Design and Analysis of Distributed Interacting Systems

Lecture 3 – Modeling with Automata and Statecharts (cont.)

*Prof. Dr. Joel Greenyer*

April 18, 2013
Last time

- Recap: Finite State Machines
- Kripke structure
- Labeled transition systems
  - extended with variables and conditions
  - different forms of parallelism and communication: shared variables, handshaking, messages and channels

Light

Switch

Light || Switch
Today

• Tool: The Labelled Transition System Analyser (LTSA)
• UML Statecharts

• Requirements Engineering and Design Basics
The LTSA Tool with Animator (for Simulation)
Labelled Transition System Analyser (LTSA)

- Modeling and verification tool for concurrent systems
- Modeling with *Finite State Processes (FSP)*
  - they can be transformed into Labeled Transition Systems (LTS)

- Supports some rich language features
  - composition via handshaking “||”
  - shared variables
  - multiple “instances” of processes (process labeling)

- Supports deadlock detection and automated verification of safety and liveness properties (more on those later)
Finite State Processes (FSP)

- The graphical notation for LTSs becomes unmanageable for big processes / transition systems
- Alternative: Textual, algebraic notation

\[
\text{SWITCH} = \text{RELEASED}, \\
\text{RELEASED} = (\text{press} \rightarrow \text{PRESSED}), \\
\text{PRESSED} = (\text{release} \rightarrow \text{RELEASED}).
\]
Finite State Processes (FSP)

- The following process definitions are equivalent:

\[
\begin{align*}
\text{SWITCH} &= \text{RELEASED}, \\
\text{RELEASED} &= (\text{press} \rightarrow \text{PRESSED}), \\
\text{PRESSED} &= (\text{release} \rightarrow \text{RELEASED}).
\end{align*}
\]

\[
\begin{align*}
\text{SWITCH} &= \text{RELEASED}, \\
\text{RELEASED} &= (\text{press} \rightarrow (\text{release} \rightarrow \text{RELEASED})).
\end{align*}
\]

\[
\begin{align*}
\text{SWITCH} &= (\text{press} \rightarrow (\text{release} \rightarrow \text{SWITCH})).
\end{align*}
\]

again the corresponding LTS:
Choice

• "**LIGHT_LOW** engages either in the actions **hold** or **press**. Then behaves as **LIGHT_HIGH** or **LIGHT_OFF**, respectively”

```
LIGHT_LOW = (hold->LIGHT_HIGH|press->LIGHT_OFF)
```

• Full **Light** example:

```
LIGHT  = LIGHT_OFF,
LIGHT_OFF = (press->LIGHT_LOW),
LIGHT_LOW  = (hold->LIGHT_HIGH|press->LIGHT_OFF),
LIGHT_HIGH = (press->LIGHT_OFF).
```
Variables and Conditions

const N = 3
range Brightness = 0..N

LIGHT = OFF[0],
OFF[b:Brightness] = (press->ON[b]),
ON[b:Brightness] = (press->OFF[b]
    |when (b<N) hold->ON[b+1]
    |when (b==N) hold->ON[0]).
Parallel Composition

• Composite process definitions are preceded by “||”
• Shared actions must be executed at the same time by all processes that share the action

Example:

\[ \text{||LIGHTSWITCH} = (\text{LIGHT} \mid \mid \text{SWITCH}). \]
The LTSA Tool with Animator (for Simulation)
UML Statecharts

- Statecharts are essentially finite-state automata with hierarchy and parallelism
Orthogonal Regions (also “AND”-States)

- Statechart can at once be in one sub-states per region
  - quasi-parallel execution
  - different semantics implemented in different tools
Design and Analysis of Distributed Interacting Systems

Lecture 3 – Requirements Engineering and Design

Prof. Dr. Joel Greenyer

April 18, 2013
We have already introduced a range of modeling languages.

But WHAT do we model HOW?
- How to obtain the requirements?
- How to come up with a (good) design?
Requirements Engineering

• A systematic requirements engineering is crucial
• Many projects fail or exceed their original budget due to insufficient requirements engineering

– areas without radio coverage were not considered
– mobile devices did not inform users correctly on emergencies
– multiple ambulances were called to one emergency

• Many other examples: Toll system for the German Motorways, …, (some presented previously)
The V-Model is the development process model used in most industries for technical and software-intensive systems – comes in many variants.
V-Model Adopted by the Automotive Industry

- V-model from an automotive supplier following Automotive SPICE (Software Process Improvement and Capability Determination, ISO/IEC 15504)

Software vs. System

- Different scopes of Systems and Software Engineering
Example: Coffee Machine

- coffee machine
- software controller
- bean storage
- grinder
- display
- boiler
- cash dispenser
- remote monitoring component
- customer
- irrelevant environment
- technician's car
- technician's employer
- customer's employer
- bean vendor
- water supplier
- energy supply
- bean supply
- water supply
- drainage
- building infrastructure

Defining the system and software boundary can be very hard!
Some systems consist of many mechatronic components (hardware and software)

The (Early) Conceptual Design Phase

Planning and clarifying the task → Conceptual design on system level → Conceptual design on module level → Concept integration

Task analysis → Environment analysis → Definition of application scenarios → Planning the product structure → Definition of (informal) requirements

The result is a model of the basic structure and behavior of the system, containing all the information for the subsequent discipline-specific implementation.

Environment Analysis

- Example: Production Cell

plates leave system on deposit belt

blanks enter system on feed belt

TableSensor

ArmA

ArmB

Controller

Press
Environment Analysis

- Context diagram
Different Choices of System Boundary and Relevant Context

- The system boundary and relevant context could also be chosen differently
Scenarios

- Positive scenarios describe possible behaviors
  
  A blank arrives via the feed belt on the table. Then ArmA picks up the blank, transports it to the press...

- Sometimes scenarios describe mandatory behavior
  
  After ArmA releases a blank into the press, ArmA must return to the table and the press must start pressing...

- Negative scenarios describe excluded/forbidden behaviors
  
  After ArmA picked up the blank and moves to the table, it must not release the blank before it arrives at the table...
Scenarios help us determine communication relationships and events/actions in the system.
Scenarios help us determine communication relationships and events/actions in the system.

Note: here actions are shared by two processes, but actions can also be shared by multiple processes.
Control of Actions

- Some actions are software-controlled, some actions are controlled by the context or the non-software part of the system
  - controllable vs. uncontrollable (by the software)
  - sometimes we call uncontrollable actions events

- Some uncontrollable actions are observable by the software, some are non-observable
  - observable: also called shared
  - non-observable: also called unshared

Environment Analysis

- Scenarios help us determine communication relationships and events/actions in the system.
Scenarios help us...

- **elicit system requirements:**
  - properties whose satisfaction will fully satisfy the customer
  - may not be directly implementable solely by the software
  - if not directly implementable, the software will need to cooperate with the non-software part of the system

- **capture domain knowledge (or environment assumptions):**
  - properties of the context or the non-software parts of the system
  - often required to help the software fulfill the system requirements

- **and specify software requirements (or software specification):**
  - properties implementable by the software of the system
  - can be given to a software development team

The software specification ($S$) and the domain knowledge ($K$) must be sufficient to guarantee that the system requirements ($R$) are satisfied:

$$S, K \models R$$

R1: For each blank that arrives on the table via the feed belt, a pressed plate must be placed on the deposit belt.
• The physical movement of ArmA

K1: If ArmA is ordered to move to the press, it must eventually arrive at the press (unless it is ordered to move to the table midway).

K2: If ArmA is ordered to move to the table, it must eventually arrive at the table (unless it is ordered to move to the press midway).
• The physical movement of ArmA

K3: If ArmA is ordered to move to the press, it will not arrive at the table (unless it is ordered to move to the table midway).

K4: If ArmA is ordered to move to the table, it will not arrive at the press (unless it is ordered to move to the press midway).
• Controlling ArmA:

S1: When a blank plate arrives at the table, ArmA must pick it up, move it to the press, and release it into the press. ArmA then must move back to the table, where it must arrive before the next blank arrives.
The production cell can only work if it is sufficiently fast to handle incoming blanks.

K5: If a blank arrives at the table, ArmA immediately picks up the blank, moves to the press, and then back to the table, the next blank must not arrive before ArmA has returned to the table.
S2: When Arm A releases the blank into the press, the press must press.
System Specification

- ArmB:

  S3: After the press pressed the plate, ArmB must pick up the plate and transport it to the deposit belt.

  S4: After ArmB has arrived at the deposit belt, it must release the processed plate and then move back to the press.
ArmB:

K6: If ArmB is ordered to move to the d.belt, it must eventually arrive at the d.belt (unless it is ordered to move to the press midway).

K7: If ArmB is ordered to move to the press, it must eventually arrive at the press (unless it is ordered to move to the d.belt midway).
Safety and Liveness

• Most properties are either safety or liveness properties

• **Safety**: Something **bad** must **never** happen
  – “If ArmA is ordered to move to the press, it will not arrive at the table (unless it is ordered to move to the table midway).”
  – “If a blank arrives at the table, ArmA picks up the blank, and moves to the press, the next blank must not arrive before ArmA has returned to the table.”

• **Liveness**: Something **good** must **eventually** happen
  – “When ArmA releases the blank into the press, the press must press.”
  – “If ArmB is ordered to move to the press, it must eventually arrive at the press”

Invariants express that some condition must be true always.—They can be expressed as safety properties: Some condition must never be false.
What to do with System Requirements, Domain Knowledge, Software Specification

- Check that $S, K \models R$ holds
  - if not, we have to rething $S, K$ or $R$
  - checking this manually is error-prone
  - there are tools which allow us to check this automatically
    - properties must be specified formally
    - can take very long or is too complex to compute

- Alternative
  - Create a state-based model of the software, the non-software part of the system, and the context.
  - check whether $S, K$ hold, and ultimately $R$ holds
    - change model or rething $S, K$ or $R$ if not
  - there are tools which allow us to check this automatically
    - again: properties must be specified formally
    - can also take very long or is too complex to compute