Software Design, Modelling and Analysis in UML
Lecture 14: Hierarchical State Machines I

2016-12-22

Prof. Dr. Andreas Podelski, Dr. Bernd Westphal
Albert-Ludwigs-Universität Freiburg, Germany

Content

- **Missing Pieces**: Create and Destroy Transformers
- **Putting It All Together (Again)**
  - Initial States
  - Consistency wrt. OCL Constraints
- **Hierarchical State Machines**
  - Overview
  - **Abstract Syntax**: States
    - pseudo-states, regions, ...
  - (Legal) System Configurations
  - **Abstract Syntax**: Transitions
  - **Enabledness of Fork/Join Transitions**
    - scope, priority, maximality, ...
**Initial States**

**Recall:** a labelled transition system is \((S, A, \rightarrow, S_0)\).
We have

- \(S\): system configurations \((\sigma, \varepsilon)\)
- \(\rightarrow\): labelled transition relation \((\sigma, \varepsilon) \xrightarrow{\text{cons, Snd}} (\sigma', \varepsilon')\).

**Wanted:** initial states \(S_0\).

**Proposal:**
Require a (finite) set of **object diagrams** \(\mathcal{OD}\) as part of a UML model

\((\mathcal{C}, \mathcal{M}, \mathcal{OD})\).

And set

\[ S_0 = \{ (\sigma, \varepsilon) \mid \sigma \in G^{-1}(\mathcal{OD}), \quad \mathcal{OD} \in \mathcal{OD}, \quad \varepsilon \text{ empty} \}. \]

**Other Approach:** (used by Rhapsody tool) multiplicity of classes (plus initialisation code).
We can read that as an abbreviation for an object diagram.
The semantics of the UML model

\[ \mathcal{M} = (\mathcal{C}, \mathcal{S}, \mathcal{O}) \]

where

- some classes in \( \mathcal{C} \) are stereotyped as ‘signal’ (standard), some signals and attributes are stereotyped as ‘external’ (non-standard).
- there is a 1-to-1 relation between classes and state machines,
- \( \mathcal{O} \) is a set of object diagrams over \( \mathcal{C} \).

is the transition system \((S, A, \rightarrow, S_0)\) constructed on the previous slide(s).

The computations of \( \mathcal{M} \) are the computations of \((S, A, \rightarrow, S_0)\).

---

**OCL Constraints and Behaviour**

- Let \( \mathcal{M} = (\mathcal{C}, \mathcal{S}, \mathcal{O}) \) be a UML model.
- We call \( \mathcal{M} \) consistent iff, for each OCL constraint \( expr \in \text{Inv}(\mathcal{C}) \),

  \[ \sigma \models expr \]

  for each “reasonable point” \((\sigma, \epsilon)\) of computations of \( \mathcal{M} \).

(Cf. tutorial for discussion of “reasonable point”)

**Note:** we could define \( \text{Inv}(\mathcal{S}) \) similar to \( \text{Inv}(\mathcal{C}) \).
Last Missing Piece: Create and Destroy Transformer

Transformer: Create

<table>
<thead>
<tr>
<th><strong>abstract syntax</strong></th>
<th><strong>concrete syntax</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>create(C, expr, v)</code></td>
<td><code>expr, v = new C</code></td>
</tr>
</tbody>
</table>

**intuitive semantics**
Create an object of class $C$ and assign it to attribute $v$ of the object denoted by expression $expr$.

**well-typedness**
- $expr : T_D, v \in atr(D)$,
- $atr(C) = \{(v_i : T_l, expr_i^0) | 1 \leq i \leq n\}$

**semantics**
...\[...

**observables**
...\[...

**{error} conditions**
$I[expr](\sigma, \beta)$ not defined.

$x = (\text{new } C), y = (\text{new } D), z,$
- can be written as
- $expr, z = \text{new } C$,
- $expr, y = \text{new } D$,
- $x = \text{time}, y = \text{time}, z$.
abstract syntax  
create(C, expr, v)  

concrete syntax  
create(C, expr, v)  

intuitive semantics  
Create an object of class C and assign it to attribute v of the object denoted by expression expr.  

well-typedness  
expr : T_D, v ∈ atr(D), atr(C) = \{(v_1 : T_1, expr_1^0) | 1 ≤ i ≤ n\}  

semantics  
...  

observables  
...  

(error) conditions  
I[expr](σ, β) not defined.  

How To Choose New Identities?  

• Re-use: choose any identity that is not alive now, i.e. not in dom(σ).  
  • Doesn’t depend on history.  
  • May ‘undangle’ dangling references – may happen on some platforms.  

• Fresh: choose any identity that has not been alive ever, i.e. not in dom(σ) and any predecessor in current run.  
  • Depends on history.  
  • Dangling references remain dangling – could mask “dirty” effects of platform.  

• We use an “and assign”-action for simplicity – it doesn’t add or remove expressive power, but moving creation to the expression language raises all kinds of other problems since then expressions would need to modify the system state.  
• Also for simplicity: no parameters to construction (~ parameters of constructor). Adding them is straightforward (but somewhat tedious).
Create an object of class $C$ and assign it to attribute $v$ of the object denoted by expression $expr$.

**Abstract Syntax**: $\text{create}(C, expr, v)$

**Concrete Syntax**: $\text{create}(C, expr, v)$

**Intuitive Semantics**

Create an object of class $C$ and assign it to attribute $v$ of the object denoted by expression $expr$.

**Well-typedness**

$\text{expr} : T_D, v \in \text{attr}(D)$,

$\text{attr}(C) = \{ (v_1 : T_1, \text{expr}_1^0) \ | \ 1 \leq i \leq n \}$

**Semantics**

$((\sigma, \varepsilon), (\sigma', \varepsilon')) \in t_{\text{create}(C, expr, v)[u_x]}$ implies

$\sigma' = \sigma[u_0 \mapsto \sigma(u_0)[v \mapsto u]] \cup \{ u \mapsto \{ v_i : d_i \ | \ 1 \leq i \leq n \} \}$,

$\varepsilon' = [u_0][\varepsilon] ; u \in \mathcal{D}(C)$ fresh, i.e. $u \notin \text{dom}(\sigma)$;

$u_0 = I[\text{expr}](\sigma, u_x) ; d_i = I[\text{expr}_i^0](\sigma, \emptyset)$ if $\text{expr}_i^0 \neq \emptyset$ and arbitrary value from $\mathcal{D}(T_i)$ otherwise.

**Observables**

$\text{Obs}_{\text{create}[u_x]} = \{ (\ast, u) \}$

**Error Conditions**

$I[\text{expr}](\sigma, u_x)$ not defined.

---

**Create Transformer Example**

$SM_D$: $/n := \text{new } C$

Create $C$, $expr$, $v$

$t_{\text{create}(C, expr, v)[u_x]}(\sigma, \varepsilon) = ...$
**Transformer: Destroy**

<table>
<thead>
<tr>
<th>Abstract Syntax</th>
<th>Concrete Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>destroy(expr)</code></td>
<td></td>
</tr>
</tbody>
</table>

**Intuitive Semantics**

Destroy the object denoted by expression `expr`.

**Well-Typedness**

\[ expr : T_C, C \in \mathcal{C} \]

**Semantics**

...  

**Observables**

\[ \text{Obs}_{\text{destroy}}[u_x] = \{(u_x, \perp, (+, \emptyset), u)\} \]

**Error Conditions**

\[ I[expr](\sigma, \beta) \text{ not defined.} \]

---

**What to Do With the Remaining Objects?**

Assume object \( u_0 \) is destroyed...

- object \( u_1 \) may still refer to it via association \( r \):  
  - allow dangling references?  
  - or remove \( u_0 \) from \( \sigma(u_1)(r) \)?

- object \( u_0 \) may have been the last one linking to object \( u_2 \):  
  - leave \( u_2 \) alone?  
  - or remove \( u_2 \) also? (garbage collection)

- Plus: (temporal extensions of) OCL may have dangling references.

**Our choice**: Dangling references and no garbage collection!  
This is in line with “expect the worst”, because there are target platforms which don’t provide garbage collection – and models shall (in general) be correct without assumptions on target platform.

**But**: the more “dirty” effects we see in the model, the more expensive it often is to analyse. Valid proposal for simple analysis: monotone frame semantics, no destruction at all.
Destroy Transformer: Destroy

<table>
<thead>
<tr>
<th>abstract syntax</th>
<th>concrete syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>destroy(expr)</code></td>
<td><code>destroy(expr)</code></td>
</tr>
</tbody>
</table>

intuitive semantics

Destroy the object denoted by expression `expr`.

well-typedness

```
expr : T, C ∈ C
```

semantics

```
t_{destroy(expr)}[u_x](σ, ε) = \{ (σ', ε) \}
```

where `σ' = σ|_{dom(σ) \{u\}}` with `u = I[expr](σ, u_x)`.

observables

```
Obs_{destroy(expr)}[u_x] = \{ (+, u) \}
```

(error) conditions

```
I[expr](σ, u_x) not defined.
```

---

Destroy Transformer Example

\[ S_{MC} : \]

```
s_1 \quad /\ delete n \quad s_2
```

\[ \text{destroy}(expr) \]

```
t_{destroy(expr)}[u_x](σ, ε) = ...
```

\[ \sigma : \]

```
: D \quad n \quad : C
```

\[ \varepsilon : \]

```
\text{cloud}
```

```
... 
```

\[ \sigma' : \]

```
\text{cloud}
```

```
\text{cloud}
```
The Full Story

UML distinguishes the following kinds of states:

- **simple state**
  - entry/act<sub>1</sub>
  - do/act<sub>2</sub>
  - exit/act<sub>3</sub>
  -...
  - E<sub>n</sub>/act<sub>E</sub>

- **final state**

- **composite state**
  - OR
  - AND

- **pseudo-state**
  - initial
  - (shallow) history
  - deep history
  - fork/join
  - junction, choice
  - entry point
  - exit point
  - terminate

- **submachine state**

\[ S : s \]
Plan:

**States / Syntax:**
- What is the abstract syntax of a diagram?

**States / Semantics:**
- what is the type of the implicit st attribute?
- what are legal system configurations?

**Transitions / Syntax:**
- what are legal / well-formed transitions?

**Transitions / Semantics:**
- when is a legal transition enabled?
- which effects do transitions have?

For example: From $s_1, s_5$, 
- what may happen on $E$?
- what may happen on $E, F$?
- can $E, G$ kill the object?
- ...
So far:

$$(S, s_0, \rightarrow), \quad s_0 \in S, \quad \rightarrow \subseteq S \times (\mathcal{D} \cup \{\_\}) \times \text{Expr}_\mathcal{D} \times \text{Act}_\mathcal{D} \times S$$

From now on: (hierarchical) state machines

$$(S, \text{kind}, \text{region}, \rightarrow, \psi, \text{annot})$$

where

- $S \supseteq \{\text{top}\}$ is a finite set of states \hspace{1cm} (new: \textit{top}).
- $\text{kind} : S \rightarrow \{\text{st, init, fin, shist, dhist, fork, join, junc, choi, ent, exi, term}\}$ is a function which labels states with their \textit{kind}. \hspace{1cm} (new)
- $\text{region} : S \rightarrow 2^2S$ is a function which characterises the \textit{regions} of a state. \hspace{1cm} (new)
- $\rightarrow$ is a set of transitions. \hspace{1cm} (changed)
- $\psi : (\rightarrow) \rightarrow 2^S \times 2^S$ is an \textit{incidence function}, and \hspace{1cm} (new)
- $\text{annot} : (\rightarrow) \rightarrow (\mathcal{D} \cup \{\_\}) \times \text{Expr}_\mathcal{D} \times \text{Act}_\mathcal{D}$ provides an annotation for each transition. \hspace{1cm} (new)

($s_0$ is then redundant – replaced by proper state (!) of kind 'init.')
Well-Formedness: Regions

<table>
<thead>
<tr>
<th>∈ S</th>
<th>kind</th>
<th>region ⊆ 2^S</th>
<th>child ⊆ S</th>
</tr>
</thead>
<tbody>
<tr>
<td>final state</td>
<td>s</td>
<td>fin</td>
<td>∅</td>
</tr>
<tr>
<td>pseudo-state</td>
<td>s</td>
<td>init, ...</td>
<td>∅</td>
</tr>
<tr>
<td>simple state</td>
<td>s</td>
<td>st</td>
<td>∅</td>
</tr>
<tr>
<td>composite state</td>
<td>s</td>
<td>st</td>
<td>{S_1, ..., S_n}, n ≥ 1</td>
</tr>
<tr>
<td>implicit top state</td>
<td>top</td>
<td>st</td>
<td>{S_1}</td>
</tr>
</tbody>
</table>

• Final and pseudo states must not comprise regions.
• States s ∈ S with kind(s) = st may comprise regions.

Naming conventions can be defined based on regions:
- No region: simple state.
- One region: OR-state.
- Two or more regions: AND-state.
- Each state (except for top) must lie in exactly one region.

Note: The region function induces a child function.
Note: Diagramming tools (like Rhapsody) can ensure well-formedness.

From UML to Hierarchical State Machine: By Example

(S, kind, region, →, ψ, annot)

<table>
<thead>
<tr>
<th>example</th>
<th>∈ S</th>
<th>kind</th>
<th>region</th>
</tr>
</thead>
<tbody>
<tr>
<td>simple state</td>
<td>s</td>
<td>s ∈</td>
<td>∅</td>
</tr>
<tr>
<td>final state</td>
<td>q</td>
<td>q ∈ fin</td>
<td>∅</td>
</tr>
<tr>
<td>composite state</td>
<td>s</td>
<td>s ∈</td>
<td>{S_1, S_2, S_3}</td>
</tr>
<tr>
<td>OR</td>
<td>s</td>
<td>s ∈</td>
<td>{S_1, S_2, S_3}</td>
</tr>
<tr>
<td>AND</td>
<td>s</td>
<td>s ∈</td>
<td>{S_1, S_2, S_3}, {S_4, S_5}, {S_6, S_7}, {S_8}</td>
</tr>
<tr>
<td>submachine state</td>
<td>(later)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>pseudo-state</td>
<td>q</td>
<td>q ∈</td>
<td>(s, kind(s)) for short</td>
</tr>
</tbody>
</table>
... denotes \((S, \text{kind}, \text{region}, \rightarrow, \psi, \text{annot})\) with

- \(S = \{\text{top}, s_1, s, s_2\}\)
- \(\text{kind} = \{\text{top} \mapsto \text{st}, s_1 \mapsto \text{init}, s \mapsto \text{st}, s_2 \mapsto \text{fin}\}\)
- or \((S, \text{kind}) = \{(\text{top}, \text{st}), (s_1, \text{init}), (s, \text{st}), (s_2, \text{fin})\}\)
- \(\text{region} = \{\text{top} \mapsto \{s_1, s, s_2\}, s_1 \mapsto \emptyset, s \mapsto \emptyset, s_2 \mapsto \emptyset\}\)
- \(\rightarrow, \psi, \text{annot}: \text{in a minute.}\)

Recall

Plan:
- States / Syntax:
  - What is the abstract syntax of a diagram?
- States / Semantics:
  - what is the type of the implicit \(st\) attribute?
  - what are legal system configurations?
- Transitions / Syntax:
  - what are legal / well-formed transitions?
- Transitions / Semantics:
  - when is a legal transition enabled?
  - which effects do transitions have?

For example: From \(s_1, s_5\),
- what may happen on \(E\)?
- what may happen on \(E, F\)?
- can \(E, G\) kill the object?
- ...
• The type of (implicit attribute) \( st \) is from now on a set of states, i.e. \( D(S_{MC}) = 2^S \).
• A set \( S_1 \subseteq S \) is called (legal) state configuration if and only if
  - \( \text{top} \in S_1 \), and
  - for each region \( R \) of a state in \( S_1 \), exactly one (non pseudo-state) element of \( R \) is in \( S_1 \), i.e.

\[
\forall s \in S_1 \forall R \in \text{region}(s) \cdot \{ s \in R \mid \text{kind}(s) \in \{ \text{st}, \text{fin} \} \} \cap S_1 = 1.
\]

• Examples:

- \( S_1 = \{ s_0 \} \times \) \( S_2 = \{ s_1, \text{top} \} \times \) \( s_3 = \{ s_1, s_2, s_3, s_4 \} \times \) \( S_5 = \{ s_5, s_6, s_7, s_8, s_9 \} \times \) \( S_6 = \{ s_6, \text{top}, s_7, s_8 \} \checkmark \)
- \( S_5 = \{ s_1, s_2, s_3, s_4 \} \checkmark \) \( S_7 = \{ s_4, s_5, s_6, s_7 \} \times \)

**Recall**

**Plan:**

- **States** / Syntax:
  - What is the abstract syntax of a diagram? *[✓]*
- **States** / Semantics:
  - what is the type of the implicit \( st \) attribute?
  - what are legal system configurations?
- **Transitions** / Syntax:
  - what are legal / well-formed transitions?
- **Transitions** / Semantics:
  - when is a legal transition enabled?
  - which effects do transitions have?

For example: From \( s_1, s_5 \),
- what may happen on \( E \)?
- what may happen on \( E, F \)?
- can \( E, G \) kill the object?
- ...
For simplicity, we consider transitions with (possibly) multiple sources and targets, i.e.
\[ \psi : (\rightarrow) \rightarrow (2^S \setminus \emptyset) \times (2^S \setminus \emptyset) \]

For instance,

\[ s_1 \rightarrow s_2 \rightarrow s_3 \rightarrow s_4 \rightarrow s_5 \rightarrow s_6 \]

translates to

\[ (S, \text{kind}, \text{region}, \{t_1\}, \{t_1 \mapsto (\{s_2, s_3\}, \{s_5, s_6\})\}, \{t_1 \mapsto (\text{tr, gd, act})\}) \]

Naming convention: \( \psi(t) = (\text{source}(t), \text{target}(t)) \).

---

Orthogonal States

Two states \( s_1, s_2 \in S \) are called orthogonal, denoted \( s_1 \perp s_2 \), if and only if

- they “live” in different regions of one AND-state, i.e.

\[ \exists s, \text{region}(s) = \{S_1, \ldots, S_n\}, 1 \leq i \neq j \leq n : s_1 \in \text{child}(S_i) \land s_2 \in \text{child}(S_j), \]
A hierarchical state-machine \((S, \text{kind}, \text{region}, \rightarrow, \psi, \text{annot})\) is called **well-formed** if and only if for all transitions \(t \in \rightarrow\),

- source (and destination) states are pairwise orthogonal, i.e.
  - \(\forall s, s' \in \text{source}(t) \land \text{target}(t) \land s \perp s'\),

- the top state is neither source nor destination, i.e.
  - \(\text{top} \notin \text{source}(t) \cup \text{target}(t)\).

**Recall:** final states are not sources of transitions.

**Example:**

```
\begin{figure}
\centering
\includegraphics[width=\textwidth]{example}
\caption{Example of a hierarchical state-machine.}
\end{figure}
```
For the Create Action, we have two main choices:
- re-use identities ("nasty semantics").
- use fresh identities ("clean semantics", depends on history).
Similar for Destroy.

Hierarchical State Machines introduce Regions.
- Thereby, states can lie within states as children.
- The implicit variable $st$ becomes set-valued.

Transitions may now have
- multiple source states, multiple destination states,
- but need to adhere to well-formedness conditions.

Enabledness of a set (!) of transitions
is a bit tricky to define (→ scope, priority, maximality).

Steps are a proper generalisation of core state machines.
References
