Formal Specification of Software

UML State Machines

Bernhard Beckert

UNIVERSITÄT KOBLENZ-LANDAU
UML State Machines

Important type of UML diagrams

For modelling behaviour

- Lifecycle of objects
- Behaviour of operations

History

- Invented by D. Harel (State Charts)
- Made popular by J. Rumbaugh (OMT)
Notions Related to State Machines

- State
- Transition
- Event
- Action, Activity
- Guards
- Sending messages
- Nesting
- Concurrency
- History states
Example: Chess

A chess game consists of alternate moves of Black and White. White moves first.
Example: Chess

A chess game consists of alternate moves of Black and White. White moves first.
Example: Chess

A chess game consists of alternate moves of Black and White. White moves first.

The game can end both when it is White’s and when it is Black’s turn.
Example: Chess

A chess game consists of alternate moves of Black and White. White moves first. The game can end both when it is White’s and when it is Black’s turn.
Example: Chess

A chess game consists of alternate moves of Black and White. White moves first.
The game can end both when it is White’s and when it is Black’s turn.
The moving player can end the game: winning (checkmate),
Example: Chess

A chess game consists of alternate moves of Black and White. White moves first.
The game can end both when it is White’s and when it is Black’s turn.
The moving player can end the game: winning (checkmate),
Example: Chess

A chess game consists of alternate moves of Black and White. White moves first.
The game can end both when it is White’s and when it is Black’s turn.
The moving player can end the game: winning (checkmate), loosing (resign),
Example: Chess

A chess game consists of alternate moves of Black and White. White moves first.

The game can end both when it is White’s and when it is Black’s turn.

The moving player can end the game: winning (checkmate), loosing (resign),
Example: Chess

A chess game consists of alternate moves of Black and White. White moves first. The game can end both when it is White’s and when it is Black’s turn. The moving player can end the game: winning (checkmate), loosing (resign), or with a draw.
Example: Chess

A chess game consists of alternate moves of Black and White. White moves first. The game can end both when it is White’s and when it is Black’s turn. The moving player can end the game: winning (checkmate), loosing (resign), or with a draw.
State Machines

State Machine

Labelled, finite graph (cycles possible)
State Machines

**State Machine**

Labelled, finite graph (cycles possible)

**States**

Nodes of the graph

Labelled with: name, do-, entry-, exit-action, ...

Initial and final states have special shapes
State Machines

State Machine
Labelled, finite graph (cycles possible)

States
Nodes of the graph
Labelled with: name, do-, entry-, exit-action, ...
Initial and final states have special shapes

Transitions
Edges of the graph
Labelled with: event, guard, action, ...
When to Use State Machines

Use State Machines . . .

- at an early stage of software development
- when behaviour of an object (lifecycle) or operation is not well understood yet
When to Use State Machines

Use State Machines . . .

- at an early stage of software development
- when behaviour of an object (lifecycle) or operation is not well understood yet

Do NOT use State Machines . . .

- when several objects are involved (interaction diagrams are better)
State

Abstract view

- the same response to the same stimuli
- the same active behaviour
State

Abstract view

- the same response to the same stimuli
- the same active behaviour

Implementation view

- certain attributes have certain values
Event

Properties

- observable in the environment of the current object
- takes place at certain point in time (has no duration)
- has possibly parameters
Event

Properties
- observable in the environment of the current object
- takes place at certain point in time (has no duration)
- has possibly parameters

Role in diagram
- triggers a transition
- is “consumed” when transition is executed
- can be saved under certain circumstances
Types of Events

Signal event

An object that is dispatched (thrown) and received (caught)

Call event

 Represents the dispatch of an operation

Time event

 Represents the passage of a certain amount of time

Change event

 Represents the fact that a Boolean expression is changed to true

The expression is checked continuously (polling)
Transition

Properties

- describes change from one state to another state
- without duration when executed
Transition

Properties

- describes change from one state to another state
- without duration when executed

Role in diagram

- triggering controlled by events, guards, state exit conditions
- execution can cause actions
Example: ATM

The customer must pass authentication before withdrawing money.
Example: ATM

The customer must pass authentication before withdrawing money.
The customer must pass authentication before withdrawing money. Authentication is done by checking a PIN.
Example: ATM

The customer must pass authentication before withdrawing money. Authentication is done by checking a PIN.
Example: ATM

The customer must pass authentication before withdrawing money. Authentication is done by checking a PIN. The PIN can be correct or not.
Example: ATM

The customer must pass authentication before withdrawing money. Authentication is done by checking a PIN. The PIN can be correct or not.
Example: ATM

The customer must pass authentication before withdrawing money. Authentication is done by checking a PIN. The PIN can be correct or not. Unsuccessful attempts are counted.
The customer must pass authentication before withdrawing money. Authentication is done by checking a PIN. The PIN can be correct or not. Unsuccessful attempts are counted,
Example: ATM

The customer must pass authentication before withdrawing money.

Authentication is done by checking a PIN.

The PIN can be correct or not.

Unsuccessful attempts are counted,

If the counter exceeds a limit, the customer is rejected.
Example: ATM

The customer must pass authentication before withdrawing money. Authentication is done by checking a PIN. The PIN can be correct or not. Unsuccessful attempts are counted, If the counter exceeds a limit, the customer is rejected.
Internal Transitions

Notation

Written as

\[ \text{Event}[\text{Guard}] / \text{Action} \]

within the state box

Example

\[
\begin{array}{|c|}
\hline
\text{Authentication} \\
\hline
\text{reset/clearScreen} \\
\hline
\end{array}
\]

Difference to self transition

Entry and exit actions are not dispatched
Entry and Exit Actions

Notation

Written as

\[
\text{entry} / Action \quad \text{resp.} \\
\text{exit} / Action
\]

within the state box

Semantics

Dispatched on entering resp. exiting the state
Activities

Notation

Written as

\[ \text{do/Action} \]

within the state box

Semantics

- have duration
- can be finished by event for outgoing transitions
Example: ATM (Alternative Formalisation)

Customer at ATM

- **Authentication**
  - do/CheckPIN
  - entry/maskScreen
  - exit/unmaskScreen

- **WithdrawMoney**
  - withdraw(amount)
    - [amount <= balance]
    - /changeAccount
  - deposit(amount)
    - /changeAccount
  - close

- **Rejected**
  - when(correct)
  - when(incorrect and ErrCnt < Max)
    - /increaseErrCounter
  - when(incorrect and ErrCnt >= Max)
    - ^Authorization Failed

- **Closed**
  - close
Exercise

A student must complete the basic level before entering the advanced level.

After both levels, the student has to pass five examinations.

An examination can be retaken at most twice. After the third failed attempt the student’s registration is cancelled.
Exercise: Student

A student must complete the basic level before entering the advanced level.
Exercise: Student

A student must complete the basic level before entering the advanced level.
Exercise: Student

A student must complete the basic level before entering the advanced level.

After both levels, the student has to pass five examinations.
Exercise: Student

A student must complete the basic level before entering the advanced level. After both levels, the student has to pass five examinations.
Exercise: Student

A student must complete the basic level before entering the advanced level.

After both levels, the student has to pass five examinations.

An examination can be retaken at most twice. After the third failed attempt the student’s registration is cancelled.
Exercise: Student

A student must complete the basic level before entering the advanced level. After both levels, the student has to pass five examinations. An examination can be retaken at most twice. After the third failed attempt the student’s registration is cancelled.
Criticism

Not really a good model, because …

- the student leaves the basic level to take an exam
- the student can cheat by repeating a passed exam
- the student cannot enter parallel exams
- the student has to complete each exam once tried
- the student cannot pass exams of the advanced level while in the basic level
Advanced Constructs Can Help

- Deferred event
- Composite state
- Concurrent composite state
- Join state, Fork State
- Concurrent transition
- Junction state
- Sync state
Events: The Detailed View

1. Event is generated: Event raised (somewhere) by some action
2. Event is conveyed: Event transported to current object (transposition does not change event)
3. Event is received: Event placed on event queue of current object
4. Event is dispatched: Event de-queued from event queue (becomes current event)
5. Event is consumed: Event is processed

Events: The Detailed View

1. Event is generated: Event raised (somewhere) by some action
Events: The Detailed View

<table>
<thead>
<tr>
<th>State1</th>
<th>Ev(Arg)[Guard]/ActTrans(Arg, Arg1)</th>
<th>State2</th>
</tr>
</thead>
<tbody>
<tr>
<td>entry/ActEntry1</td>
<td>exit/ActExit1</td>
<td>entry/ActEntry2</td>
</tr>
<tr>
<td>exit/ActExit1</td>
<td>do/ActDo1</td>
<td>exit/ActExit2</td>
</tr>
<tr>
<td>do/ActDo1</td>
<td></td>
<td>do/ActDo2</td>
</tr>
</tbody>
</table>

1. **Event is generated:**
   Event raised (somewhere) by some action

2. **Event is conveyed:**
   Event transported to current object
   (transposition does not change event)
Events: The Detailed View

1. Event is generated: Event raised (somewhere) by some action
2. Event is conveyed: Event transported to current object (transportation does not change event)
3. Event is received: Event placed on event queue of current object
Events: The Detailed View

1. Event is generated: Event raised (somewhere) by some action

2. Event is conveyed: Event transported to current object (transposition does not change event)

3. Event is received: Event placed on event queue of current object

4. Event is dispatched Event de-queued from event queue (becomes current event)
Events: The Detailed View

1. Event is generated: Event raised (somewhere) by some action
2. Event is conveyed: Event transported to current object (transposition does not change event)
3. Event is received: Event placed on event queue of current object
4. Event is dispatched: Event de-queued from event queue (becomes current event)
5. Event is consumed: Event is processed
Event Processing: The Detailed View

<table>
<thead>
<tr>
<th>State1</th>
<th>Ev(Arg)[Guard]/ActTrans(Arg, Arg1)</th>
<th>State2</th>
</tr>
</thead>
<tbody>
<tr>
<td>entry/ActEntry1</td>
<td></td>
<td>entry/ActEntry2</td>
</tr>
<tr>
<td>exit/ActExit1</td>
<td></td>
<td>exit/ActExit2</td>
</tr>
<tr>
<td>do/ActDo1</td>
<td></td>
<td>do/ActDo2</td>
</tr>
</tbody>
</table>

1. check Guard – if false abort
2. abort ActDo1
3. execute ActExit1
4. execute ActTrans(Arg, Arg1) (synchronous processing, i.e. wait until finished)
5. execute ActEntry2
6. execute ActDo2
1. **check** Guard – if false **abort**
Event Processing: The Detailed View

1. **check** Guard – if false **abort**

2. **abort** ActDo1
Event Processing: The Detailed View

1. **check** Guard – if false **abort**
2. **abort** ActDo1
3. **execute** ActExit1
Event Processing: The Detailed View

1. **check** Guard – if false **abort**

2. **abort** ActDo1

3. **execute** ActExit1

4. **execute** ActTrans(Arg, Arg1)

   (synchronous processing, i.e. wait until finished)
Event Processing: The Detailed View

1. check Guard – if false abort
2. abort ActDo1
3. execute ActExit1
4. execute ActTrans(Arg, Arg1)
   (synchronous processing, i.e. wait until finished)
5. execute ActEntry2
Event Processing: The Detailed View

1. **check** Guard – if false **abort**

2. **abort** ActDo1

3. **execute** ActExit1

4. **execute** ActTrans(Arg, Arg1)
   (synchronous processing, i.e. wait until finished)

5. **execute** ActEntry2

6. **execute** ActDo2
Event Processing: Completion Event

1. wait until ActDo1 finishes (raises completion event)
2. check Guard – if false abort
3. execute ActExit1
4. execute ActTrans(Arg1)
5. execute ActEntry2
6. execute ActDo2
Event Processing: Completion Event

1. wait until $\text{ActDo1}$ finishes  (raises completion event)
Event Processing: Completion Event

1. wait until ActDo1 finishes  (raises completion event)

2. check Guard – if false abort

3. execute ActExit1

4. execute ActTrans(Arg1)

5. execute ActEntry2

6. execute ActDo2
Event Processing: Change Event

1. wait until Guard switches from false to true (raises change event)
2. abort ActDo1
3. execute ActExit1
4. execute ActTrans(Arg1)
5. execute ActEntry2
6. execute ActDo2
Event Processing: Change Event

1. wait until Guard switches from false to true (raises change event)
Event Processing: Change Event

1. **wait until** Guard **switches from** false **to** true  \( \text{(raises change event)} \)

2. **abort** ActDo1

3. **execute** ActExit1

4. **execute** ActTrans(Arg1)

5. **execute** ActEntry2

6. **execute** ActDo2
Completion Event vs. Change Event

Activity

Completion event: after activity has $\text{ActDo1}$ finished

Change event: activity $\text{ActDo1}$ is aborted
Completion Event vs. Change Event

Activity

Completion event: after activity has ActDo1 finished
Change event: activity ActDo1 is aborted

Guard

Completion event: guard checked only once (on completion of activity)
Change event: guard checked continuously
Event Processing: Deferred Events

**Special action** `defer`

`Ev/defer`

Puts event \( Ev \) in list of deferred events

Can only be used in a state (not to label a transition)
Event Processing: Deferred Events

Special action `defer`

`Ev/defer`

Puts event `Ev` in list of deferred events

Can only be used in a state  (not to label a transition)

Triggering deferred events

A deferred event is activated as soon as a state is entered where it is not deferred
Event Processing: Deferred Events

**Special action** defer

`Ev/defer`

Puts event `Ev` in list of deferred events

Can only be used in a state (not to label a transition)

**Triggering deferred events**

A deferred event is activated as soon as a state is entered where it is not deferred

**Lost events**

Events that are neither handled nor deferred in the current state are lost
**Event Processing: Deferred Events – Example**

**Scenario**

State1 is current state  
Ev1 dispatched first,  
Ev dispatched afterwards
Event Processing: Deferred Events – Example

State1
Ev1/defer
entry/ActEntry1
exit/ActExit1
do/ActDo1

Ev

Ev1

State2
entry/ActEntry2
exit/ActExit2
do/ActDo2

Scenario

State1 is current state
Ev1 dispatched first, Ev dispatched afterwards

Then ...

1. event Ev1 is deferred
Event Processing: Deferred Events – Example

Scenario

State1 is current state
Ev1 dispatched first, Ev dispatched afterwards

Then …

1. event Ev1 is deferred

2. transition from State1 to State2, consuming event Ev
**Event Processing: Deferred Events – Example**

**Scenario**

State 1 is current state
Ev 1 dispatched first, Ev dispatched afterwards

Then . . .

1. event Ev 1 is deferred
2. transition from State 1 to State 2, consuming event Ev
3. event Ev 1 is re-activated
Event Processing: Deferred Events – Example

Scenario

State1 is current state
Ev1 dispatched first, Ev dispatched afterwards

Then . . .

1. event Ev1 is deferred

2. transition from State1 to State2, consuming event Ev

3. event Ev1 is re-activated

4. transition from State2 to State1, consuming Ev1
Composite States

<table>
<thead>
<tr>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Allow to model complex behaviour</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Idea</th>
</tr>
</thead>
<tbody>
<tr>
<td>Similar sub-states are grouped into a composite state (nesting hierarchy is a tree)</td>
</tr>
<tr>
<td>Composite states can have transitions, entry/exit actions, do activities, ... (transitions can connect states from different nesting levels)</td>
</tr>
<tr>
<td>Sub-states “inherit” from the composite state</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>State Machines are in fact composite states</td>
</tr>
</tbody>
</table>
Composite States: Example

Initial, Process are sub-states of On

Initial, Process "inherit" transition switchOff
Composite States: Three Equivalent Models

**Note**

These models are equivalent if entry/exit actions and do activities of $\text{On}$ are ignored.
Composite States: Three Equivalent Models

Note

These models are equivalent if entry/exit actions and do activities of \texttt{On} are ignored
Composite States: Three Equivalent Models

Note

These models are equivalent if entry/exit actions and do activities of On are ignored.
Active states
Sub-state and composite state can be active simultaneously

“Active state” now denotes a path from a top-level state to a leaf node in the state hierarchy
Composite States: Rules for Entering States

Entering a composite state

There must be an initial sub-state

Entering a sub-state

Both the composite state and sub-state are activated

Order of entry actions: top-down
Exiting a composite state
Exit active sub-state as well.

Exiting a sub-state
Order of exit actions: bottom-up
When final state becomes active sub-state, completion event is raised
Concurrent Composite States

Regions

Concurrent parts of composite state
Are activated synchronously  (when composite state is activated)
Separated by dashed lines
Concurrent Composite States

Active state

Also called *state configuration*

Now consists of  ???
Concurrent Composite States

Active state

Also called state configuration

Now consists of a sub-tree of the state hierarchy
Concurrent Composite States: Rules for Entering

Entering a composite state
There must be an initial sub-state in each region

Entering a sub-state
There must be an initial sub-state in all other regions
Concurrent Transitions

Concurrent transition

Alternative notation for entering concurrent composite state

Uses pseudo-states “fork” and “join”
History States

When re-entering composite state, establishes the last active configuration

Outgoing transition indicates default active configuration
History States: Shallow vs. Deep

Shallow (H): Records history only of composite state is belongs to

Deep (H*): Records history of sub-states as well
Synch States

Allow to synchronise regions
Used in combination with fork and join

Junction Points

Purpose

Simplify notation, allow to “factor out” transitions
Different from fork/join
Junction Points

Example without junction point

Graph showing transitions between states with labels: e1[g1 and g3], e1[g1 and g5], e2[g2 and g4], e2[g2 and g5].
Metamodel for State Machine
PseudoStateKind

- initial
- final
- deepHistory
- shallowHistory
- join
- fork
- junction
A Constraint of the Meta Model

Constraint on context of StateMachine

A state machine is aggregated within either a classifier or a behavioural feature (e.g. an operation)
A Constraint of the Meta Model

Constraint on context of StateMachine

A state machine is aggregated within either a classifier or a behavioural feature (e.g. an operation)

context StateMachine

inv self.contextnotEmpty implies

self.context.oclIsKindOf(BehavioralFeature) or

self.context.oclIsKindOf(Classifier))

Note

Nothing said about what happens if self.context.isEmpty