SE 1: Software Requirements Specification and Analysis

Lecture 5: UML and UML State and Sequence Diagrams

Nancy Day, Davor Svetinović


uw.cs.cs445
Today’s Agenda

- Basic Notations and Process (from last lecture)
- Introduction to UML and the Unified Process (UP)
- UML Sequence Diagrams
- UML State Diagrams

These diagrams capture the dynamic behaviour of the system (the reaction of the system to external events).

Reading: Arlow and Neustadt, Ch.1, 2, 12, 13.2, 21, 22
History of UML

- UML = Unified Modeling Language = collection of visual modeling diagrams

- Object-oriented analysis (OOA) grew out of desire to make requirements techniques more connected to object-oriented design techniques

- Result of effort in the early 1990’s to unify many OOA techniques

- Currently, 13 diagram types

- UML by itself is not a methodology.

A UML specification views the system as collection of interacting objects.
History of UML

- 1994 Rumbaugh (Object Modeling Techniques) joined Booch at Rational
- 1995 Rational bought Objectory (Jacobson, OOSE – use cases)
- Rumbaugh, Booch and Jacobson are known as the “Three Amigos”
- UML = OMT + Booch + OOSE + . . .
- But UML is not really a “unification” of these approaches. It’s more like the collection of these approaches
- 1997, Object Management Group (OMG) choose UML as an industry-standard OO visual modeling language
- 2004 UML 2.0
UML

- Object-oriented decomposition
- UML principles:
  - Classifier vs instance
  - Interface vs implementation
- Conceptual (static structure):
  - Class and Object Diagrams – form of ERDs
- Dynamic behaviour:
  - Interaction Diagrams (Sequence and Communication Diagrams) – forms of event traces
  - State Diagrams – form of FSMs

Diagrams provide views of the model. These diagrams are usually all developed simultaneously and must be kept consistent.

UML also contains use case diagrams, although these aren’t object-oriented.
Model Driven Architecture

MDA = Model Driven Architecture

- “Models drive production”
- Platform-independent models (PIMs) are turned into platform-specific models (PSMs) via model transformations
- Models must be complete to generate executable code
- “MDA shifts UML models from their current role as precursors to manually created source code into the primary mechanism of code production.”
Unified Process

The **Unified Process** is an *evolutionary/iterative model* of software development.

- **UP** = process that uses UML diagrams
- The **Rational Unified Process** is a particular instance of UP developed by Rational (IBM) and supported by their case tools
- Main developer: Ivar Jacobson
- Originated at Ericsson, 1967
Main ideas of UP:

- requirements and risk-driven
- architecture-driven
- iterative and incremental

Phases:

Each of these phases may contain several iterations. UP recommends having iterations that last only 2-6 weeks (and no more than 2-3 months).

Each iteration reaches a baseline.
Each iteration contains some amount of requirements, analysis, design, implementation, and test activities.
1. **Inception:**
   - vision (business goals)
   - scope
   - consider project feasibility
   - identify risks
   - commonly-used technique: use cases

2. **Elaboration:**
   - elaboration of requirements
     - capture remaining use cases
     - domain models
     - state diagrams
     - communication (collaboration) diagrams
   - create architectural baseline
   - define quality attributes.

3. **Construction:** implementation

4. **Transition:** testing, deployment, product release
The Way Forward

Today: Basics of UML Sequence and State Diagrams

Thurs: Advanced features of UML State Diagrams

Tues: Advanced features of UML State Diagrams and System State Diagrams

Thu: Reference Model for RE

Tue: Concept/Class Diagrams

Thu: Concept-level state diagrams

Note: We are not really following UP.
UML is a syntax for diagrams.

Unfortunately, UML does not include a fixed semantics for some diagram types (e.g., state diagrams).

We will present one semantics, but be aware that there are others.

In cases of ambiguity, confer with your customer to make sure you both agree on the semantics.
Sequence Diagrams

- Form of event trace (time-ordered sequence of messages)
- Dynamic view of system behaviour
- Usually, depicts a scenario (one path through a use case)
- Describe end-to-end behaviour (environmental input to output to environment)
- Easy for users to understand
Sequence Diagrams

Can be used for different levels of description:

- **System level:**
  - Show system as one lifeline
  - Show only environmental inputs and outputs

- **Concept/object level:**
  - Show each object involved in the scenario as one lifeline
  - Show interaction among objects and therefore show responsibilities of each class
1: Lift receiver
2: Dial tone
3: Dial digits
4: Route call
5: Establish conn
6: Ring phone
7: Ring tone
8: Answer
9: Stop ringing
10: Stop ring tone
11: Hangup
12: Disconnect
Sequence Diagrams: Notation

- Column = represents lifeline of object or the whole system (or subsystem)
  - column label is “name:class” (name is optional)
- Rectangle on lifeline is the focus of control or activation - the period during which the object is involved in the activity initiated at the top of the focus.
- Call to self can be shown with a nested focus of control
- Horizontal arrow expresses message conveyed by source to target:
  - sending a message
  - calling an operation (must have correct parameters)
  - creating or destroying an instance
- Messages may be sent to a class (still in UML 2.0??)
Elaboration

As you elaborate the problem domain and specify in more detail the entities that the system senses and controls, the self calls become messages to these other entities.

- create transient objects in response to a «create».

- destroy transient objects either because it receives a «destroy» message or because it destroys itself.
Ex: Telephone Switch

s: caller

:exchange

--

lift receiver

--

dial tone

--

dial(digits)

--

route call

<<create>>

:Connection

--

ring tone

--

stop ring tone

--

disconnect

--

callee_disconnect

--

ring phone

--

answer

--

stop ringing

--

hangup

--

t: callee
Advanced Features

These are not necessarily recommended: keep it simple and clear!

- Can show states on the lifeline
  - This is useful for connecting sequence diagram with a state diagram, but I wouldn’t put this in your SRS.

- Can distinguish between synchronous (wait for a reply) and asynchronous messages – ignore this

- Timing constraints: These can be useful for illustrating

\[(B - A \leq 1 \text{ day})\]

real-time constraints.
Branching

opt [condition]

means

if (condition1) then
contents of box
Branching

\[ \text{Alt} \]

\[
\begin{align*}
\text{[condition1]} & \\
\text{op1()} & \\
\end{align*}
\]

\[
\begin{align*}
\text{[condition2]} & \\
\text{op2()} & \\
\end{align*}
\]

\[
\begin{align*}
\text{[else]} & \\
\text{op3()} & \\
\end{align*}
\]

means

if (condition1) then
  contents of box1
else if (condition2) then
  contents of box2
else
  contents of box3
Loops

loop [condition]

-op1()

means

while (condition) then

contents of box
Interaction Occurrences

ref Seq1

Seq1

op1

op2
SE 1 Process

1. For each use case, create a system sequence diagram for the main scenario.
   - One lifeline for the system; one for each actor.
   - Show inputs and outputs.

2. Next, we need to create a state diagram that describes all the scenarios of the system.
Example

- **mode**
- **but2**: 12/24hr
  - start/stop
- **but3**: light
  - lap
  - reset
UML State Diagrams

- Form of hierarchical finite state machine
- Represent dynamic behaviour of the system
- Represents many behaviours succinctly
- From Harel’s statecharts
Mealy Machines

For requirements specification, input characters are possible events that can occur that the system should react to.

Output characters are actions that the system can take (and possibly the state).

States represent a history of what’s happened so far in system behaviour.
Extended Finite State Machines

An extended finite state machine is one that includes variables.

Transitions can depend on the value of conditions (expressions on variables).

Outputs can be actions can be assignments of values to variables.

These are also called state diagrams or state transition diagrams.
State diagrams can be used to describe behaviour at different levels of detail:

- **System state diagrams**
  - Show how the entire system changes over time and produces output depending on the input it has received from its environment
  - Show the behaviour of the system across several use cases or sequence diagrams

- **Class-level state diagrams**
  - Show how an object’s behaviour changes over time receiving and sending messages (operation calls) to other objects
  - Describe how one object contributes to system behaviour
Basic Components of a State Diagram

- Basic states (later we will see hierarchical states)
- Transitions between states labelled with:
  - Events
  - Conditions
  - Actions

All parts of labels on transitions are optional.
States

- States partition the behaviour of the system:
  - At different states, the system reacts differently (or not at all) to different events.
  - Affects what input the object will react to, e.g., ignoring most input in the state **OFF**.

- A state represents the history of inputs so far.

- For now, a state represents a moment in time when the system does not change and is waiting for another input before the system changes.

- In response to events and conditions, the system follows transitions to change states.
For RE, the names of states should be meaningful: a state represent a “mode” of the system. These names should make sense to the customer.

The partitioning of behaviour provided by the states helps us to understand the system.

A state can also represent an output, e.g., a state of “Show Time”.

States
States

There always needs to be a designated starting/initial state. The designator of an initial state is called a pseudostate. A pseudostate is NOT a real state (no time is spent there).

Often there is a designated final state. This is a real state.

X is the initial State

Y is a final state
**Events**

- An *event* is “a significant or noteworthy occurrence”. (Larman)

- An event is an input (message) from the environment. (Later, we will see it could be a message/operation call from another object or the passage of time.)

- An event represents a change in the environment, e.g., doorOpened, doorClosed, buttonPressed, etc.

- An event is considered to occur instantaneously – it does not persist.

- Multiple events on a transition label are ORed together!
Conditions

- A condition is a Boolean expression, which is true or false depending on the value of variables.

- The value of a condition persists until the variables involved in the condition change their values.

Examples:
- \( x > 10 \)
- \( \text{doorIsClosed} \)

- Conditions on multiple transitions leaving a state should be mutually exclusive.
Actions

- Actions are what the system does in response to events (in addition to changing state.)

- Most common actions:
  - Send a message/event to the environment
    Example: setTone(…)
  - Change the value of a variable
    Example: x := 5

- An action is non-interruptible. It completes before the destination state of the transition is entered.

- Multiple actions on a transition are separated by “;” and executed sequentially.

Note: UML2 uses something more complicated than actions. We will use actions to make the formalism simpler.
Transitions

Semantics: When in state X, if event e occurs and condition c is true, carry out and complete action a and move to state Y.

If in a state and there is not an outgoing transition triggered by an event, the event is ignored.
States

States make the requirements easier to understand by partitioning the behaviour of the system into modes:

- The reaction of the system to the same event may be different in different states.

- In some states, there may be no response to certain events.
Validation

Given the list of possible events, for each state, consider whether each event \( e \) is possible in each state \( X \). It could be the case:

1. There is a transition on \( e \) from state \( X \).

2. Event \( e \) cannot physically occur in state \( X \) (e.g., doorOpened cannot occur when the door is open) – no transition on \( e \) is needed.

3. Event \( e \) is possible but the system should ignore it (i.e., the system does not change if event \( e \) occurs in state \( X \)) – no transition on \( e \) is needed.

4. If event \( e \) occurs, it is an error, but this error should result in the system producing an error message – a transition is needed.
Common Problems

- **Overspecification:** responding to an event that cannot occur at a state, thereby not taking advantage of the partitioning of behaviour provided by states. (This is more of a problem when there are hierarchical states.)

  Because a state represents the history of inputs, it represents a precondition. Overspecification means not specifying these preconditions.

- **Underspecification:** not responding to an event that is relevant at a state, thereby leaving out requirements of the system.
Types of State Diagrams

New to UML 2 is a distinction between:

- **behavioural state machines**
  - what’s just been described

- **protocol state machines**
  - no actions or activities
  - only shows proper order of operation calls
  - transition syntax:
    - [precondition] event1, event2/[postcondition]

We will concentrate on behavioural state machines.
# State vs. Sequence Diagrams

<table>
<thead>
<tr>
<th>State Diagrams</th>
<th>Sequence Diagrams</th>
</tr>
</thead>
<tbody>
<tr>
<td>specifies behaviour</td>
<td>illustrates behaviour</td>
</tr>
<tr>
<td>object states, reactions to events</td>
<td>communication among several objects</td>
</tr>
<tr>
<td>complete object behaviour</td>
<td>individual scenarios in end-to-end behaviour (better feel for overall system behaviour)</td>
</tr>
<tr>
<td>developer oriented</td>
<td>customer oriented</td>
</tr>
<tr>
<td></td>
<td>can help developer validate state diagrams</td>
</tr>
</tbody>
</table>
Summary

- **UML state diagram** = finite state machine
- **UML sequence diagram** = event trace

UML state diagram is the specification.

Sequence diagrams illustrate the specification for particular scenarios (one path through the state diagrams for each object).

**Next Lecture:** UML State Diagrams Part II

**Reading:** No additional readings