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

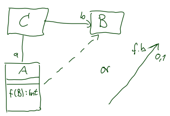
- What makes a class diagram a good class diagram?
  - The Elements of UML 2.0 Style Cont'd
  - Example: Game Architecture
- Purposes of **Behavioural Models**
- **Constructive Behavioural Models in UML**
- UML **State Machines**
  - Brief History
  - **Syntax**
  - **The Basic Causality Model**

Design Guidelines for (Class) Diagram

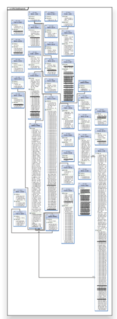
(partly following Ambler (2005))

Class Diagram Guidelines Ambler (2005)

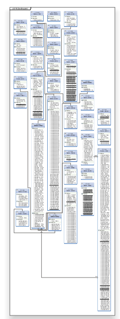
- **5.3 Relationships**
  - 112. Model Relationships Horizontally
  - 115. Model a Dependency When the Relationship is Transitory
  - 117. Always Indicate the Multiplicity
  - 118. Avoid Multiplicity "∗"
  - 119. Replace Relationship Lines with Attribute Types

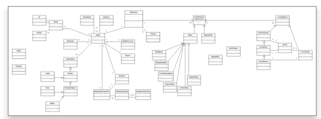


Some Example Class Diagrams

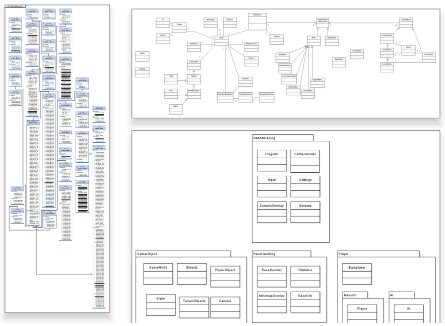


Some Example Class Diagrams

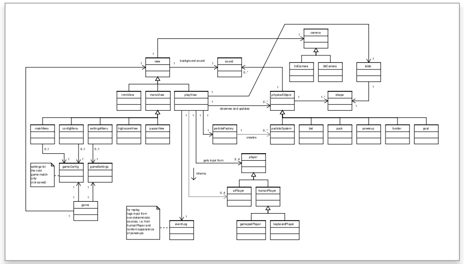




Some Example Class Diagrams



More Example Class Diagrams



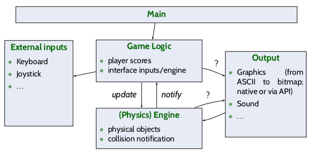
Example: Modelling Games

Modelling Structure: Common Architectures

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

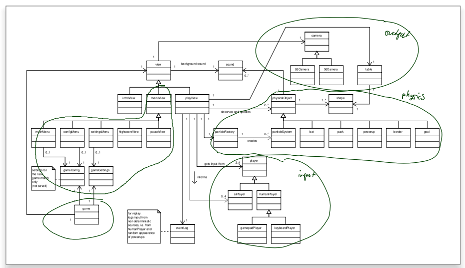
Modelling Structure: Common Architectures

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



- 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/finding/matching is easy.

Example Re-Considered



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**

$$\sigma_0, \sigma_1, \dots \in \Sigma^\omega$$

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:

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.)

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 **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...

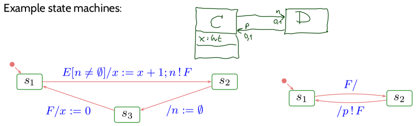
Constructive Behaviour in UML

UML provides two visual formalisms for constructive description of behaviours:

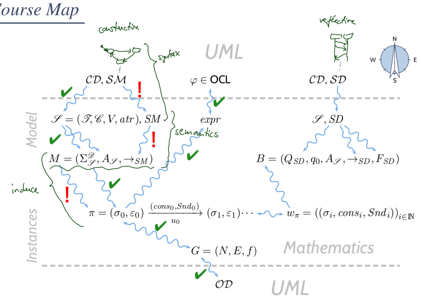
- **Activity Diagrams**
- **State-Machine Diagrams**

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:

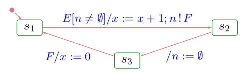


Course Map



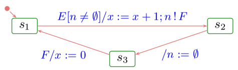
UML State Machines: Overview

UML State Machines



Brief History:

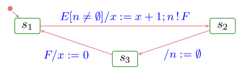
UML State Machines



Brief History:

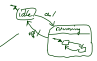
- Rooted in **Moore/Mealy machines**, Transition Systems, etc.

UML State Machines

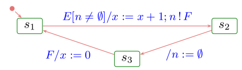


Brief History:

- Rooted in **Moore/Mealy machines**, Transition Systems, etc.
- Harel (1987): **Statecharts** as a concise notation, introduces in particular hierarchical states.



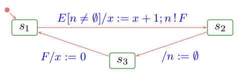
UML State Machines



Brief History:

- Rooted in **Moore/Mealy machines**, Transition Systems, etc.
- Harel (1987): **Statecharts** as a concise notation, introduces in particular hierarchical states.
- Manifest in tool **Statemate** Harel et al. (1990) (simulation, code-generation); nowadays also in **Matlab/Simulink**, etc.

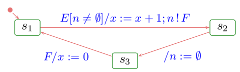
UML State Machines



Brief History:

- Rooted in **Moore/Mealy machines**, Transition Systems, etc.
- Harel (1987): **Statecharts** as a concise notation, introduces in particular hierarchical states.
- 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**)

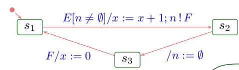
UML State Machines

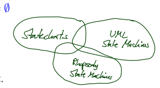


Brief History:

- Rooted in **Moore/Mealy machines**, Transition Systems, etc.
- Harel (1987): **Statecharts** as a concise notation, introduces in particular hierarchical states.
- 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.

UML State Machines





Brief History:

- Rooted in **Moore/Mealy machines**, Transition Systems, etc.
- Harel (1987): **Statecharts** as a concise notation, introduces in particular hierarchical states.
- 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...)*

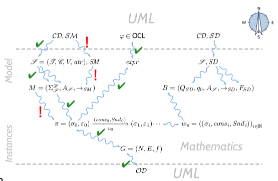
Roadmap: Chronologically

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

Signature With Signals

Definition. A tuple

$$\mathscr{S} = (\mathscr{T}, \mathscr{C}, V, atr, \mathscr{E}), \qquad \mathscr{E} \text{ a set of signals,}$$

is called signature (with signals) if and only if

$$(\mathscr{T}, \mathscr{C} \cup \mathscr{E}, V, atr)$$

is a signature (as before).

Signature With Signals

Definition. A tuple

$$\mathscr{S} = (\mathscr{T}, \mathscr{C}, V, atr, \mathscr{E}), \qquad \mathscr{E} \text{ a set of signals,}$$

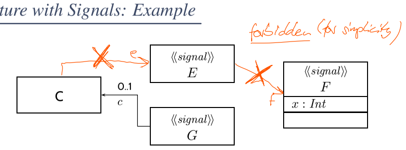
is called signature (with signals) if and only if

$$(\mathscr{T}, \mathscr{C} \cup \mathscr{E}, V, atr)$$

is a signature (as before).

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

Signature with Signals: Example



$\mathscr{S} = (\{Int\}, \{C\}, \{x : Int, c : C_{0,1}\},$
$\{C \mapsto \emptyset, E \mapsto \emptyset, G \mapsto \{c\}, F \mapsto \{x\}\},$
$\{E, F, G\})$

Core State Machine

Definition.
A core state machine over signature $\mathscr{S} = (\mathscr{T}, \mathscr{C}, V, atr, \mathscr{E})$ is a tuple

$$M = (S, s_0, \to)$$

where

- S is a non-empty, finite set of **(basic) states**,
- $s_0 \in S$ is an **initial state**,
- and

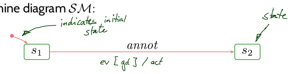
$$\to \subseteq \overset{\text{source state}}{S} \times \underbrace{(\mathscr{E} \,\dot\cup\, \{\_\})}_{\text{trigger}} \times \underbrace{Expr_{\mathscr{S}}}_{\text{guard}} \times \underbrace{Act_{\mathscr{S}}}_{\text{action}} \times \overset{\text{destination state}}{S}$$

is a labelled transition relation.

We assume a set $Expr_{\mathscr{S}}$ of boolean expressions over $\mathscr{S}$ (for instance OCL, may be something else) and a set $Act_{\mathscr{S}}$ of **actions**.

From UML to Core State Machines: By Example

UML state machine diagram $\mathcal{SM}$:



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

with

- $event \in \mathscr{E}$,
- $guard \in Expr_{\mathscr{S}}$ (**default**: *true*, assumed to be in $Expr_{\mathscr{S}}$)
- $action \in Act_{\mathscr{S}}$ (**default**: `skip`, assumed to be in $Act_{\mathscr{S}}$)

maps to

$$M(\mathcal{SM}) = \big(\underbrace{\{s_1, s_2\}}_{=S}, \underbrace{\{s_1\}}_{=s_0} \underbrace{\{(s_1, ev, gd, act, s_2)\}}_{=\rightarrow}\big)$$

Abbreviations and Defaults in the Standard

Reconsider the syntax of transition annotations:

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

where $event \in \mathscr{E}$, $guard \in Expr_{\mathscr{S}}$, $action \in Act_{\mathscr{S}}$.

Abbreviations and Defaults in the Standard

Reconsider the syntax of transition annotations:

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

where $event \in \mathscr{E}$, $guard \in Expr_{\mathscr{S}}$, $action \in Act_{\mathscr{S}}$.

What if things are missing?

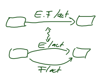
$$\begin{array}{rcl} & \rightsquigarrow & \_\ [true]\ /\ skip \\ / & \rightsquigarrow & \_\ [true]\ /\ skip \\ E\ / & \rightsquigarrow & E\ [true]\ /\ skip \\ /\ act & \rightsquigarrow & \_\ [true]\ /\ act \\ E\ /\ act & \rightsquigarrow & E\ [true]\ /\ act \end{array}$$

Abbreviations and Defaults in the Standard

Reconsider the syntax of transition annotations:

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

where $event \in \mathscr{E}$, $guard \in Expr_{\mathscr{S}}$, $action \in Act_{\mathscr{S}}$.



What if things are missing?

$$\begin{array}{rcl} & \rightsquigarrow & \_\ [true]\ /\ \mathtt{skip} \\ / & \rightsquigarrow & \_\ [true]\ /\ \mathtt{skip} \\ E\ / & \rightsquigarrow & E\ [true]\ /\ \mathtt{skip} \\ /\ act & \rightsquigarrow & \_\ [true]\ /\ act \\ E\ /\ act & \rightsquigarrow & E\ [true]\ /\ act \end{array}$$

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

State-Machines belong to Classes

In the following, we assume that

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

State-Machines belong to Classes

In the following, we assume that

- a UML model consists of a set $\mathscr{CD}$ of class diagrams and a set $\mathscr{SM}$ of **state chart diagrams** (each comprising one **state machine** $\mathcal{SM}$).
- each state machine $\mathcal{SM} \in \mathscr{SM}$ is **associated with a class** $C_{\mathcal{SM}} \in \mathscr{C}(\mathscr{S})$.
- For simplicity, we even assume a bijection, i.e. we assume that each class $C \in \mathscr{C}(\mathscr{S})$ has a state machine $\mathcal{SM}_C$ and that its class $C_{\mathcal{SM}_C}$ is C.
  If not explicitly given, then this one:

  $$\mathcal{SM}_0 := (\{s_0\}, s_0, \overset{\emptyset}{\cancel{(s_0, \_, true, \mathtt{skip}, s_0)}}).$$

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

State-Machines belong to Classes

In the following, we assume that

- a UML model consists of a set $\mathscr{CD}$ of class diagrams and a set $\mathscr{SM}$ of **state chart diagrams** (each comprising one **state machine** $\mathcal{SM}$).
- each state machine $\mathcal{SM} \in \mathscr{SM}$ is **associated with a class** $C_{\mathcal{SM}} \in \mathscr{C}(\mathscr{S})$.
- For simplicity, we even assume a bijection, i.e. we assume that each class $C \in \mathscr{C}(\mathscr{S})$ has a state machine $\mathcal{SM}_C$ and that its class $C_{\mathcal{SM}_C}$ is C.
  If not explicitly given, then this one:

$$\mathcal{SM}_0 := (\{s_0\}, s_0, (s_0, \_, \mathit{true}, \mathtt{skip}, s_0)).$$

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

Intuition 1: $\mathcal{SM}_C$ describes the behaviour of **the instances** of class C.
Intuition 2: Each instance of class C executes $\mathcal{SM}_C$.

State-Machines belong to Classes

In the following, we assume that

- a UML model consists of a set $\mathscr{CD}$ of class diagrams and a set $\mathscr{SM}$ of **state chart diagrams** (each comprising one **state machine** $\mathcal{SM}$).
- each state machine $\mathcal{SM} \in \mathscr{SM}$ is **associated with a class** $C_{\mathcal{SM}} \in \mathscr{C}(\mathscr{S})$.
- For simplicity, we even assume a bijection, i.e. we assume that each class $C \in \mathscr{C}(\mathscr{S})$ has a state machine $\mathcal{SM}_C$ and that its class $C_{\mathcal{SM}_C}$ is C.
  If not explicitly given, then this one:

$$\mathcal{SM}_0 := (\{s_0\}, s_0, (s_0, \_, \mathit{true}, \mathtt{skip}, s_0)).$$

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

Intuition 1: $\mathcal{SM}_C$ describes the behaviour of **the instances** of class C.
Intuition 2: Each instance of class C executes $\mathcal{SM}_C$.

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.

Rhapsody Demo II

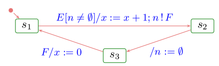
Towards UML State Machines Semantics: The Basic Causality Model

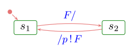
6.2.3 The Basic Causality Model (OMG, 2011b, 11)

*"'**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).





6.2.3 The Basic Causality Model (OMG, 2011b, 11)

*"'**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)

15.3.12 StateMachine (OMG, 2011b, 574)

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

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**.

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.

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**.

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.

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.

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.

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~~ stable configurations of the state machine.

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**.

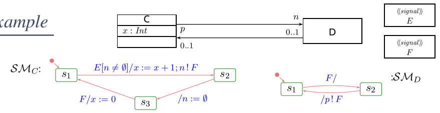
15.3.12 StateMachine (OMG, 2011b, 574)

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

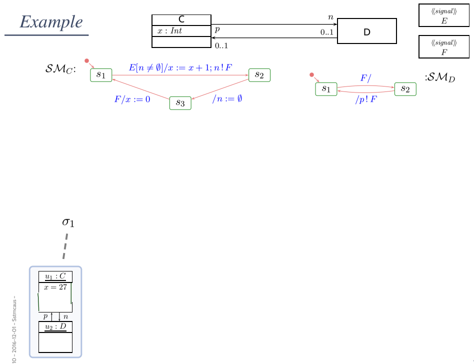
15.3.12 StateMachine (OMG, 2011b, 574)

- 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**.
- 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**. [...]

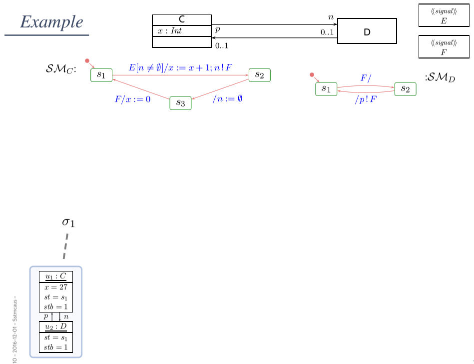
Example



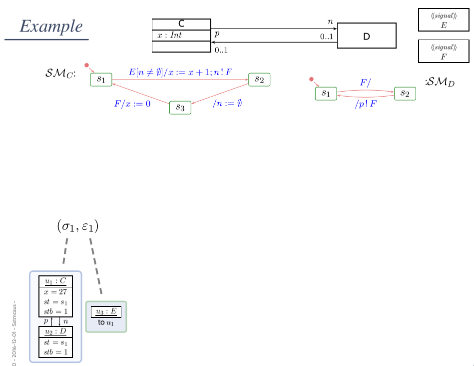
Example



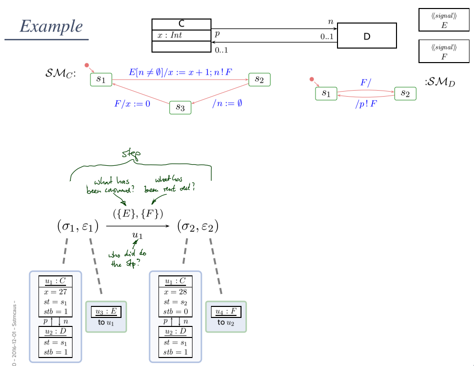
Example



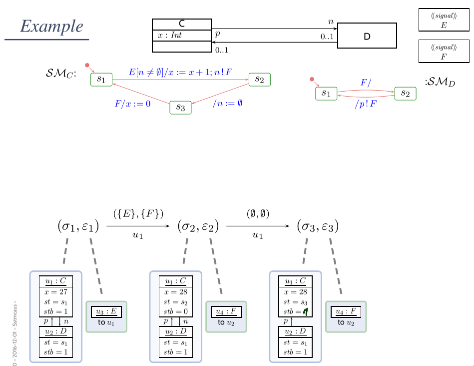
Example



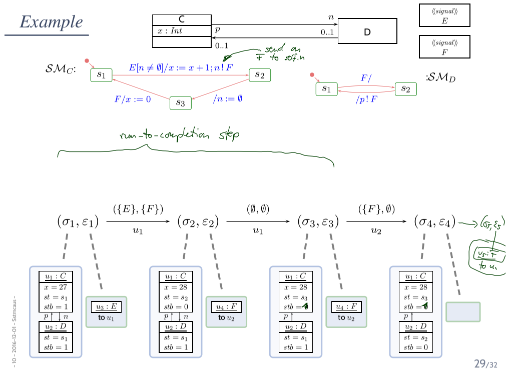
Example



Example



Example



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

References

Ambler, S. W. (2005). *The Elements of UML 2.0 Style*. Cambridge University Press.

Crane, M. L. and Dingel, J. (2007). UML vs. classical vs. rhapsody statecharts: not all models are created equal. *Software and Systems Modeling*, 6(4):415–435.

Dobing, B. and Parsons, J. (2006). How UML is used. *Communications of the ACM*, 49(5):109–114.

Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274.

Harel, D., Lachover, H., et al. (1990). Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414.

OMG (2011a). Unified modeling language: Infrastructure, version 2.4.1. Technical Report formal/2011-08-05.

OMG (2011b). Unified modeling language: Superstructure, version 2.4.1. Technical Report formal/2011-08-06.