Overview

‣ In the last chapter:
  • The development process (overview)

‣ In this chapter:
  • Specification
  • State machines and modeling languages
  • UML state charts and sequence diagrams
  • SDL and MSC (Part II)
What are we modeling?

<table>
<thead>
<tr>
<th>Transitional Systems</th>
<th>Reactive Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>input-output transformation</td>
<td>event-driven</td>
</tr>
<tr>
<td>e.g. scientific computation, compilers</td>
<td>e.g. <strong>communication protocols</strong>, operating systems, control systems</td>
</tr>
<tr>
<td>correctness criteria:</td>
<td>correctness criteria:</td>
</tr>
<tr>
<td>- termination</td>
<td>- non-termination under normal conditions</td>
</tr>
<tr>
<td>- correctness of input-output transformation</td>
<td>- correctness of event-response actions</td>
</tr>
</tbody>
</table>

formal models describe event-response sequences, including state information

[S. Leue, Design of Reactive Systems, Lecture Notes, 2002]
Specification with State Machines

› A protocol interacts with the environment
  • triggered by events
  • responds by performing actions
  • behaviour depends on the history of past events, i.e. the state

![State Machine Diagram]

› state machines do not model the data flow, but the flow of control
Specification with State Machines

- Why state machines?
  ...and not programming languages?
  - lack of formal semantics
  - risk of overspecification
  - requirements specification should be kept implementation-independent
Finite State Machines

› Acceptors (sequence detectors)
  • produce a binary output (yes/no) on an input sequence
  • accept regular languages

› Transducers
  • Mealy machines (output determined on current state and input)
  • Moore machines (output determined on current state)
  • both models are equivalent
Mealy Machines, Example

**State Diagram**

- States: $q_0$, $q_1$, $q_2$
- Transitions:
  - $q_0$ to $q_0$: $1/1$
  - $q_0$ to $q_1$: $0/0$
  - $q_0$ to $q_2$: $0/0$
  - $q_1$ to $q_0$: $1/0$
  - $q_1$ to $q_2$: $1/1$
  - $q_2$ to $q_0$: $0/1$

**State Transition Table**

<table>
<thead>
<tr>
<th>State</th>
<th>In</th>
<th>Out</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>0</td>
<td>0</td>
<td>$q_1$</td>
</tr>
<tr>
<td>$q_0$</td>
<td>1</td>
<td>1</td>
<td>$q_0$</td>
</tr>
<tr>
<td>$q_1$</td>
<td>0</td>
<td>0</td>
<td>$q_2$</td>
</tr>
<tr>
<td>$q_1$</td>
<td>1</td>
<td>0</td>
<td>$q_1$</td>
</tr>
<tr>
<td>$q_2$</td>
<td>0</td>
<td>1</td>
<td>$q_2$</td>
</tr>
<tr>
<td>$q_2$</td>
<td>1</td>
<td>1</td>
<td>$q_0$</td>
</tr>
</tbody>
</table>
Definition of a Mealy machine

- A *Mealy state machine* is a tuple \((Q, q_0, I, O, T, G)\), where
  - \(Q\) is a finite, non-empty set of states,
  - \(q_0 \in Q\) is the initial state,
  - \(I\) is a finite set called the input alphabet,
  - \(O\) is a finite set called the output alphabet,
  - \(T\) is a transition function, \(T: S \times I \rightarrow S\), and
  - \(G\) is an output function, \(G: S \times I \rightarrow O\).
Limitations of FSMs

- **No data variables**
  
  variable values and changes have to be coded into the state space
  
  $\rightarrow$ exponential state space explosion

Example: dialing a telephone number
Limitations of FSMs

- Problem with finite memory:
  - finite variable range
  - problem when modeling communication channels:
    - size of the channel unknown
    - determining buffer size = overspecification
Limitations of FSMs

- Problem with concurrent FSMs
  - no communication channels
  - no synchronization
  - composition of interacting FSMs leads to new states and an explosion of the state space

- Communication protocols can be seen as concurrent state machines
Limitations of FSMs

- Missing abstraction, missing composition

Here, a group transition could be used.

- Dial Tone
  - Lift receiver / play dial tone
  - Dial digit

- Dialing
  - Dial digit [incomplete]
  - Dial digit [valid] / connect

- Connecting
  - Connected

- Busy
  - Busy

- Talking
  - Callee answers

- Ringing
  - Dialed number has to be coded into states!
State machines for specification

- Original FSMs are not suitable for modeling and specifying processes in distributed systems
- Extended state machine models:
  - Communicating Finite State Machines
  - Harel statecharts (superstates, concurrent states)
  - Extended Finite State Machines (variables, operations, conditions)
  - Basis for many practical modeling and specification languages such as SDL, UML.
Description Languages: Structure vs. Behaviour

- **Structural languages:**
  - describe the static, structural concept (architecture)
  - e.g. class diagrams, component diagrams

- **Behavioural languages:**
  - describe behaviour, i.e. activities, interaction
  - e.g. state machines and sequence diagrams
Description Languages: Constructive vs. reflective

- **Constructive languages:**
  - describe information for executing the model or for (executable) code generation
  - e.g. state machines

- **Reflective or assertive languages:**
  - describe views of the model, statically or during execution
  - e.g. sequence diagrams

[D. Harel: “Some thoughts on statecharts, 13 years later”, 1996]
Example: TCP state transition diagram

Example:
TCP connection state diagram

[RFC 793]
Example: TCP Sequence Diagram

Host 1

- SYN seq=x
- SYN ACK seq=y
- ACK seq=x+1
- DATA
- FIN seq=x'
- ACK
- FIN seq=y
- ACK

Host 2

connection establishment

connection termination
UML

- **Unified Modeling Language**
  - general-purpose language for modeling and specification in software engineering
  - in this context of particular interest: State machines, sequence diagrams

- **The standard**: UML 2.0 Superstructure Specification
  http://www.omg.org/spec/UML/2.0/

- see also: Lecture on *Software Design, Modelling and Analysis in UML* by Bernd Westphal, Uni Freiburg
As an example of state machine specialization, the states VerifyCard, OutOfService, and VerifyTransaction in the ATM state machine in Figure 15.42 have been specified as \{final\}, which means that they cannot be redefined (i.e., extended) in specializations of ATM. The other states can be redefined. The (verifyTransaction, releaseCard) transition has also been specified as \{final\}, meaning that the effect behavior and the target state cannot be redefined.

In Figure 15.43 a specialized ATM (which is the state machine of a class that is a specialization of the class with the ATM state machine of Figure 15.42) is defined by extending the composite state by adding a state and a transition, so that users can enter the desired amount. In addition a transition is added from an inherited state to the newly introduced state.

[UML Superstructure Specification v2.2]
States and Transitions

- **StateName**
  - simple state

- **StateName**
  - entry activity
  - exit activity
  - do activity
  - state with compartment

- **initial pseudostate**
- **history pseudostate (shallow history)**
- **history pseudostate (deep history)**

- **fork and join pseudostate**

- **final state**

- **terminate pseudostate**

- **event [guard] / action**

- **transition**
Junction pseudostates

junctions realize *merges* or *static conditional branches*

[UML Superstructure Specification v2.2]
Choice pseudostates

choices realize dynamic conditional branches

[UML Superstructure Specification v2.2]
The notation for a fork and join is a short heavy bar (Figure 15.25). The bar may have one or more arrows from source states to the bar (when representing a joint). The bar may have one or more arrows from the bar to states (when representing a fork). A transition string may be shown near the bar.

Presentation Options

If all guards associated with triggers of transitions leaving a choice Pseudostate are binary expressions that share a common left operand, then the notation for choice Pseudostate may be simplified. The left operand may be placed inside the diamond-shaped symbol and the rest of the Guard expressions placed on the outgoing transitions. This is exemplified in Figure 15.26.

Figure 15.26 - Alternative Notation for Choice Pseudostate

Multiple trigger-free and effect-free transitions originating on a set of states and targeting a junction vertex with a single outgoing transition may be presented as a state symbol with a list of the state names and an outgoing transition symbol corresponding to the outgoing transition from the junction.

Network Protocol Design and Evaluation
Stefan Rührup, Summer 2009

Computer Networks and Telematics
University of Freiburg
**Actions**

- **counter++**
  - Action

- **ack**
  - Receive signal action

- **msg**
  - Send signal action

- **ack**
  - Signal receipt triggers a transition
Example

Lynch’s protocol

\[
c := \text{get next char}
\]

\[
\text{IDLE} \quad \text{ACK} \quad c := \text{get next char} \quad \text{ACK}(c)
\]

\[
\text{ERR} \quad \text{NACK}(c) \quad \text{NACK} \quad \text{ACK}(c)
\]

\[
\text{receive signal} \quad \text{send signal}
\]

action
Composite states

Example:

[Diagram of composite state with entry points entry1, exit points exitA, exitB, substate Sin, and composite state Scomp.]

[UML Superstructure Specification v2.2]
Composite states, Example

Dialing

Start
- entry/start dial tone
- exit/stop dial tone

Partial Dial
- entry/number.append(n)
- [number.isValid()]

HiddenComposite
- entry/start dial tone
- exit/stop dial tone

HiddenComposite

[UML Superstructure Specification v2.2]
Substate entry

The diagram illustrates different types of substate entries:

- **Explicit entry**: A direct transition from one state to another.
- **Default entry**: Transition to a default state when no explicit entry is specified.
- **Entry point entry**: A state that serves as an entry point for substates.
- **Shallow history entry**: A state that restores the most recently active substate, but not recursively.
- **Deep history entry**: A state that restores the most recently active substate recursively in all sublevels.

Each entry is represented by a node, with arrows indicating the transitions between states.
In Figure 15.39 the statemachine shown in Figure 15.38 on page 562 is referenced in a submachine state, and the presentation option with the exit points on the state symbol is shown.

An example of the notation for entry and exit points for composite states is shown in Figure 15.21 on page 544.

Notation for protocol state machines

The two differences that exist for state in protocol state machines versus states in behavioral state machines, are as follows:

Several features in behavioral state machines do not exist for protocol state machines (entry, exit, do). States in protocol state machines can have an invariant. The textual expression of the invariant will be represented by placing it after or under the name of the state, surrounded by square brackets.

Figure 15.40 - State with invariant - notation

Rationale

Submachine states with usages of entry and exit points define in the corresponding state machine have been introduced in order for state machines with submachines to scale and in order to provide encapsulation.

Figure 15.39 - SubmachineState with usage of exit point

verifyCard

outOfService/

CardReleased

readAmount : ReadAmountSM

outOfService/

verifyTransaction

aborted

acceptCard/

ATM

[Network Protocol Design and Evaluation
Stefan Rührup, Summer 2009]
A classifier may have several protocol state machines. This happens frequently, for example, when a class inherits several parent classes having protocol state machine, when the protocols are orthogonal. An alternative to multiple protocol state machines can always be found by having one protocol state machine, with sub state machines in concurrent regions.

Notation

The notation for protocol state machine is very similar to the one of behavioral state machines. The keyword \{protocol\} placed close to the name of the state machine differentiates graphically protocol state machine diagrams.

15.3.7 ProtocolTransition (from ProtocolStateMachines)

Generalizations

- "Transition (from BehaviorStateMachines)" on page 572

Description

A protocol transition (transition as specialized in the ProtocolStateMachines package) specifies a legal transition for an operation. Transitions of protocol state machines have the following information: a pre-condition (guard), on trigger, and a post-condition. Every protocol transition is associated to zero or one operation (referred BehavioralFeature) that belongs to the context classifier of the protocol state machine.

The protocol transition specifies that the associated (referred) operation can be called for an instance in the origin state under the initial condition (guard), and that at the end of the transition, the destination state will be reached under the final condition (post).

Attributes

No additional attributes

Associations

- /referred: Operation\[0..*\]

This association refers to the associated operation. It is derived from the operation of the call trigger when applicable.

Example:

```
Door \{protocol\}

opened

[doorway->isEmpty()] close/

locked

unlock/

open/

create/
```

[Network Protocol Design and Evaluation
Stefan Rührup, Summer 2009]
A classifier may have several protocol state machines. This happens frequently, for example, when a class inherits several parent classes having protocol state machines, when the protocols are orthogonal. An alternative to multiple protocol state machines can always be found by having one protocol state machine, with sub state machines in concurrent regions.

**Notation**

The notation for protocol state machine is very similar to the one of behavioral state machines. The keyword \{protocol\} placed close to the name of the state machine differentiates graphically protocol state machine diagrams.

### 15.3.7 ProtocolTransition (from ProtocolStateMachines)

**Generalizations**

- "Transition (from BehaviorStateMachines)" on page 572

**Description**

A protocol transition (transition as specialized in the ProtocolStateMachines package) specifies a legal transition for an operation. Transitions of protocol state machines have the following information: a pre-condition (guard), on trigger, and a post-condition. Every protocol transition is associated to zero or one operation (referred BehavioralFeature) that belongs to the context classifier of the protocol state machine.

The protocol transition specifies that the associated (referred) operation can be called for an instance in the origin state under the initial condition (guard), and that at the end of the transition, the destination state will be reached under the final condition (post).

**Attributes**

No additional attributes

**Associations**

- /referred: Operation[0..*]
  
  This association refers to the associated operation. It is derived from the operation of the call trigger when applicable.

---

[UML Superstructure Specification v2.2]
Protocol Transitions

- Notation of transitions:
  
  \[ \text{[precondition]} \text{ event} / \text{[postcondition]} \]

- Protocol transitions have **no associated actions**
  (in contrast to state machine transitions)
Send signal actions are not modeled here.
Protocol state machines

- Protocol state machines cannot describe responses such as sending acknowledgement messages.
- Protocol state machines allow a **reflective** description of behaviour.
- For a **constructive** description, state machines should be used.
Semantic variation points

- Some UML elements have *semantic variation points*
- e.g. unexpected event reception (see UML Spec. 15.3.7)
  - What to do if there is a new message in the queue that cannot be handled?
    - ignore the event (delete the message)?
    - defer the event (leave the message in queue)?
    - raise an exception?
- e.g. concurrency: can two processes really be concurrent?
  - code generators enforce determinism
Semantic variation points

- Concurrency: Which transition is triggered first?

After event e1, states S1 and S4 are active. Assume, e2 is the next event.
Modeling example (1)

- **Modeling a telephone:**
  1. play a dial tone after the caller lifts the receiver
  2. then allow the user to dial digits
     - quit after a timeout
     - quit after invalid digit
  3. establish connection
     - play busy tone if busy
     - play ringing tone otherwise
  4. enable talking until the caller hangs up
Figure 15.41 - State machine diagram representing a state machine

- **Idle**
  - Lift receiver /get dial tone
  - Caller hangs up /disconnect

- **Active**
  - ActiveEntry
  - DialTone
    - Do/ play dial tone
  - Time-out
    - Do/ play message
    - After (15 sec.)
    - Dial digit(n)
  - Connecting
    - Dial digit(n)[valid]
    - Connect
  - Dialing
    - Dial digit(n)[invalid]
    - Do/ play message
    - After (15 sec.)
    - Dial digit(n)
  - Talking
    - Callee answers
    - Do/ play busy tone
    - Callee hangs up
  - Pinned
  - Callee answers
  - Do/ play message
  - Busy
    - Callee answers
    - Do/ play busy tone
  - Ringing
    - Callee answers /enable speech

- **Terminated**
  - Abort
  - Terminate

- **States**
  - Idle
  - Active
  - ActiveEntry
  - DialTone
  - Dialing
  - Connecting
  - Talking
  - Pinned
  - Busy
  - Ringing
  - Terminate

- **Transitions**
  - Lift receiver /get dial tone
  - Caller hangs up /disconnect
  - Time-out: After (15 sec.)
  - Dial digit(n)
  - Connecting: Dial digit(n)[valid]
  - Connect
  - Dialing: Dial digit(n)[invalid]
  - Do/ play message
  - After (15 sec.)
  - Dial digit(n)
  - Dialing: Do/ play busy tone
  - Callee hangs up
  - Talking: Callee answers
  - Do/ play busy tone
  - Callee hangs up
  - Pinned: Callee answers
  - Do/ play message
  - Busy: Callee answers
  - Do/ play busy tone
  - Ringing: Callee answers /enable speech
  - Talking: Do/ play ringing tone
  - Terminate

- **Events**
  - Do/ play message
  - Dial digit(n)
  - Connect
  - Do/ play busy tone
  - Callee answers
  - Callee hangs up
  - Pinned: Callee answers
  - Do/ play message
  - Busy: Callee answers
  - Do/ play busy tone
  - Ringing: Callee answers /enable speech
  - Talking: Do/ play ringing tone
  - Terminate
  - Abort
  - Connect
  - Do/ play busy tone
  - Callee answers
  - Callee hangs up
  - Pinned: Callee answers
  - Do/ play message
  - Busy: Callee answers
  - Do/ play busy tone
  - Ringing: Callee answers /enable speech
  - Talking: Do/ play ringing tone
  - Terminate
  - Abort
  - Terminate

- **Conditions**
  - After (15 sec.)
  - Dial digit(n)
  - Dial digit(n)[invalid]
  - Dial digit(n)[valid]
  - Connect
  - Do/ play busy tone
  - Callee answers
  - Callee hangs up
  - Pinned: Callee answers
  - Do/ play message
  - Busy: Callee answers
  - Do/ play busy tone
  - Ringing: Callee answers /enable speech
  - Talking: Do/ play ringing tone
  - Terminate
  - Abort
  - Terminate

- **Others**
  - Network Protocol Design and Evaluation
  - Stefan Rührup, Summer 2009
  - Computer Networks and Telematics
  - University of Freiburg
UML Sequence Diagrams

- Model process interaction (variant of interaction diagrams)
- Focus on message exchange
- Example:

![UML Sequence Diagram]

- Name of Interaction
- Local Attribute
- Lifeline
- Message

[UML Superstructure Specification v2.2]
Example:
Elements of Sequence Diagrams

Messages:
- Asynchronous message
- Reply
- Call (synchronous message)
Figure 14.26 - Sequence Diagram with time and timing concepts

The Sequence Diagram in Figure 14.26 shows how time and timing notation may be applied to describe time observation and timing constraints. The :User sends a message Code and its duration is measured. The :ACSystem will send two messages back to the :User. CardOut is constrained to last between 0 and 13 time units. Furthermore, the interval between the sending of Code and the reception of OK is constrained to last between \( d \) and \( 3d \) where \( d \) is the measured duration of the Code signal. We also notice the observation of the time point \( t \) at the sending of OK and how this is used to constrain the time point of the reception of CardOut.

Communication Diagrams

Communication Diagrams focus on the interaction between Life lines where the architecture of the internal structure and how this corresponds with the message passing is central. The sequencing of Messages is given through a sequence numbering scheme.

Communication Diagrams correspond to simple Sequence Diagrams that use none of the structuring mechanisms such as InteractionUses and CombinedFragments. It is also assumed that message overtaking (i.e., the order of the receptions are different from the order of sending of a given set of messages) will not take place or is irrelevant.
Constraints

The multiplicity of firstEvent must be 2 if the multiplicity of constrainedElement is 2; otherwise, the multiplicity of firstEvent is 0. (The constraint is a requirement on the duration from the execution time from (constrainedElement[1], firstEvent[1]) to (constrainedElement[2], firstEvent[2]). If the multiplicity of constrainedElement is 1, then the constraint is a requirement on the duration given by the duration of the execution of that constrainedElement.)

if (constrainedElement->size() = 2)
then (firstEvent->size() = 2)
else (firstEvent->size() = 0)

Semantics

The semantics of a DurationConstraint is inherited from Constraints. All traces where the constraints are violated are negative traces i.e., if they occur in practice the system has failed.

Notation

A DurationConstraint is shown as some graphical association between a DurationInterval and the constructs that it constrains. The notation is specific to the diagram type.

Examples

See example in Figure 13.15 on page 440 where the TimeConstraint is associated with the duration of a Message and the duration between two OccurrenceSpecifications.

Figure 13.15 - DurationConstraint and other time-related concepts

Changes from previous UML

This metaclass has been added.

13.3.11 DurationInterval

(from SimpleTime)

Generalizations

• "Interval (from SimpleTime)" on page 444

Network Protocol Design and Evaluation
Stefan Rührup, Summer 2009

Computer Networks and Telematics
University of Freiburg
TCP Example

TCP Example Diagram:

- SYN seq=x
- SYN ACK seq=y
- ACK seq=x+1
- DATA
- FIN seq=x'
- ACK
- FIN seq=y
- ACK
Examples

The Interaction described by a Communication Diagram in Figure 14.27 shows messages m1 and m3 being sent concurrently from :r towards two instances of the part s[k]:B and s[u]:B. The sequence numbers show how the other messages are sequenced. 1b.1 follows after 1b and 1b.1.1 thereafter etc. 2 follows after 1a and 1b.

Sequence expression

The sequence-expression is a dot-separated list of sequence-terms followed by a colon (':').

sequence-term  . . .  ':'

Each term represents a level of procedural nesting within the overall interaction. If all the control is concurrent, then nesting does not occur. Each sequence-term has the following syntax:

[ integer | name ] [ recurrence ]

The integer represents the sequential order of the Message within the next higher level of procedural calling. Messages that differ in one integer term are sequentially related at that level of nesting. Example: Message 3.1.4 follows Message 3.1.3 within activation 3.1. The name represents a concurrent thread of control. Messages that differ in the final name are concurrent at that level of nesting. Example: Message 3.1a and Message 3.1b are concurrent within activation 3.1. All threads of control are equal within the nesting depth.

The recurrence represents conditional or iterative execution. This represents zero or more Messages that are executed depending on the conditions involved. The choices are:

' * ' ' [ iteration-clause ] ' 

'a branch

Figure 14.27 - Communication diagram
UML Review

- Collection of diagrams and notations
- Semantics is not always clear (this is also a consequence of historical and political decisions)
- Useful for specification and documentation
- (Partly) supported by modeling tools
- Model-checking based on UML is still a research topic

more on semantics: lecture *Software Design, Modelling and Analysis in UML* by Bernd Westphal, Software Engineering workgroup, Uni Freiburg
UML Review

- UML state machines describe the behaviour in general (constructive description), used for
  - specification
  - documentation

- UML sequence diagrams describe the specific behaviour during execution (reflective description), used for
  - describing test sequences
  - visualization of simulations
  - documentation
FSM Implementation

- Generic techniques (for C++, Java, ...):
  - The nested switch/case technique
    - define a switch for the states, in each state define a switch for events
    - change of behaviour by conditional statements
  - The State Design Pattern
    - define an abstract superclass with an event handler and derive a concrete class for each state
    - associate the state with the class holding the context (the state machine)
    - change of behaviour by object change
enum State {q0, q1, q2, ...};
enum Event {e1, e2, ...};

static State s = q0;

void handle(Event e) {
    switch(s) {
    case q0:
        switch(e) {
        case e1:
            s = q1;
            break;
        case e2:
            s = q2;
            break;
        [...]
        }
        break;
    case q1:
        switch(e) {
        case e1:
            s = q2;
            break;
        case e2:
            s = q0;
            break;
        [...]
        }
        break;
    case q2:
        switch(e) {
        case e1:
            s = q1;
            break;
        case e2:
            s = q1;
            break;
        [...]
        }
        break;
    [...]
    }
}

State pattern

- Separate classes for different states
- State change by instantiating a new object
State pattern

class Context {
    private State state;
    public void setState(State s) {
        state = s;
    }
    handleEvent(Event e) {
        state.handle(e, this);
    }
}

interface State {
    public void handle(Event e, Context c)
}

class ConcreteState1 implements State {
    public void handle(Event e, Context c) {
        switch (e)
        case e1: context.setState(new State1); break;
        case e2: context.setState(new State2); break;
    }
}

class ConcreteState2 implements State {
    [...]
}

[...]
FSM Implementation

- Nested switch/case
  - suitable for small number of states and events with only few actions
  - hopefully you don’t need to program and maintain this by hand...

- State design pattern
  - generally better maintainable
  - oversized for small state machines
  - state classes can be tested separately
Automatic code generation

- Code generation from state charts
- Used in Model-driven Software Engineering

The Model-driven Software Architecture paradigm

Automatic code generation

- Code generation from state charts can be performed by tools for Model-Driven Software Engineering (MDSE), e.g. IBM/Telelogic Rhapsody
  - Graphical UML state machine modeling
  - C++/Java code generation
  - Simulation and animation
    (special instructions inserted into the code)
  - Simulation run can be shown in a sequence diagram
State machines in MDSE

Modeling state charts with Rhapsody®
State machines in MDSE

State chart animation with Rhapsody®

[IBM/Telelogic Rhapsody 7.4 Tutorial, 2008]
State machines in MDSE

Sequence diagram from state chart animation with Rhapsody®

[IBM/Telelogic Rhapsody 7.4 Tutorial, 2008]