Design and Analysis of Distributed Interacting Systems

Lecture 9 – Modal Sequence Diagrams (MSDs)

Prof. Dr. Joel Greenyer

June 13, 2013
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
  - $\Sigma = O \times Name \times O$ 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, 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

- Diagram showing objects and their connections:
  - ArmA
  - ArmB
  - TableSensor
  - Press
  - Controller

- Object system:
  - a
  - b
  - c
  - p
  - 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:
\[ \pi = (ts, \text{blankArrived}, c), (c, \text{pickUp}, a), (c, \text{moveToTable}, a), (a, \text{arrivedAtTable}, c), \ldots \]
Controller of the Object System

• An object system $O$ can be controlled by a transition system (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 traces of $C$, written as $L(C)$.

controller:

object system:
• An object system $O$ can be controlled by a transition system (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 traces of $C$, written as $L(C)$.

**controller:**

\[
\begin{align*}
\text{(ts, blankArrived, c)} \\
\text{(c, pickUp, a)} \\
\text{(c, moveToTable, a)} \\
\text{...}
\end{align*}
\]

**object system:**

- ts: TableSensor
- c: Controller
- a: ArmA
- b: ArmB
- p: Press
- do not confuse the controller of the object system with the object that is called 'c:Controller' in this example
Satisfying an MSD Specification

• 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
  – similar to LTL formulae, which we covered earlier

• Existential MSDs specify temporal properties that must be satisfied by at least one run of the controller
  – “a behavior that must be possible”
  – similar to existential quantification in CTL, not covered yet
Satisfying an MSD Specification – Illustration

universal MSDs:

controller:

(all runs satisfy each uMSD?)

existential MSDs:

(exists a run satisfying each?)

exists a run satisfying each?
Satisfying an MSD Specification – Illustration

universal MSDs:

exists a run satisfying each?

c:Controller

a:ArmA

moveToPress

pickUp

blankArrived

controller:

we will focus on universal MSDs in the remaining lecture

existential MSDs:

all runs satisfy each uMSD?

(c, moveToTable, a)

(c, pickUp, a)

(c, moveToTable, a)

...
Modal Sequence Diagrams (MSDs)

- A Modal Sequence Diagram (MSD)
  - Each lifeline represents an object in the object system
- In universal MSDs, message 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**
- **executed**
- **monitored**
- **cold**
- **hot**
- **must not be violated (safety)**
- **must eventually happen (liveness)**

---

### ArmATransportBlankToPress

- **ts:** TableSensor
- **c:** Controller
- **a:** ArmA

#### Transitions:
- **blankArrived**
- **pickUp**
- **moveToPress**
- **arrivedAtPress**
- **releaseBlank**
- **moveToTable**
- **arrivedAtTable**
<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>

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>monitored</th>
<th>cold</th>
<th>hot</th>
</tr>
</thead>
<tbody>
<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 occurs in the system that can be unified with the first message in an MSD, an active copy of the MSD (also called active MSD) is created.
• The cut remembers that the first message was unified
• The cut progresses as subsequent messages are unified
• If the cut is in front of a message on its sending and receiving lifeline, the message is called *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.
• 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 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 then active MSD terminates. This is called a **cold violation**.
- violations are forbidden in a hot cut: **safety violation**

```
ArmATransportBlankToPress

ts:TableSensor               c:Controller               a:ArmA

blankArrived

pickUp

moveToPress

arrivedAtPress

releaseBlank

moveToTable

arrivedAtTable
```
Universal MSD – Semantics by Example

- An active MSD must not remain forever in an executed cut – liveness violation
- An active MSD may remain forever in a monitored cut
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 \((ts, \text{blankArrived}, c)\) as \(bA\), etc., so
\[
\Sigma_d = \{bA, pU, mTP, aAP, rB, mTT, aAT\}, \quad \Sigma_d \subseteq \Sigma
\]
Universal MSDs and Büchi Automata

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

### ArmATransportBlankToPress

<table>
<thead>
<tr>
<th>ts: TableSensor</th>
<th>c: Controller</th>
<th>a: ArmA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>blankArrived</td>
<td></td>
</tr>
<tr>
<td></td>
<td>pickUp</td>
<td>c0</td>
</tr>
<tr>
<td></td>
<td>moveToPress</td>
<td>c1</td>
</tr>
<tr>
<td></td>
<td>arrivedAtPress</td>
<td>c2</td>
</tr>
<tr>
<td></td>
<td>releaseBlank</td>
<td>c3</td>
</tr>
<tr>
<td></td>
<td>moveToTable</td>
<td>c4</td>
</tr>
<tr>
<td></td>
<td>arrivedAtTable</td>
<td>c5</td>
</tr>
</tbody>
</table>

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

```
requestEnter

enterAllowed/enterDenied

closed/blocked

rc: RailCab

rcr: CrossingControl

b: Barriers
```

```
RequestEnterAtCrossing

- requestEnter: (c/m)
- enterDenied: (h/e)
- closeBarriers: (h/e)
- barriersBlocked: (c/e)
```

```
Σ \ bA
Σ \ Σ
Σ \ Σ
d
Σ \ Σ
d
Σ \ Σ
d
Σ \ Σ
d
sv
Σ
Σ \ Σ
Σ \ Σ
eD
bB
Σ \ Σ
cB
Σ \ Σ
Σ \ Σ
c0
c1
c2
c3
Σ \ Σ
Σ \ Σ
Σ \ Σ
rE
Σ \ Σ
Σ \ Σ
Σ \ Σ
Σ \ Σ
```

```
31
```
• Example: RailCab requests permission to enter crossing

RequestEnterAtCrossing

rc: RailCab
 crc: CrossingControl
 b: Barriers

requestEnter (c/m)
 enterDenied (h/e)

closeBarriers (h/e)

barriersBlocked (c/m)

cold violation by event unifiable with non-first message.

cold violation by event unifiable with first message. Represents termination and immediate reactivation.

Universal MSDs and Büchi Automata Cont.
Iterative vs. Invariant Semantics

• what is supposed to happen here?

• “…if entering the crossing is denied, the RailCab should retry requesting the permission to enter the crossing”
  – Another active copy of the MSD is created while the previous copy terminates

• Invariant interpretation: multiple active copies, in different cuts of the same MSD are possible at the same time.
Universal MSD – Iterative vs. Invariant Semantics

- If the first message of an MSD appears also later in the diagram, there may be multiple active copies of this MSD – *invariant* interpretation

run: m1, m2, m1, ...

```
<table>
<thead>
<tr>
<th>a:A</th>
<th>b:B</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1</td>
<td></td>
</tr>
<tr>
<td>m2</td>
<td></td>
</tr>
<tr>
<td>m1</td>
<td></td>
</tr>
<tr>
<td>m3</td>
<td></td>
</tr>
</tbody>
</table>
```

invariant interpretation:

```
<table>
<thead>
<tr>
<th>a:A</th>
<th>b:B</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1</td>
<td></td>
</tr>
<tr>
<td>m2</td>
<td></td>
</tr>
<tr>
<td>m1</td>
<td></td>
</tr>
<tr>
<td>m3</td>
<td></td>
</tr>
</tbody>
</table>
```

```
<table>
<thead>
<tr>
<th>a:A</th>
<th>b:B</th>
</tr>
</thead>
<tbody>
<tr>
<td>m1</td>
<td></td>
</tr>
<tr>
<td>m2</td>
<td></td>
</tr>
<tr>
<td>m1</td>
<td></td>
</tr>
<tr>
<td>m3</td>
<td></td>
</tr>
</tbody>
</table>
```
Iterative vs. Invariant Semantics

- what is supposed to happen here?

```
Iterative interpretation: at most one active copy of an MSD
```

“...if the coffee machine button is pressed after the controller ordered the brewer unit to prepare the coffee, the coffee preparation must be stopped”

- Here we do not expect two active copies of the MSD
Universal MSD – Iterative vs. Invariant Semantics

- Only one active copy of an MSD is allowed
  - *iterative* interpretation

run: m1, m2, m1, ...

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a:A</td>
<td>b:B</td>
</tr>
<tr>
<td>m1</td>
<td></td>
</tr>
<tr>
<td>m2</td>
<td></td>
</tr>
<tr>
<td>m1</td>
<td></td>
</tr>
<tr>
<td>m1</td>
<td></td>
</tr>
<tr>
<td>m3</td>
<td></td>
</tr>
</tbody>
</table>
```

iterative interpretation:

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a:A</td>
<td>b:B</td>
</tr>
<tr>
<td>m1</td>
<td></td>
</tr>
<tr>
<td>m2</td>
<td></td>
</tr>
<tr>
<td>m1</td>
<td></td>
</tr>
<tr>
<td>m1</td>
<td></td>
</tr>
<tr>
<td>m3</td>
<td></td>
</tr>
</tbody>
</table>
```

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a:A</td>
<td>b:B</td>
</tr>
<tr>
<td>m1</td>
<td></td>
</tr>
<tr>
<td>m2</td>
<td></td>
</tr>
<tr>
<td>m1</td>
<td></td>
</tr>
<tr>
<td>m1</td>
<td></td>
</tr>
<tr>
<td>m3</td>
<td></td>
</tr>
</tbody>
</table>
```
Iterative vs. Invariant Semantics

- what is supposed to happen here?

- “...if entering the crossing is denied, the RailCab should retry requesting the permission to enter the crossing and report a problem to the area control”

- Invariant interpretation: what is the problem here?
  - inevitable safety violation after rE, cB, bB, eD, rE
Iterative vs. Invariant Semantics

• solution: specify two MSDs

Under which condition should we report a problem? For example like this...

RequestEnterAtCrossing

ReportProblemIfBarriersBlocked
Concurrently Active MSDs

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

- releaseBlank
- press
- pressingFinished
- pickUp

**ArmATransportBlankToPress**
- ts:TableSensor
- c:Controller
- a:ArmA

- blankArrived
- pickUp
- moveToPress
- arrivedAtPress
- h/e
- releaseBlank
- moveToTable
- arrivedAtTable
Concurrently Active MSDs

PressPlateAfterArmARelasesBlankPlate

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

- releaseBlank
- press
- pressingFinished
- pickUp

ArmATransportBlankToPress

- ts:TableSensor
- c:Controller
- a:ArmA

- blankArrived
- pickUp
- moveToPress
- arrivedAtPress
- releaseBlank
- h/e
- moveToTable
- arrivedAtTable
Concurrently Active MSDs

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

- releaseBlank
- press
- pressingFinished
- pickUp
- h/e

**ArmATransportBlankToPress**
- ts: TableSensor
- c: Controller
- a: ArmA

- blankArrived
- pickUp
- moveToPress
- arrivedAtPress
- releaseBlank
- moveToTable
- arrivedAtTable
- h/m
Concurrently Active MSDs

PressPlateAfterArmARelasesBlankPlate

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

- releaseBlank
- press
- pressingFinished
- pickUp

ArmATransportBlankToPress

- ts:TableSensor
- c:Controller
- a:ArmA

- blankArrived
- pickUp
- moveToPress
- arrivedAtPress
- releaseBlank
- moveToTable
- arrivedAtTable
- h/m
- h/m
Concurrently Active MSDs

PressPlateAfterArmAReleasesBlankPlate


releaseBlank  press  pressingFinished  pickUp  h/e

ArmATransportBlankToPress

ts:TableSensor  c:Controller  a:ArmA

blankArrived  pickUp  moveToPress  arrivedAtPress  releaseBlank  moveToTable  h/m  arrivedAtTable
Concurrently Active MSDs

PressPlateAfterArmARelasesBlankPlate

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

PressingFinished 
press
releaseBlank
pickUp

ArmATransportBlankToPress

- ts: TableSensor
- c: Controller
- a: ArmA

blankArrived
pickUp
moveToPress
arrivedAtPress
releaseBlank
moveToTable
h/m
arrivedAtTable
Concurrently Active MSDs

PressPlateAfterArmARelasesBlankPlate

- releaseBlank
- press
- pressingFinished
- pickUp

ArmATransportBlankToPress

- blankArrived
- pickUp
- moveToPress
- arrivedAtPress
- releaseBlank
- moveToTable
- arrivedAtTable
Concurrently Active MSDs

- Does a specification with these two MSDs accept the run m1, m2, m1, m2, ...?

- Yes: Imagine the corresponding Büchi automata, we visit an accepting state in each automaton infinitely often.

- However, there are tools that have a different interpretation:
  - infinitely often there must be no active MSDs or only active MSDs with monitored cuts
  - in this case, the run will not be accepted
  - we call this interpretation the *global acceptance condition*
Composed Controllers of the Object System

- The controller of an object system can be composed, via handshaking, of multiple smaller controllers
  - a controller can control one or many objects
  - one object can be controlled only by one controller
  - extreme: one controller per object
• Let $O_C$ be the objects controlled by a controller $C = (S, \Sigma_C, T, I)$
  
  – then $\Sigma_C = \Sigma \cap ((O_C \times \text{Name} \times O) \cup (O \times \text{Name} \times O_C))$ are the events of messages sent and received by the objects in C

• i.e., a controller cannot control messages sent and received by objects that it does not control
  
  – but a controller can control, also block, the sending as well as the receiving of messages of the objects it controls
Open Systems

- Typically, we consider systems where software can control only a subset of objects; the other objects are uncontrollable, e.g. humans, a physical environment, or external software.

- Given an object system $O$, we define $O_E$ and $O_S$, $O_E \cup O_S = O$, $O_E \cap O_S = \emptyset$
  - $O_S$ is called the controllable objects or system objects
  - $O_E$ is called the uncontrollable objects or environment objects

- Our goal is to find a controller for $O_S$ so that the global controller formed with any possible controller for objects $O_E$ satisfies the MSD specification
  - Usually we must assume that anything can happen in the environment, but we can also formulate assumptions (later)
System Objects and Environment Objects

- Example: Production Cell

Sometimes environment objects are represented with cloud-like shapes.
Synchrony Assumption

- Our goal is to find a controller for $O_S$ so that the global controller formed with any possible controller for objects $O_E$ satisfies the MSD specification.

- Sometimes we can make the following general assumption to make the task easier:
  - Be $C_E$ and $C_S$ the controllers for the environment resp. system objects, the we assume
    - if an environment event occurs, the system can take finitely many steps before it listens for the next environment event, i.e.,
      - the environment must never block system events
      - but the system must always eventually listen for environment events

Can you think of criteria for the LTSs of the environment / system controllers that ensure this?
The Play-Out Algorithm

- Our goal is to find a controller for $O_S$ so that the global controller formed with any possible controller for objects $O_E$ satisfies the MSD specification (under the synchrony assum.)

- Idea: Execute the MSDs!
  - Wait for any environment event to occur
  - While there are executed cuts, choose an enabled executed message to execute next, if it does not lead to a safety violation
  - Repeat the process.
• What is the problem when playing-out this MSD?