given set, including the empty sequence)
- All other languages are not regular
- Examples:
  - regular: All words of as and bs with an even number of bs
  - not regular: All words of $n$-many as followed by $n$-many bs, $n \geq 1$
Regular Expressions

- Examples of the basic operators, given an alphabet $\Sigma=\{a, b\}$
  - $L(ab) = \{"ab"\}$
  - $L(a|b) = \{"a", "b"\}$
  - $L(ab?a) = \{"aa", "aba"\}$
  - $L(a^*) = \{"", "a", "aa", "aaa", \ldots\}$
  - $L(a^+) = \{"a", "aa", "aaa", \ldots\}$
  - $L((a|b)^*a) = \{"a", "aa", "ba", "aaa", "aba", "baa", "bba", \ldots\}$
  - $L(a^*(ba^*b)^*a^*) = \text{"all words with an even number of bs"}$. 
Properties of Regular Languages

• Why is $L_{ab} = \{a^n b^n \mid n \geq 1\}$ not regular?

• All words of a regular language accepted by a finite-state automaton with $n$-many states must have a repeated pattern if they exceed length $n$.

• **Pumping Lemma for regular languages**: Let $L$ be a regular language. Then there exists a constant $n$ such for every word $w$ in $L$, $|w| \geq n$ we can break up $w$ into three parts $w = xyz$, s.t.
  – $y$ is not an empty sequence
  – $|xy| \leq n$
  – for all $k \geq 0$, the string $xy^kz$ is also in $L$
  – (If $L = L(A)$, $A = (Q, \Sigma, T, I, F)$, we can pick any $n \geq |Q|$)
Constructing a Deterministic Finite-State Automaton from a Non-Deterministic One

• For every non-deterministic finite-state automaton, we can construct a deterministic finite-state automaton that accepts the same language
  – Example:

```
Non-deterministic

Deterministic
```

1  3  5
2  4

(a,b,c|a,b,c|a,b,c|a,b,c|a,b,c)

(a,b,c|a,b,c|a,b,c|a,b,c|a,b,c)
Constructing a Deterministic Finite-State Automaton from a Non-Deterministic One

- For every non-deterministic finite-state automaton, we can construct a deterministic finite-state automaton that accepts the same language
  - Example:
Complement of Regular Languages

• Regular languages have nice properties, for example
  – The complement $\overline{L} = \Sigma^* \setminus L$ of a regular language is regular
• Given a determinisitc automaton $A = (Q, \Sigma, T, q_0, F)$, with $T : Q \times \Sigma \rightarrow Q$ being a total function, $\overline{L(A)} = L(B)$ where $B = (Q, \Sigma, T, q_0, Q \setminus F)$
  – This means that we can construct an automaton that accepts the complement of a language by taking a deterministic finite-state automaton that accepts the language and complement the final states. (What does it mean that $T$ is a total function?)
Product Construction

- Since $L_1 \cap L_2 = \overline{L_1} \cup \overline{L_2}$, $L_1 \cap L_2$ is also regular
- Given finite-state automata accepting $L_1$ and $L_2$, we can construct a finite-state automaton accepting $L_1 \cap L_2$ by product construction
- Exercise: Systematically construct a finite-state automaton for accepting $L_1 \cap L_2$ where
  - $L_1$ is the set of all words over $\{a, b\}$ with an even number of $a$s
  - $L_2$ is the set of all words over $\{a, b\}$ with at least one $a$

Further reading:
*Introduction to Automata Theory, Languages, and Computation*  
John E. Hopcroft, Rajeev Motwani und Jeffrey D. Ullman
Modeling Reactive Systems

• Reactive systems continuously interact with their environment, i.e., Conceptually they never terminate.
• We need models which capture infinite executions

• There are two very basic types of models
  – *Kripke Structures (KS)*: States are labeled and transitions change states; used in model-checking algorithms where we care about states (and not by which action states change)
  – *Labeled Transition Systems (LTS)*: Transitions are labeled by actions; more suited when we are interested in what affects the state change, and for defining communicating processes
• They can be mapped to each other (with some restrictions and subtleties)
Kripke Structures

- Given is a finite set of *atomic propositions* $\text{AP}$
  - Boolean variables or Boolean expressions over other variables
  - Properties of a system that we are interested in
- A *Kripke Structure* over $\text{AP}$ is a tuple $KS = (S, S_0, R, L)$ in which
  - $S$ is a finite, non-empty set of states
  - $S_0 \subseteq S$ is a set of initial states
  - $R \subseteq S \times S$ is a transition relation that must be total
  - $L : S \rightarrow 2^{\text{AP}}$ is a function that labels each state with a set of atomic propositions that are true in this state

$$\text{AP} = \{\text{lightOn}, \text{savesEnergy}\}$$

![Diagram of Kripke Structure]

(state names have no meaning, to highlight this, we call them $s_0$, $s_1$, ...)
Paths, Runs, and Traces in a Kripke Structure

- We say that a proposition \( a \) is valid in a state \( s \) if \( a \in L(s) \)
- We write \( s_1 \rightarrow s_2 \) instead of \( (s_1, s_2) \in R \)
- A state \( s_2 \) is a successor state of \( s_1 \) if \( s_1 \rightarrow s_2 \)
- A path is a sequence of states \( \pi = s_1, s_2, s_3, \ldots \) if \( \forall i \in \mathbb{N}, s_i \rightarrow s_{i+1} \)
- A run is an infinite path starting in a start state
- A trace is a sequence of sets of properties corresponding to a path \( \sigma = L(s_1), L(s_2), L(s_3) \)

\[
\begin{align*}
\{\text{lightOn}, \text{savesEnergy}\} & \quad \{\text{savesEnergy}\} & \quad \{\text{lightOn}\} \\
\text{s}_0 & \quad \text{s}_1 & \quad \text{s}_3
\end{align*}
\]
A transition system is a tuple $TS = (S, \Sigma, T, I)$ in which

- $S$ is a set of states
- $I \subseteq S$ is the set of initial states
- $\Sigma$ is an alphabet, an element in $\Sigma$ is called a symbol
  (a symbol is also called an input, event, or action)
- $T \subseteq S \times \Sigma \times S$ is a transition relation
- A TS is called finite if $S$ and $\Sigma$ are finite

$S = \{\text{off, low, high}\}$
$I = \{\text{off}\}$
$\Sigma = \{\text{press, hold}\}$
$T = \{(\text{off, press, low}),
(\text{low, hold, high}),
(\text{low, press, off}),
(\text{high, press, off})\}$
A path of a transition system is a sequence of transitions $(s_i, a_i, s'_i)$ that follow each other, i.e., $\forall i: s'_i = s_{i+1}$.

(off, press, low), (low, hold, high), (high, press, off), ...
Executions

• An execution is a path starting from an initial state. An execution is complete or maximal if it cannot be extended, otherwise an execution is partial.
  – Complete executions are also called runs
  – Typically we speak of complete executions. So if we say just execution in the following, we mean a complete execution.

• Executions are finite if they end in a state without outgoing transitions, called a deadlock state. They are infinite otherwise.
  – We assume that an automaton never remains in a non-deadlock state forever.

• If we talk about the “behavior” of a TS, we typically mean the set of all its executions.
Variables, Conditions, and Assignments

- When modeling real-life systems, it is often convenient to
  - consider variables
  - consider guarded transitions
  - and side-effects of transitions on variables (assignments)

- Consider the following extended automaton. (We introduce these concepts in a by-example fashion.)
  - (we can also represent sequential programs this way)

![Diagram]

```plaintext
var:
int[0..3] b; // brightness

off
  press
  press
  on

hold [b<3] / b++
hold [b=3] / b=0
```

variable declaration

guard condition

assignment
A automaton with variables, guard conditions, and assignments can be transformed into a TS via *unfolding*

```plaintext
var:
   int[0..3] b; // brightness

off
   press
   on
      press
      hold [b<3] / b++
          hold [b=3] / b=0

unfolded TS without variables, guards and assignments:

("off b=2" is just the name of a state---not a variable!---to illustrate the correspondence)
Parallelism and Communication

• Often systems consist of many sequentially executed software components that run concurrently
  – different “functions” realized as parallel processes, e.g. Therac 25: Data entry, magnet positioning, …
  – system is physically distributed, e.g. cars, trains, phones, …

• We want to define a composition operator “||”
  – such that we can talk about $TS = T_{S_1} || ... || T_{S_n}$

• Different kinds of parallelism and communication
  – parallel execution
  – interleaving
  – communication via shared variables
  – handshaking
  – messages and channels (synchronous and asynchronous)
Parallel Execution

- Processes execute independently
- Transitions may take place simultaneously

Parallel execution of $TrafficLight_A$ and $TrafficLight_B$
Interleaving

- No simultaneous firing of transitions (partly independent)
- Like parallel threads scheduled on one processor

Interleaving of \( \text{T}rafficLight_A \) and \( \text{T}rafficLight_B \), we write \( \text{T}rafficLight_A \parallel \parallel \text{T}rafficLight_B \)
Communication via Shared Variables

- Example: Printer Manager
Handshaking

• Processes synchronize on certain common events
  – transitions with these events are executed simultaneously
  – only if both processes are “ready”
  – the other transitions are interleaved

Light

Switch

Light ||{press, hold} Switch
Handshaking

Let $TS_1 = (S_1, \Sigma_1, T_1, I_1)$ and $TS_2 = (S_2, \Sigma_2, T_2, I_2)$ be two transition systems and $H \subseteq \Sigma_1 \cap \Sigma_2$ then $TS_1 \|_H TS_2$ is defined as follows

- $TS_1 \|_H TS_2 = (S_1 \times S_2, \Sigma_1 \cup \Sigma_2, T, I_1 \times I_2)$
- where $T$ is defined by the following rules
  - if $a \in H$ and $(s_p, a, s'_p) \in T_1$ and $(s_2, a, s'_2) \in T_2$ then $(<s_p, s_2>, a, <s'_p, s'_2>) \in T$
  - if $a \in \Sigma_1 \setminus H$ and $(s_p, a, s'_p) \in T_1$ then $(<s_p, s_2>, a, <s'_p, s_2>) \in T$
  - if $a \in \Sigma_2 \setminus H$ and $(s_2, a, s'_2) \in T_2$ then $(<s_p, s_2>, a, <s'_p, s'_2>) \in T$

- If $H = \Sigma_1 \cap \Sigma_2$ we just write $TS_1 \| TS_2$ instead of $TS_1 \|_H TS_2$
- If $H = \emptyset$ then $TS_1 \|_H TS_2$ is equivalent to $TS_1 \|\| TS_2$
- Handshaking is also called synchronous message passing
Reachable States

- Note: Only some states in $S_1 \times S_2$ may be reachable

**Light**

- off \rightarrow press \rightarrow low \rightarrow hold \rightarrow high

- press

**Switch**

- rel \rightarrow press \rightarrow pr

- release

**Light \parallel_{\{press, hold\}} Switch**

- off, rel \rightarrow press \rightarrow low, pr

- release

- off, rel \rightarrow press \rightarrow low, rel

- press

- high, pr \rightarrow release

- high, rel \rightarrow release
Handshaking

• Handshake is associative if transition systems synchronize over their common actions
  \[(TS_1 \parallel TS_2) \parallel TS_3 = TS_1 \parallel (TS_2 \parallel TS_3)\]

• But handshaking is not generally associative
  \[(TS_1 \parallel_H TS_2) \parallel_H TS_3 \neq TS_1 \parallel_H (TS_2 \parallel_H TS_3) \text{ for } H \neq H\]
Pairwise Handshaking

• Sometimes we want to model processes that communicate in a pairwise fashion (an “n-ary” composition operator)
  
  $TS = TS_1 \parallel TS_2 \parallel \ldots \parallel TS_n$

• $TS_i$ and $TS_j$ ($0 < i \neq j \leq n$) must perform any transition labeled with an event from $H_{i,j} = \Sigma_i \cap \Sigma_j$ together
  
  – associative if $H_{i,j} \cap \Sigma_k = \emptyset$ for any $k \notin \{i, j\}$

![Diagram of a system with processes Switch, MotionDetector, LightController, and LightBulb communicating through pairwise handshaking.](Diagram.png)

```
Switch || MotionDetector || LightController || LightBulb
```
Messages and Channels

- Processes communicate by sending messages over channels
  - asynchronous: messages can be stored in a FIFO buffer (we will get back to these later)
  - synchronous: messages are sent and received at the same time
    - similar to handshaking, but introduces a direction
    - buffer of size “zero”
Messages and Channels

- “c!press” means that “press” is sent over channel “c”
- “c?press” means that “press” is received over channel “c”
Messages and Channels

- Some tools have simplified concepts of channels and events
- In UPPAAL, the channel name is the event name
  - Example: The light switch modeled in UPPAAL

```plaintext
// global declarations
chan press, hold, release;
```

**Light**

- start state has a double circle in UPPAAL syntax

**Switch**

- can only be sent if also received – what do we do?
- synchronous channels
Example: The Dining Philosophers

- Popular example by Edsger Dijkstra
- Five philosophers sit at a round table
- They think or eat rice from a bowl
- For eating, they need two chopsticks
- There is one chopstick between two neighboring philosophers
Example: The Dining Philosophers

what is the problem?
Statecharts are essentially finite-state automata with hierarchy and parallelism.

(Screenshot from Yakindu Statechart Tools)
Summary

• We will cover UML Statecharts in more detail next time

• There are different variants of automata for modeling reactive systems

• There are different ways to model the communication and coordination of different components of a system (and the of a system with its interaction environment)