PressPlateAfterArmAReleasesBlankPlate

- c: Controller
- a: ArmA
- p: Press
- b: ArmB

- releaseBlank
- press
- pressingFinished
- pickUp
Learning Objective Today

• Understand a fundamental approach to modeling software systems

• Learn about a formal scenario-based design approach
Formal Methods in Software Engineering

Lecture 10 – Third Mini-Project
Prof. Dr. Joel Greenyer

December 16, 2014
Third Mini-Project

• You can choose from one of two tasks

• Or invent your own task!

• **Task 1: Build an LTL model checker for LTS networks**
  - similar to the CTL model checker, but now implement an automaton-based approach for LTL model-checking (as presented in the lecture)
  - you can use an existing tools (LTL2BA) for the LTL-to-BA conversion
  - It's also fine to require the user to specify BAs manually (hence, build an BA checker, not a LTL checker)
Third Mini-Project

- **Task 2: Model checking Yakindu statecharts**
  - implement a mapping from Yakindu statechart models to the UPPAAL model-checker
  
  - so that you can verify CTL formulas in UPPAAL
  - Create some examples to test and demonstrate the mapping
    - if you can, think about encoding time constraints and parallelism

- **Links:**
  - [http://www.uppaal.org/](http://www.uppaal.org/)
  - [http://statecharts.org/](http://statecharts.org/)
Third Mini-Project

• **Task 3: Make up your own task**
  – Maybe you have encountered some sort of formal models or verification problems in other lectures, or outside of university – think of ways to formally verify these models
    • Business process models, controllers of automation systems, etc.
    • Create a mapping to some suitable model-checker (as in Task 2)
  – Or do you have another, idea?

  – If you make up your own task, send me via email a short description of the task by the end of the week (Friday, 19th)
    • then I can tell you whether this is a suitable task
Formal Methods in Software Engineering

Lecture 10 – Modeling Methodology

Prof. Dr. Joel Greenyer

December 16, 2014
A typical Software/Systems Development Process...

- **requirements analysis**
- **specification**
- **design**
- **implementation**
- **unit tests**
- **integration/system tests**
- **use and maintenance**

```java
public void run(){
    ...;
}
```
A typical Software/Systems Development Process...

How do we formalize requirements and create formal models of (software) systems?
Software vs. System

• Different scopes of Systems and Software Engineering
Example: Coffee Machine
Example: Train Crossing

- Train passengers
- Other trains
- Energy supply
- Train conductor
- Car driver
- Barriers
- Crossing controller
- Crossing system
- Coffee machine context
- Irrelevant environment
- Remote monitoring component
- Train direction
- Approaching, enter, exit
Modeling the Train Example with LTSs

- How did we model the system with LTSs?

Disclaimer: This is a **very** simple model!

**Very abstract:** For example the barrier may not work, a car may not exit the crossing, etc.

**Strange:** The software controller can actually block the train from entering before the barriers are closed.

(Ideas how to set this right?)
Example: Train Crossing

- **crossing-controller**
- **train direction**
- **approaching**
- **enter**
- **exit**
- **barriers**
- **open**
- **close**
- **car**
- **car-enter**
- **car-exit**
- **open**
- **approaching**
- **enter**
- **exit**
- **train**
- **approaching**
- **enter**
- **exit**

**System Software**

- **c0**
- **c1**
- **c2**
- **c3**
- **c4**

- **t0**
- **t1**
- **t2**
We also call a model of the system's non-software parts the **domain knowledge** or **environment assumptions**.

We also call a model of the system's software the **software specification** or **software requirements**.
Example: Train Crossing

We also call a model of the system's non-software parts the **domain knowledge or environment assumptions**

"There must never be a car and a train on the crossing together"

Properties that must be satisfied by the overall system (formed by the system's software and its non-software parts) are called **requirements** or **system requirements**

We also call a model of the system's software the **software specification** or **software requirements**
The software specification \((S)\) and the domain knowledge \((K)\) must be sufficient to guarantee that the system requirements \((R)\) are satisfied:

\[ S, K \models R \]


In our train example:

\[ \text{controller} \parallel \text{train} \parallel \text{barriers} \parallel \text{car} \models G \neg (\text{trainOnCrossing} \land \text{carOnCrossing}) \]

\[ S, K \models R \]

(LTSs extended with according atomic propositions)
System Requirements, Domain Knowledge, Software Specification

• **Requirements or 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

• **Domain knowledge or environment assumptions:**
  – properties of the context or the non-software system parts
  – often required to help software fulfill the system requirements

• **Software specification or software requirements:**
  – properties implementable by the software of the system
  – can be given to a software development team

• **Requirements or system requirements:**
  - properties whose satisfaction will fully satisfy the customer

\[
\begin{align*}
\text{controller} \parallel \text{train} \parallel \text{barriers} \parallel \text{car} & \models G \neg (\text{trainOnCrossing} \land \text{carOnCrossing}) \\
S, & \models K \models R
\end{align*}
\]

**Requirements:** “properties whose satisfaction will fully satisfy the customer”

Coming up with a complete temporal logics (LTL or CTL) specification of the requirements can be very difficult
Formal Methods in Software Engineering

Lecture 10 – Formal Scenario-Based Design

Prof. Dr. Joel Greenyer

December 16, 2014
Another Example: Production Cell

plates leave system on deposit belt

Controller

blanks enter system on feed belt

ArmB

ArmA

TableSensor

Press
Environment Analysis

• Context diagram
Humans usually describe and conceive behavioral properties in the form of **scenarios**

- What can, must, or must not happen in a certain situation?
Scenarios

- Scenarios can describe possible behavior

  A blank can arrive on the table. Then ArmA can pick up the blank and transport it to the press...

- Scenarios can describe mandatory behavior

  After ArmA releases a blank into the press, ArmA must return to the table ...

- 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...
Vision

• Specify **temporal properties** with **scenarios** instead of using temporal logics (like LTL or CTL), LTSs, or other automata
  – It is very difficult to express complex properties in CTL or LTL
  – building an model using LTSs or other state-based models can also be very difficult: we must find a model that satisfies many properties
  – using scenarios is easier

• As we will see, we can also do more things with scenarios
  – Simulate a system
  – Find software controllers that satisfy a scenario-based specification (or determine that this is not possible)
• **Modal Sequence Diagrams** (MSDs) are a formal interpretation of UML Sequence Diagrams
  – Defined by David Harel and Shahar Maoz

• Example:

> After ArmA releases a blank into the press, the press must press the blank. Then, when pressing is finished, ArmB must pick up the pressed item from the press.
Modal Sequence Diagrams (MSDs)

An MSD specification is a tuple \( MS = (O, \Sigma, D) \) where

- \( O \) is a set of objects, called the object system
- \( \Sigma = O \times \text{Name} \times O \) is the alphabet; an element of the alphabet is a message event (or simply event)
  - a message with a particular name sent from one object to another
  - for now we consider only synchronous messages: sending and receiving of a message is one event
  - \( \pi \in \Sigma^\omega \) is called a run
- \( D \) is a set of MSDs
  - an MSD \( d \in D \) accepts a set of runs, which are called the language of \( d \), \( L(d) \subseteq \Sigma^\omega \)
- there are two kinds of MSDs, \( D_u \cup D_e = D, D_u \cap D_e = \emptyset \)
  - \( D_u \subseteq D \): universal MSDs
  - \( D_e \subseteq D \): existential MSDs
Modal Sequence Diagrams (MSDs)

• Example: Production Cell

object system:

- TableSensor
- ArmA
- ArmB
- Controller
- Press
- b
- c
- p
- a
- ts
Modal Sequence Diagrams (MSDs)

- Example: Production Cell

Here we consider that objects are instances of classes. But this is an implicit extension of the definition.
Modal Sequence Diagrams (MSDs)

- Example: Production Cell

\[ \Sigma = \{(ts, \text{blankArrived}, c), (c, \text{pickUp}, a), (c, \text{moveToTable}, a), (a, \text{arrivedAtTable}, c), (c, \text{release}, a), \ldots\} \]

A possible run of the production cell:
\[ \pi = (ts, \text{blankArrived}, c), (c, \text{pickUp}, a), (c, \text{moveToTable}, a), (a, \text{arrivedAtTable}, c), \ldots \]
We assume that an object system $O$ is controlled by an LTS $C = (S, \Sigma, T, I)$, which we call its **controller**.

If an object system $O$ is controlled by a controller $C$ then the possible runs of the object system is the set of runs of $C$, written as $L(C)$.

**controller:**

```
(t, blankArrived, c)  
(c, pickUp, a)  
(c, moveToTable, a)  
...  
```

**object system:**

```
controller: c:Controller  
b:ArmB  
c:Controller  
p:Press  
a:ArmA  
...  
```
Controller of the Object System

- We assume that an object system $O$ is controlled by an LTS $C = (S, \Sigma, T, I)$, which we call its **controller**.

- If an object system $O$ is controlled by a controller $C$ then the possible runs of the object system is the set of runs of $C$, written as $L(C)$.

**controller:**

- $(ts, blankArrived, c)$
- $(c, pickUp, a)$
- $(c, moveToTable, a)$

**object system:**

- $b: ArmB$
- $p: Press$
- $a: ArmA$
- $ts: TableSensor$

**Diagram Note:**

Do not confuse the controller of the object system with the object that is called 'c:Controller' in this example.
A controller satisfies an MSD specification, written $C \models MS$ iff

- for all $d_u \in D_u$ and for all $\pi \in L(C)$: $\pi \in L(d_u)$
  - i.e., all runs are accepted by all universal MSDs
  - or: $L(C) \subseteq L(d_{u1}) \cap \ldots \cap L(d_{un})$, $\{d_{u1}, \ldots, d_{un}\} = D_u$
- for all $d_e \in D_e$ there exists a $\pi \in L(C)$: $\pi \in L(d_e)$
  - i.e., for all existential MSDs there exists a run that is accepted by it

Universal MSDs specify temporal properties that must hold for every run of a controller of the object system

- similar to LTL formulas, which we covered earlier

Existential MSDs specify temporal properties that must be satisfied by at least one run of the controller of the object system

- “a behavior that must be possible”
- similar to existentially quantified formulas in CTL

we will focus on universal MSDs in the remaining lecture
Modal Sequence Diagrams (MSDs)

• A Modal Sequence Diagram (MSD)
  – Each lifeline represents an object in the object system

• In universal MSDs, messages can have
  – an *execution kind*:
    • monitored: something expected *may* occur
    • executed: something expected *must eventually* occur
  – a *temperature*: hot or cold
    • hot: something expected at elsewhere in the scenario is forbidden
    • cold: something expected at elsewhere in the scenario is allowed, *but will “abort” the scenario*

<table>
<thead>
<tr>
<th></th>
<th>cold</th>
<th>hot</th>
</tr>
</thead>
<tbody>
<tr>
<td>monitored</td>
<td>← (c/m)</td>
<td>← (h/m)</td>
</tr>
<tr>
<td>executed</td>
<td>(c/e)</td>
<td>(h/e)</td>
</tr>
</tbody>
</table>
Universal MSD – Semantics by Example

May happen

<table>
<thead>
<tr>
<th>monitored</th>
<th>executed</th>
</tr>
</thead>
<tbody>
<tr>
<td>cold</td>
<td>hot</td>
</tr>
</tbody>
</table>

"Abort" scenario if violated

Must not be violated (safety)

Must eventually happen (liveness)
the first message is always cold and monitored: **monitored**: “blankArrived” may eventually happen **cold**: all other events in the MSD are allowed before “blankArrived” occurs)
**Universal MSD – Semantics by Example**

<table>
<thead>
<tr>
<th>monitored</th>
<th>cold</th>
<th>hot</th>
</tr>
</thead>
<tbody>
<tr>
<td>executed</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**executed**: “pickUp” must eventually happen

**hot**: no other event in the diagram must happen before “pickUp” occurred
**Universal MSD – Semantics by Example**

<table>
<thead>
<tr>
<th></th>
<th>cold</th>
<th>hot</th>
</tr>
</thead>
<tbody>
<tr>
<td>monitored</td>
<td></td>
<td></td>
</tr>
<tr>
<td>executed</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**monitored**: “arrivedAtPress” may happen

**hot**: no other event in the diagram must happen before “arrivedAtPress” occurred, e.g., ArmA must not release the blank, no new blank must arrive, etc.
Message Unification

- A message event in the system can be *unified* with a message in an MSD (called diagram message) if
  - the message names are equal
  - the source object is represented by the source lifeline
  - the target object is represented by the target lifeline
If a message event occurs that can be unified with the first MSD message, an active copy of the MSD (active MSD) is created

- The cut remembers that the first message was unified
- The cut progresses as subsequent messages are unified

we assume that there is always only one first message
If the cut is in front of a message on its sending and receiving lifeline, the message is **enabled**

- if an executed message is enabled, the cut is **executed**, otherwise it is **monitored**
- if a hot message is enabled, the cut is **hot**, otherwise it is **cold**
If a message occurs that can be unified with an enabled message, the cut progresses.
Universal MSD – Semantics by Example

- If a message occurs that can be unified with an enabled message, the cut progresses
• If a message occurs that can be unified with an enabled message, the cut progresses
• If a message occurs that can be unified with an enabled message, the cut progresses
• If a message occurs that can be unified with an enabled message, the cut progresses.
• If a message occurs that can be unified with an enabled message, the cut progresses.
• If a message occurs that can be unified with an enabled message, the cut progresses

• The active MSD **terminates** when the cut reaches the end of the MSD. The active copy of the MSD is then discarded.
• If a message occurs that can be unified with a message in an active MSD that is not currently enabled, this is a **violation**
  - violations are allowed in a cold cut, then the active MSD terminates. This is called a **cold violation**.
  - violations are forbidden in a hot cut: **safety violation**
Universal MSD – Semantics by Example

• An active MSD must not remain forever in an executed cut — otherwise, this is a liveness violation

• An active MSD may remain forever in a monitored cut

- An active MSD must not remain forever in an executed cut — otherwise, this is a liveness violation
- An active MSD may remain forever in a monitored cut

message event must eventually occur
Universal MSDs and Büchi Automata

- A universal MSD accepts a set of runs
- It can therefore be mapped to a Büchi Automaton

we abbreviate \( (ts, \text{blankArrived}, c) \) as \( bA \), etc., so \( \Sigma_d = \{bA, pU, mTP, aAP, rB, mTT, aAT\} \), \( \Sigma_d \subseteq \Sigma \)
Universal MSDs and Büchi Automata

• A universal MSD accepts a set of runs

• It can therefore be mapped to a Büchi Automaton

we abbreviate \((ts, blankArrived, c)\) as \(bA\), etc., so \(\Sigma_d = \{bA, pU, mTP, aAP, rB, mTT, aAT\}\), \(\Sigma_d \subseteq \Sigma\)
Universal MSDs and Büchi Automata

- A universal MSD accepts a set of runs
- It can therefore be mapped to a Büchi Automaton

We abbreviate \((\text{ts}, \text{blankArrived}, c)\) as \(bA\), etc., so 
\[ \Sigma_d = \{bA, pU, mTP, aAP, rB, mTT, aAT\}, \Sigma_d \subseteq \Sigma \]
Example: RailCab requests permission to enter crossing
• Example: RailCab requests permission to enter crossing

- RailCab requests permission to enter crossing
- RailCab: rc
- Crossing Control: crc
- Barriers: b

RequestEnterAtCrossing

- requestEnter
- closeBarriers
- enterDenied

Cold violation by event unifiable with non-first message.

Cold violation by event unifiable with first message. Represents termination and immediate reactivation.
MSDs – So what can we do with them?

- **Model-checking**: Checking whether a controller of an object system satisfies an MSD specification
  - e.g. \( \text{controller} \parallel \text{train} \parallel \text{barriers} \parallel \text{car} \models \)

```
+ counter example
  (how the specification can be violated)
```

```
false
```

```
true
```

```
modify model (usually the error is here)
```

```
- or modify specification (may also be wrong)
```
MSDs – So what can we do with them?

- **Model checking**: Checking whether a controller of an object system satisfies an MSD specification
  
  - e.g. \( \text{controller} \parallel \text{train} \parallel \text{barriers} \parallel \text{car} \models \) \[
  \begin{array}{c}
  \text{controller} \\
  \text{train} \\
  \text{barriers} \\
  \text{car}
  \end{array}
  \]

- **Execute/Simulate**: With MSDs, we can model complete system/software requirements, and environment assumptions
  
  - instead of using state-based models
  
  - we can simulate the behavior emerging from the interplay of the scenarios

- **Formal Synthesis**: Given an MSD specification, we can compute a state-based software controller that satisfies the specification, or find out that the specification is unrealizable, and no such controller exists