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))
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
Example: Modelling Games
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.
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

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

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.)
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.)
**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...
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:
$\varphi \in \text{OCL}$

$\mathcal{I} = (\mathcal{T}, \mathcal{C}, V, \text{atr}), SM$

$M = (\Sigma_{\mathcal{I}}, A_{\mathcal{I}}, \rightarrow_{SM})$

$\pi = (\sigma_0, \varepsilon_0) \xrightarrow{(\text{cons}_0, \text{Snd}_0)} (\sigma_1, \varepsilon_1) \cdots$

$B = (Q_{SD}, q_0, A_{\mathcal{I}}, \rightarrow_{SD}, F_{SD})$

$w_\pi = ((\sigma_i, \text{cons}_i, \text{Snd}_i))_{i \in \mathbb{N}}$

$G = (N, E, f)$
UML State Machines: Overview
**Brief History:**
Brief History:

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

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

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

is called signature (with signals) if and only if

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

is a signature (as before).
Definition. A tuple

$$\mathcal{I} = (\mathcal{T}, \mathcal{C}, V, atr, \mathcal{E}), \quad \mathcal{E} \text{ a set of signals},$$

is called signature (with signals) if and only if

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

is a signature (as before).

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

\[ \mathcal{S} = (\{\text{Int}\}, \{\mathcal{E}\}, \{x: \text{Int}, c: \mathcal{C}_0\}, \{ \mathcal{C} \cap \emptyset, \mathcal{E} \cap \emptyset, \mathcal{G} \cap \{c\}, \mathcal{F} \cap \{\mathcal{E}\}, \{\mathcal{E}, \mathcal{F}, \mathcal{G}\}) \]
**Definition.**

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

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

where

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

$$\rightarrow \subseteq S \times (\mathcal{C} \cup \{_{-}\}) \times \text{Expr}_{\mathcal{S}} \times \text{Act}_{\mathcal{S}} \times S$$

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 actions.
UML state machine diagram $SM$:

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

with

- $event \in \mathcal{E}$,
- $guard \in \text{Expr}_{\mathcal{F}}$ (default: $true$, assumed to be in $\text{Expr}_{\mathcal{F}}$)
- $action \in \text{Act}_{\mathcal{F}}$ (default: $\text{skip}$, assumed to be in $\text{Act}_{\mathcal{F}}$)

maps to

$$M(SM) = (\{s_1, s_2\}, \{s_1\}, \{\langle s_1, ev, gd, act, s_2\rangle\})$$

$$= S = s_6 \Rightarrow$$
Reconsider the syntax of transition annotations:

\[
\text{annot} ::= \mathbf{[} \langle \text{event} \rangle \mathbf{.} \langle \text{event} \rangle \mathbf{]}^* \mathbf{]} \mathbf{[} [ \langle \text{guard} \rangle \mathbf{]} \mathbf{]} \mathbf{[} / \langle \text{action} \rangle \mathbf{]} \mathbf{]} \\
\text{where} \ \text{event} \in \mathcal{E}, \ \text{guard} \in \text{Expr} \mathcal{S}, \ \text{action} \in \text{Act} \mathcal{S}.
\]
Reconsider the syntax of transition annotations:

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

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

What if things are missing?

\[
\begin{align*}
\vdash & \quad \_ \left[ \text{true} \right] / \text{skip} \\
/ & \quad \vdash \_ \left[ \text{true} \right] / \text{skip} \\
E / & \quad \vdash E \left[ \text{true} \right] / \text{skip} \\
/ \text{act} & \quad \vdash \_ \left[ \text{true} \right] / \text{act} \\
E / \text{act} & \quad \vdash E \left[ \text{true} \right] / \text{act}
\end{align*}
\]
Reconsider the syntax of transition annotations:

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

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

What if things are missing?

\[
\begin{align*}
\_ & \rightsquigarrow \_ \langle \text{true} \rangle / \text{skip} \\
/ & \rightsquigarrow \_ \langle \text{true} \rangle / \text{skip} \\
E / & \rightsquigarrow E \langle \text{true} \rangle / \text{skip} \\
E / \text{act} & \rightsquigarrow \_ \langle \text{true} \rangle / \text{act} \\
E / \text{act} & \rightsquigarrow E \langle \text{true} \rangle / \text{act}
\end{align*}
\]

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 \( C_D \) of class diagrams and a set \( S_M \) of state chart diagrams (each comprising one state machine \( S_M \)).
- each state machine \( S_M \in S_M \) is associated with a class \( C_{S_M} \in C(S) \).
In the following, we assume that

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

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

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

If not explicitly given, then this one:

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

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 \( C \mathcal{D} \) of class diagrams and a set \( \mathcal{SM} \) of state chart diagrams (each comprising one state machine \( SM \)).

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

- For simplicity, we even assume a bijection, i.e. we assume that each class \( C \in C(S) \) 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 \( SM_C \).
State-Machines belong to Classes

In the following, we assume that

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

- each state machine $SM \in \mathcal{SM}$ 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.
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).
“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.”

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

\[ E[n \neq \emptyset]/x := x + 1; n! F \]

\[ F/x := 0 \]

\[ /n := \emptyset \]

\[ \langle \langle \text{signal} \rangle \rangle \]

\[ \langle \langle \text{signal} \rangle \rangle \]

\[ F \]

\[ /p! F \]

\[ SM_C: \]

\[ SM_D: \]

\[ C \]

\[ x : \text{Int} \]

\[ p \]

\[ 0..1 \]

\[ D \]

\[ n \]

\[ 0..1 \]
Example

$SM_C:\quad E[n \neq \emptyset]/x := x + 1; n! F$

$SM_D:\quad F/\quad /p! F$

$\sigma_1$

$\begin{array}{l}
u_1 : C \\
x = 27 \\
p \uparrow \quad n \\
u_2 : D \end{array}$
Example

\[ C \]
\[ x : \text{Int} \]
\[ p \]
\[ 0..1 \]

\[ D \]
\[ n \]
\[ 0..1 \]

\[ \langle \langle \text{signal} \rangle \rangle \]
\[ E \]
\[ F \]

\[ SMC: \]

\[ s_1 \]
\[ E[n \neq \emptyset]/x := x + 1; n! F \]

\[ s_2 \]

\[ s_3 \]
\[ F/x := 0 \]
\[ /n := \emptyset \]

\[ SM_D \]

\[ s_1 \]
\[ F/ \]
\[ /p! F \]

\[ s_2 \]

\[ \sigma_1 \]

\[ u_1 : C \]
\[ x = 27 \]
\[ st = s_1 \]
\[ stb = 1 \]
\[ p \]
\[ n \]

\[ u_2 : D \]
\[ st = s_1 \]
\[ stb = 1 \]
Example

\[ \mathbb{S}_C: \]

\[ \mathbb{S}_D: \]

\( (\sigma_1, \varepsilon_1) \)

\( u_1 : C \)

\[
\begin{align*}
  x & = 27 \\
  st & = s_1 \\
  stb & = 1
\end{align*}
\]

\( p \downarrow n \)

\( u_2 : D \)

\[
\begin{align*}
  st & = s_1 \\
  stb & = 1
\end{align*}
\]

\( u_3 : E \)

\( \text{to} \ u_1 \)
Example

\[ E[n \neq \emptyset]/x := x + 1; n! F \]

\[ F/x := 0 \]

\[ /n := \emptyset \]

\[ (\sigma_1, \varepsilon_1) \xrightarrow{u_1} (\sigma_2, \varepsilon_2) \]

\[ u_1 : C \]
\[ x = 27 \]
\[ st = s_1 \]
\[ stb = 1 \]
\[ p \]
\[ n \]

\[ u_3 : E \]
\[ x = 28 \]
\[ st = s_2 \]
\[ stb = 0 \]
\[ p \]
\[ n \]

\[ u_2 : D \]
\[ st = s_1 \]
\[ stb = 1 \]
Example

$SM_C$:

$E[n \neq \emptyset]/x := x + 1; n!F$

$F/x := 0$

$\langle \langle \text{signal} \rangle \rangle$

$E$

$\langle \langle \text{signal} \rangle \rangle$

$F$

$\langle \langle \text{signal} \rangle \rangle$

$F$

$\langle \langle \text{signal} \rangle \rangle$

$SU_C$

$s_1$

$s_2$

$s_3$

$s_1$

$s_2$

$SM_D$

$(\sigma_1, \varepsilon_1)$

$(\{E\}, \{F\})$

$(\sigma_2, \varepsilon_2)$

$(\emptyset, \emptyset)$

$(\sigma_3, \varepsilon_3)$

$u_1$

$u_1$

$u_1$

$u_1$

$(x = 27, st = s_1, stb = 1)$

$(x = 28, st = s_2, stb = 0)$

$(x = 28, st = s_3, stb = 1)$

$(x = 27, st = s_1, stb = 1)$

$(x = 28, st = s_3, stb = 1)$

$(x = 28, st = s_3, stb = 1)$

$(x = 28, st = s_3, stb = 1)$

$(x = 28, st = s_3, stb = 1)$

$(x = 28, st = s_3, stb = 1)$

$(x = 28, st = s_3, stb = 1)$

$(x = 28, st = s_3, stb = 1)$

$(x = 28, st = s_3, stb = 1)$

$(x = 28, st = s_3, stb = 1)$

$(x = 28, st = s_3, stb = 1)$
Example

\[ SMC: \]

\[ E[n \neq \emptyset] \backslash x := x + 1; n ! F \]

\[ F/x := 0 \]

\[ /n := \emptyset \]

\[ \text{run-to-completion step} \]

\[ (\sigma_1, \varepsilon_1) \rightarrow (\{ E \}, \{ F \}) \rightarrow u_1 \]

\[ (\sigma_2, \varepsilon_2) \rightarrow (\emptyset, \emptyset) \rightarrow u_1 \]

\[ (\sigma_3, \varepsilon_3) \rightarrow (\{ F \}, \emptyset) \rightarrow u_2 \]

\[ (\sigma_4, \varepsilon_4) \rightarrow (\sigma_5, \varepsilon_5) \rightarrow u_1 \]

\[ u_1 : C \]

\begin{align*}
&x = 27 \\
&st = s_1 \\
&stb = 1 \\
&\quad p \quad n
\end{align*}

\[ u_2 : D \]

\begin{align*}
&st = s_1 \\
&stb = 1
\end{align*}

\[ u_3 : E \]

\begin{align*}
&x = 28 \\
&st = s_2 \\
&stb = 0 \\
&\quad p \quad n
\end{align*}

\[ u_4 : F \]

\begin{align*}
&x = 28 \\
&st = s_3 \\
&stb = 0 \\
&\quad p \quad n
\end{align*}

\[ u_5 : F \]

\begin{align*}
&x = 28 \\
&st = s_3 \\
&stb = 0 \\
&\quad p \quad n
\end{align*}
Tell Them What You’ve Told Them...

- 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


