Software Design, Modelling and Analysis in UML

Lecture 10: Modelling Behaviour

2016-12-01

Prof. Dr. Andreas Podelski, Dr. Bernd Westphal

Albert-Ludwigs-Universität Freiburg, Germany

Content

- What makes a class diagram a good class diagram?
- The Elements of UML 2.0 Style Cont’d
- Example: Game Architecture
- Purposes of Behavioural Models
- Constructive Behavioural Models in UML
- UML State Machines
  - Brief History
  - Syntax
  - The Basic Causality Model
Class Diagram Guidelines Ambler (2005)

• 5.3 Relationships
  
  • 112. Model Relationships Horizontally
  • 115. Model a Dependency When the Relationship is Transitory
  • 117. Always Indicate the Multiplicity
  • 118. Avoid Multiplicity “∗”
  • 119. Replace Relationship Lines with Attribute Types

![Diagram of class relationships](image-url)
Some Example Class Diagrams
Some Example Class Diagrams

More Example Class Diagrams
Example: Modelling Games

Modelling Structure: Common Architectures

- Many domains have common, canonical architectures.
- For games, for example:
Many domains have common, canonical architectures.
For games, for example:

- **Main**
  - External inputs
    - Keyboard
    - Joystick
    - ...
  - Game Logic
    - player scores
    - interface inputs/engine
      - update
      - notify
  - (Physics) Engine
    - physical objects
    - collision notification
  - Output
    - Graphics (from ASCII to bitmap; native or via API)
    - Sound
    - ...

Adept readers try to see/find/match the common architecture if they know that a model is from a particular domain.
We can do those readers a favour by grouping/positioning things in the diagram so that seeing/find/matching is easy.

---

**Example Re-Considered**

- Keyboard
- Joystick
- ...
- player scores
- interface inputs/engine
  - update
  - notify
- physical objects
- collision notification
- Graphics (from ASCII to bitmap; native or via API)
- Sound
- ...
- gameSettings
  - 0..1
  - gameConfig
    - 0..1
    - eventLog
      - 1
      - goal
        - 0..*
      - shape
        - 1
        - 1..*
      - border
        - 1
      - mainMenu
        - 1
        - introView
          - 0..1
          - 1
          - 1
          - 1
          - 1
          - 1
          - 1
          - 1
        - configMenu
          - 0..1
          - 1
          - 1
          - 1
          - 1
          - 1
          - 1
          - 1
        - menuView
          - 1
          - 1
          - 1
        - background sound
          - 1
          - 1
        - gets input from
          - particleFactory
            - 1
            - 1
            - 2..*
        - observes and updates
          - creates
          - goal
            - 0..*
        - informs
          - 1
          - 0..4
        - logs input from
          - non-deterministic sources, i.e. from humanPlayer and random appearance of powerups
          - settings for the next game match only (not saved)
Modelling Behaviour

Stocktaking...

**Have:** Means to model the *structure* of the system.
- Class diagrams graphically, concisely describe sets of system states.
- OCL expressions logically state constraints/invariants on system states.

**Want:** Means to model *behaviour* of the system.
- Means to describe how system states *evolve over time*.
  that is, to describe sets of *sequences*

\[
\sigma_0, \sigma_1, \ldots \in \Sigma^\omega
\]

of *system states.*
What Can Be Purposes of Behavioural Models?

Example: Pre-Image
(the UML model is supposed to be the blue-print for a software system).

A description of behaviour could serve the following purposes:

• Require Behaviour.
  "This sequence of inserting money and requesting and getting water must be possible."
  (Otherwise the software for the vending machine is completely broken.)
What Can Be Purposes of Behavioural Models?

Example: Pre-Image
(the UML model is supposed to be the blue-print for a software system).

A description of behaviour could serve the following purposes:

- **Require** Behaviour.
  
  “This sequence of inserting money and requesting and getting water must be possible.”
  
  (Otherwise the software for the vending machine is completely broken.)

- **Allow** Behaviour.
  
  “After inserting money and choosing a drink, the drink is dispensed (if in stock).”
  
  (If the implementation insists on taking the money first, that’s a fair choice.)

- **Forbid** Behaviour.
  
  “This sequence of getting both, a water and all money back, must not be possible.”
  
  (Otherwise the software is broken.)
What Can Be Purposes of Behavioural Models?

Example: Pre-Image
(the UML model is supposed to be the blue-print for a software system).

A description of behaviour could serve the following purposes:

- **Require** Behaviour.
  “This sequence of inserting money and requesting and getting water must be possible.”
  (Otherwise the software for the vending machine is completely broken.)

- **Allow** Behaviour.
  “After inserting money and choosing a drink, the drink is dispensed (if in stock).”
  (If the implementation insists on taking the money first, that’s a fair choice.)

- **Forbid** Behaviour.
  “This sequence of getting both, a water and all money back, must not be possible.” (Otherwise the software is broken.)

Note: the latter two are trivially satisfied by doing nothing...

What Can Be Purposes of Behavioural Models?

Example: Pre-Image
(the UML model is supposed to be the blue-print for a software system).

A description of behaviour could serve the following purposes:

- **Require** Behaviour.  
  “System definitely does this”
  “This sequence of inserting money and requesting and getting water must be possible.”
  (Otherwise the software for the vending machine is completely broken.)

- **Allow** Behaviour.  
  “System does subset of this”
  “After inserting money and choosing a drink, the drink is dispensed (if in stock).”
  (If the implementation insists on taking the money first, that’s a fair choice.)

- **Forbid** Behaviour.  
  “System never does this”
  “This sequence of getting both, a water and all money back, must not be possible.” (Otherwise the software is broken.)

Note: the latter two are trivially satisfied by doing nothing...
UML provides two visual formalisms for constructive description of behaviours:

- Activity Diagrams
- State-Machine Diagrams

We (exemplary) focus on State-Machines because

- somehow "practice proven" (in different flavours),
- prevalent in embedded systems community,
- indicated useful by Dobing and Parsons (2006) survey, and
- Activity Diagram's intuition changed (between UML 1.x and 2.x) from transition-system-like to petri-net-like...

Example state machines:

![Example state machines diagram]

---

**Course Map**
**Brief History:**

- E[n ≠ 0]/x := x + 1; n! F
- F/x := 0
- /n := ∅
Brief History:

• Rooted in Moore/Mealy machines, Transition Systems, etc.

• Harel (1987): Statecharts as a concise notation, introduces in particular hierarchical states.
Brief History:

- Rooted in Moore/Mealy machines, Transition Systems, etc.
- Manifest in tool Statemate Harel et al. (1990) (simulation, code-generation); nowadays also in Matlab/Simulink, etc.

From UML 1.x on: State Machines (not the official name, but understood: UML-Statecharts)
Brief History:

- Rooted in Moore/Mealy machines, Transition Systems, etc.
- Manifest in tool Statemate Harel et al. (1990) (simulation, code-generation); nowadays also in Matlab/Simulink, etc.
- From UML 1.x on: State Machines (not the official name, but understood: UML-Statecharts)
- Late 1990’s: tool Rhapsody with code-generation for state machines.

Note: there is a common core, but each dialect interprets some constructs subtly different Crane and Dingel (2007). (Would be too easy otherwise...)
Roadmap: Chronologically

Syntax:
(i) UML State Machine Diagrams.
(ii) Def.: Signature with signals.
(iii) Def.: Core state machine.
(iv) Map UML State Machine Diagrams to core state machines.

Semantics:
The Basic Causality Model
(v) Def.: Ether (aka. event pool)
(vi) Def.: System configuration.
(vii) Def.: Event.
(viii) Def.: Transformer.
(ix) Def.: Transition system, computation.
(x) Transition relation induced by core state machine.
(xi) Def.: step, run-to-completion step.
(xii) Later: Hierarchical state machines.

UML State Machines: Syntax
Definition. A tuple

\[ \mathcal{I} = (\mathcal{R}, \mathcal{C}, V, atr, \mathcal{E}), \quad \mathcal{E} \text{ a set of signals}, \]

is called signature (with signals) if and only if

\[ (\mathcal{R}, \mathcal{C} \cup \mathcal{E}, V, atr) \]

is a signature (as before).

Note: Thus conceptually, a signal is a class and can have attributes of plain type, and participate in associations.
**Core State Machine**

**Definition.**
A core state machine over signature \( \mathcal{S} = (\mathcal{R}, \mathcal{E}, V, \text{atr}, \mathcal{E}) \) is a tuple

\[
M = (S, s_0, \rightarrow)
\]

where
- \( S \) is a non-empty, finite set of \textbf{(basic) states},
- \( s_0 \in S \) is an \textbf{initial state},
- and

\[
\rightarrow \subseteq S \times (\mathcal{E} \cup \{\_\}) \times \text{Expr}_\mathcal{S} \times \text{Act}_\mathcal{S} \times S
\]

is a labelled transition relation.

We assume a set \( \text{Expr}_\mathcal{S} \) of boolean expressions over \( \mathcal{S} \) (for instance OCL, may be something else) and a set \( \text{Act}_\mathcal{S} \) of \textbf{actions}.
Abbreviations and Defaults in the Standard

Reconsider the syntax of transition annotations:

\[ \text{annot} ::= [\langle \text{event} \rangle] [\langle \text{event} \rangle^*] [\langle \text{guard} \rangle] [\langle \text{action} \rangle] \]

where \( \text{event} \in \mathcal{E} \), \( \text{guard} \in \text{Expr}_\mathcal{S} \), and \( \text{action} \in \text{Act}_\mathcal{S} \).
Reconsider the syntax of transition annotations:

\[ \text{annot} ::= [\langle \text{event} \rangle, \langle \text{event} \rangle^*] \ [\langle \text{guard} \rangle] \ [\text{/} \langle \text{action} \rangle] \]

where \( \text{event} \in \mathcal{E} \), \( \text{guard} \in \mathcal{E} \), \( \text{action} \in \mathcal{A} \).

What if things are missing?

\[
\begin{align*}
\text{⇝} & \quad \langle \text{true} \rangle / \text{skip} \\
\text{⇝} & \quad \langle \text{true} \rangle / \text{skip} \\
\text{⇝} & \quad \langle \text{true} \rangle / \text{skip} \\
\text{⇝} & \quad \langle \text{false} \rangle / \text{act} \\
\text{⇝} & \quad \langle \text{false} \rangle / \text{act}
\end{align*}
\]

In the standard, the syntax is even more elaborate:

- \( E(v) \) – when consuming \( E \) in object \( u \), attribute \( v \) of \( u \) is assigned the corresponding attribute of \( E \).
- \( E(v : T) \) – similar, but \( v \) is a local variable, scope is the transition
In the following, we assume that

- a UML model consists of a set $\mathcal{D}$ of class diagrams and a set $\mathcal{M}$ of state chart diagrams (each comprising one state machine $SM$).
- each state machine $SM \in \mathcal{M}$ is associated with a class $C_{SM} \in \mathcal{C}(\mathcal{I})$.

For simplicity, we even assume a bijection, i.e. we assume that each class $C \in \mathcal{C}(\mathcal{I})$ has a state machine $SM_C$ and that its class $C_{SM_C}$ is $C$.

If not explicitly given, then this one:

$$SM_0 := \{s_0\}, s_0, \text{true, skip, } s_0.$$ 

We will see later that this choice does no harm semantically.
In the following, we assume that

- a UML model consists of a set $\mathcal{D}$ of class diagrams and a set $\mathcal{M}$ of state chart diagrams (each comprising one state machine $SM$).

- each state machine $SM \in \mathcal{M}$ is associated with a class $C_{SM} \in \mathcal{C}(\mathcal{I})$.

- For simplicity, we even assume a bijection, i.e. we assume that each class $C \in \mathcal{C}(\mathcal{I})$ has a state machine $SM_C$ and that its class $C_{SM_C}$ is $C$.

If not explicitly given, then this one:

$$SM_0 := (\{s_0\}, s_0, (s_0, \_true, skip, s_0)).$$

We will see later that this choice does no harm semantically.

Intuition 1: $SM_C$ describes the behaviour of the instances of class $C$.

Intuition 2: Each instance of class $C$ executes $SM_C$.

Note: we don’t consider multiple state machines per class. We will see later that this case can be viewed as a single state machine with as many AND-states.
Towards UML State Machines Semantics:
The Basic Causality Model
“Causality model” is a specification of how things happen at run time [...].

The causality model is quite straightforward:

- Objects respond to messages that are generated by objects executing communication actions.
- When these messages arrive, the receiving objects eventually respond by executing the behavior that is matched to that message.
- The dispatching method by which a particular behavior is associated with a given message depends on the higher-level formalism used and is not defined in the UML specification (i.e., it is a semantic variation point).

The causality model also subsumes behaviors invoking each other and passing information to each other through arguments to parameters of the invoked behavior, [...].

This purely ‘procedural’ or ‘process’ model can be used by itself or in conjunction with the object-oriented model of the previous example.”
15.3.12 StateMachine (OMG, 2011b, 574)

- Event occurrences are detected, dispatched, and then processed by the state machine, one at a time.
Event occurrences are detected, dispatched, and then processed by the state machine, one at a time.

The semantics of event occurrence processing is based on the run-to-completion assumption, interpreted as run-to-completion processing.

Run-to-completion processing means that an event [...] can only be taken from the pool and dispatched if the processing of the previous [...] is fully completed.
Event occurrences are detected, dispatched, and then processed by the state machine, one at a time.

The semantics of event occurrence processing is based on the run-to-completion assumption, interpreted as run-to-completion processing.

Run-to-completion processing means that an event [...] can only be taken from the pool and dispatched if the processing of the previous [...] is fully completed.

The processing of a single event occurrence by a state machine is known as a run-to-completion step.

Before commencing on a run-to-completion step, a state machine is in a stable state configuration with all entry/exit/internal-activities (but not necessarily do-activities) completed.
Event occurrences are detected, dispatched, and then processed by the state machine, one at a time.

The semantics of event occurrence processing is based on the run-to-completion assumption, interpreted as run-to-completion processing.

Run-to-completion processing means that an event [...] can only be taken from the pool and dispatched if the processing of the previous [...] is fully completed.

The processing of a single event occurrence by a state machine is known as a run-to-completion step.

Before commencing on a run-to-completion step, a state machine is in a stable state configuration with all entry/exit/internal-activities (but not necessarily do-activities) completed.

The same conditions apply after the run-to-completion step is completed.

Thus, an event occurrence will never be processed [...] in some intermediate and inconsistent situation.
15.3.12 StateMachine (OMG, 2011b, 574)

- Event occurrences are detected, dispatched, and then processed by the state machine, one at a time.

- The semantics of event occurrence processing is based on the run-to-completion assumption, interpreted as run-to-completion processing.

- Run-to-completion processing means that an event [...] can only be taken from the pool and dispatched if the processing of the previous [...] is fully completed.

- The processing of a single event occurrence by a state machine is known as a run-to-completion step.

- Before commencing on a run-to-completion step, a state machine is in a stable state configuration with all entry/exit/internal-activities (but not necessarily do-activities) completed.

- The same conditions apply after the run-to-completion step is completed.

- Thus, an event occurrence will never be processed [...] in some intermediate and inconsistent situation.

- [IOW.] The run-to-completion step is the passage between two state configurations of the state machine.

- The run-to-completion assumption simplifies the transition function of the StM, since concurrency conflicts are avoided during the processing of event, allowing the StM to safely complete its run-to-completion step.
Event occurrences are detected, dispatched, and then processed by the state machine, one at a time.

The semantics of event occurrence processing is based on the run-to-completion assumption, interpreted as run-to-completion processing.

Run-to-completion processing means that an event [...] can only be taken from the pool and dispatched if the processing of the previous [...] is fully completed.

The processing of a single event occurrence by a state machine is known as a run-to-completion step.

Before commencing on a run-to-completion step, a state machine is in a stable state configuration with all entry/exit/internal-activities (but not necessarily do-activities) completed.

The same conditions apply after the run-to-completion step is completed.

Thus, an event occurrence will never be processed [...] in some intermediate and inconsistent situation.

[IOW.] The run-to-completion step is the passage between two state configurations of the state machine.

The run-to-completion assumption simplifies the transition function of the StM, since concurrency conflicts are avoided during the processing of event, allowing the StM to safely complete its run-to-completion step.

The order of dequeuing is not defined, leaving open the possibility of modeling different priority-based schemes.

Run-to-completion may be implemented in various ways. [...]

15.3.12 StateMachine (OMG, 2011b, 574)
Example

\begin{center}
\begin{tabular}{c|c|c}
\hline
 & C & D \\
\hline
x - Int & P & 0.1 \\
\hline
\end{tabular}
\end{center}

\begin{center}
\textbf{SM}_C:
\end{center}

\begin{center}
\begin{tikzpicture}
\node at (0,0) [circle, draw, fill=green!20, inner sep=1pt] (S1) {$s_1$};
\node at (1,0) [circle, draw, fill=green!20, inner sep=1pt] (S3) {$s_3$};
\node at (2,0) [circle, draw, fill=green!20, inner sep=1pt] (S2) {$s_2$};
\node at (3,0) [circle, draw, fill=green!20, inner sep=1pt] (S4) {$s_4$};
\draw[->] (S1) -- node [left] {$E[n \neq \emptyset]/x := x + 1, n!F$} (S3);
\draw[->] (S3) -- node [right] {$/n := b$} (S2);
\end{tikzpicture}
\end{center}

\begin{center}
\textbf{SM}_D
\end{center}

\begin{center}
\begin{tikzpicture}
\node at (0,0) [circle, draw, fill=green!20, inner sep=1pt] (S1) {$s_1$};
\node at (1,0) [circle, draw, fill=green!20, inner sep=1pt] (S3) {$s_3$};
\node at (2,0) [circle, draw, fill=green!20, inner sep=1pt] (S2) {$s_2$};
\node at (3,0) [circle, draw, fill=green!20, inner sep=1pt] (S4) {$s_4$};
\draw[->] (S1) -- node [left] {$F/x := 0$} (S3);
\draw[->] (S3) -- node [right] {$/p!F$} (S2);
\end{tikzpicture}
\end{center}
Example

\[
\begin{align*}
SM_C: & \quad E[n \neq \emptyset] / x := x + 1, n! F \\
& \quad F / x := 0 \quad s_1 \quad s_3 \\
& \quad /n := b \\
SM_D: & \quad F / \sigma_1 \quad s_1 \quad s_2 \\
& \quad \sigma_1 = (\sigma_1, \varepsilon_1)
\end{align*}
\]
Example

\[ x = \text{Int} \quad \begin{array}{c} P \end{array} \quad \frac{n}{0.1} \quad D \]

\[ \{\text{signal}\} \quad \begin{array}{c} E \end{array} \quad \; \{\text{signal}\} \quad \begin{array}{c} F \end{array} \]

\[ SM_C: \]
\[ s_1 \quad E[n \neq 0]/x := x + 1, n! F \]
\[ F/x := 0 \quad s_2 \]
\[ F/n := 0 \]

\[ SM_D \]

Example

\[ x = \text{Int} \quad \begin{array}{c} P \end{array} \quad \frac{n}{0.1} \quad D \]

\[ \{\text{signal}\} \quad \begin{array}{c} E \end{array} \quad \; \{\text{signal}\} \quad \begin{array}{c} F \end{array} \]

\[ SM_C: \]
\[ s_1 \quad E[n \neq 0]/x := x + 1, n! F \]
\[ F/x := 0 \quad s_2 \]
\[ F/n := 0 \]

\[ SM_D \]

Example

\[ x = \text{Int} \quad \begin{array}{c} P \end{array} \quad \frac{n}{0.1} \quad D \]

\[ \{\text{signal}\} \quad \begin{array}{c} E \end{array} \quad \; \{\text{signal}\} \quad \begin{array}{c} F \end{array} \]

\[ SM_C: \]
\[ s_1 \quad E[n \neq 0]/x := x + 1, n! F \]
\[ F/x := 0 \quad s_2 \]
\[ F/n := 0 \]

\[ SM_D \]
Tell Them What You’ve Told Them...

- Ambler (2005): The Elements of UML 2.0 Style.
- One rule-of-thumb: if there is a standard architecture, make it easy to recognise how the standard architecture is concretised.

- Behaviour can be modelled using UML State Machines.
- UML State Machines are inspired by Harel’s Statecharts.
- State Machines belong to Classes.
- State machine behaviour follows the Basic Causality Model of UML, in particular
  - Objects process events.
  - Objects can be stable or not.
  - Events are processed in a run-to-completion step, processing only starts when being stable.
References


