Software Design, Modelling and Analysis in UML

Lecture 15: Hierarchical State Machines I

or: Core State Machines V

2015-01-08

Prof. Dr. Andreas Podelski, Dr. Bernd Westphal

Albert-Ludwigs-Universität Freiburg, Germany
Contents & Goals

Last Lecture:
- RTC-Rules: Discard, Dispatch, Commence, Step, RTC

This Lecture:
- Educational Objectives: Capabilities for following tasks/questions.
  - What does this State Machine mean? What happens if I inject this event?
  - Can you please model the following behaviour.
  - What is: initial state.
  - What does this hierarchical State Machine mean? What may happen if I inject this event?
  - What is: AND-State, OR-State, pseudo-state, entry/exit/do, final state, ...

- Content:
  - Transformer: Create and Destroy, Divergence
  - Putting It All Together
  - Hierarchical State Machines Syntax
Missing Transformers: Create and Destroy
### Transformer: Create

<table>
<thead>
<tr>
<th>abstract syntax</th>
<th>concrete syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>create($C, expr, v$)</td>
<td>$\text{expr} \cdot v \leftarrow \text{new} \cdot C$</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

\[
\text{expr} : \tau_D, \ v \in \text{atr}(D), \\
\text{atr}(C) = \{ (v_i : \tau_i, \text{expr}_i^0) \mid 1 \leq i \leq n \} 
\]

#### Semantics

...  

#### Observables

...  

#### (Error) Conditions

\[ I[\text{expr}_i^0](\sigma, u) \] not defined for some $i$.

So note: \[ x := (\text{new} \cdot C).x + (\text{new} \cdot C).y; \]

If needed: \[ \text{tmp}_1 := \text{new} \cdot C; \]
\[ \text{tmp}_2 := \text{new} \cdot C; \]
\[ x := \text{tmp}_1.x + \text{tmp}_2.y; \]

So note: \[ \text{new} \cdot \text{Circle}(0.5); \]

If needed: \[ \text{tmp} := \text{new} \cdot \text{Circle}; \]
\[ \text{tmp} \cdot \text{in}(0.5); \]
### Transformer: Create

<table>
<thead>
<tr>
<th>Abstract Syntax</th>
<th>Concrete Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>create(C, expr, v)</code></td>
<td><code>create(C, expr, v)</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

\[
expr : \tau_D, \ v \in \text{atr}(D), \\
\text{atr}(C) = \{ \langle v_i : \tau_i, expr_i^0 \rangle | 1 \leq i \leq n \}
\]

#### Semantics

...  

#### Observables

...  

#### (Error) Conditions

\[I[expr_i^0](\sigma, u_x) \text{ not defined for some } i.\]

- 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 such as order of evaluation (and thus creation).
- Also for simplicity: no parameters to construction (\(\sim\) parameters of constructor). Adding them is straightforward (but somewhat tedious).
Create Transformer Example

\[ S M_C : \]

\[ s_1 \xrightarrow{\text{n := new } C} s_2 \]

\[
\text{create}(C, expr, v) \\
\quad t_{\text{create}(C,expr,v)}[u_x](\sigma, \varepsilon) = \ldots
\]

\[
\begin{array}{c}
\sigma: \\
\hline
\text{d : D} \\
\text{n = } \emptyset
\end{array}
\]

\[
\begin{array}{c}
\epsilon: \\
\end{array}
\]

\[
\begin{array}{c}
\text{d', D} \\
\text{13 e : C} \\
\text{x = 0} \\
\text{y = 3}
\end{array}
\]

\[
\begin{array}{c}
\text{d : D} \\
\text{14 e : C} \\
\text{x' = 0} \\
\text{y' = 3}
\end{array}
\]

\[
\begin{array}{c}
\text{d : D} \\
\text{x = 0} \\
\text{y = 3}
\end{array}
\]
Create Transformer Example

$SM_C$:

\[
\begin{align*}
\text{create}(C, \text{expr}, v) \\
t_{\text{create}}(C, \text{expr}, v)[u_x](\sigma, \varepsilon) &= \ldots
\end{align*}
\]

$\sigma$:

\[
\begin{array}{c}
d : D \\
n = \emptyset
\end{array}
\]

$\varepsilon$:

$\in D(C) \setminus \text{dom } \sigma$

(by init. value $\varepsilon$ expression)

$\in D(x(y)) = D(\text{Int})$

(non-det. choice)
How To Choose New Identities?

- **Re-use**: choose any identity that is not alive now, i.e. not in \( \text{dom}(\sigma) \).
  - 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 \( \text{dom}(\sigma) \) and any predecessor in current run.
  - Depends on history.
  - Dangling references remain dangling – could mask “dirty” effects of platform.
### Transformer: Create

<table>
<thead>
<tr>
<th>abstract syntax</th>
<th>concrete syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>create($C, expr, v$)</td>
<td></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

$$expr : \tau_D, \; v \in \text{atr}(D), \quad \text{atr}(C) = \{ \langle v_i^1 : \tau_i^1, expr_0^i \rangle \mid 1 \leq i \leq n \}$$

#### Semantics

$$((\sigma, \varepsilon), (\sigma', \varepsilon')) \in t \iff \sigma' = \sigma[u_0 \mapsto \sigma(u_0)[v \mapsto u]] \cup \{ u \mapsto \{ v_i \mapsto d_i \mid 1 \leq i \leq n \} \}, \quad \varepsilon' = [u](\varepsilon); \quad u \in \mathcal{D}(C) \text{ fresh, i.e. } u \notin \text{dom}(\sigma);$$

$$u_0 = I[expr](\sigma, u_x); \quad d_i = I[expr_0^i](\sigma, u_x) \text{ if } expr_0^i \neq \star \quad \text{and } d_i \in \mathcal{D}(\tau_i) \text{ otherwise (non-determinism).}$$

#### Observables

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

#### (Error) Conditions

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

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

**intuitive semantics**

*Destroy the object denoted by expression* `expr`.

**well-typedness**

`expr : τ_C, C ∈ C`

**semantics**

\[
\text{observables}
\]

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

**(error) conditions**

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

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

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

\[ t_{\text{destroy}(expr)[u_x]}(\sigma, \varepsilon) = \ldots \]

\[ t_{\text{destroy}(expr)}[u_x](\sigma, \varepsilon) = \ldots \]

\[ u \text{ is gone} \]

\[ n \text{ is a dangling reference now} \]
What to Do With the Remaining Objects?

Assume object $u_0$ is destroyed by $v_3$.

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

- object $u_0$ may have been the last one linking to object $u_2$:
  - leave $u_2$ alone?
  - or remove $u_2$ also?

- 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.
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>-- expr --</code></td>
</tr>
</tbody>
</table>

**Intuitive Semantics**

`Destroy the object denoted by expression expr.`

**Well-Typedness**

`expr : τ, C ∈ C`

**Semantics**

\[ t[\text{ux}](\sigma, \varepsilon) = (\sigma', \varepsilon) \]

where \( \sigma' = \sigma|_{\text{dom}(\sigma)\setminus\{u\}} \) with \( u = I[expr](\sigma, \text{ux}) \).

**Observables**

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

**(Error) Conditions**

\( I[expr](\sigma, \text{ux}) \) not defined.
Step and Run-to-completion Step
**Note:** we call one evolution \((\sigma, \varepsilon) \xrightarrow{(cons, Snd)} u \xrightarrow{u} (\sigma', \varepsilon')\) a step.

Thus in our setting, a step **directly corresponds** to

one object (namely \(u\)) takes a **single transition** between regular states.

(We have to extend the concept of “single transition” for hierarchical state machines.)

**That is:** We’re going for an interleaving semantics without true parallelism.

**Remark:** With only methods (later), the notion of step is not so clear. For example, consider

- \(c_1\) calls \(f()\) at \(c_2\), which calls \(g()\) at \(c_1\) which in turn calls \(h()\) for \(c_2\).

- Is the completion of \(h()\) a step?
- Or the completion of \(f()\)?
- Or doesn’t it play a role?

It does play a role, because **constraints/invariants** are typically (＝ by convention) assumed to be evaluated at step boundaries, and sometimes the convention is meant to admit (temporary) violation in between steps.
Notions of Steps: The Run-to-Completion Step

What is a run-to-completion step...?

- **Intuition**: a maximal sequence of steps, where the first step is a dispatch step and all later steps are commence steps.
- **Note**: one step corresponds to one transition in the state machine.

A run-to-completion step is in general not syntactically definable — one transition may be taken multiple times during an RTC-step.

**Example:**

\[ E[x > 0]/ \]

\[ \sigma:\]
\[
\begin{array}{c}
: C \\
x = 2
\end{array}
\]

\[ \varepsilon:\]
\[ E \text{ for } u \]
**Proposal:** Let

\[
(\sigma_0, \varepsilon_0) \xrightarrow{(\text{cons}_0, \text{Snd}_0)} u_0 \ldots \xrightarrow{(\text{cons}_{n-1}, \text{Snd}_{n-1})} u_{n-1} \xrightarrow{(\sigma_n, \varepsilon_n), \ n > 0,}
\]

be a finite (!), non-empty, maximal, consecutive sequence such that

- object \( u \) is alive in \( \sigma_0 \),
- \( u_0 = u \) and \((\text{cons}_0, \text{Snd}_0)\) indicates dispatching to \( u \), i.e. \( \text{cons} = \{(u, \vec{v} \mapsto \vec{d})\} \),
- there are no receptions by \( u \) in between, i.e.

\[
\text{cons}_i \cap \{u\} \times \text{Evs}(\mathcal{E}, \mathcal{D}) = \emptyset, i > 1,
\]

- \( u_{n-1} = u \) and \( u \) is stable only in \( \sigma_0 \) and \( \sigma_n \), i.e.

\[
\sigma_0(u)(\text{stable}) = \sigma_n(u)(\text{stable}) = 1 \text{ and } \sigma_i(u)(\text{stable}) = 0 \text{ for } 0 < i < n,
\]

Let \( 0 = k_1 < k_2 < \cdots < k_N = n \) be the maximal sequence of indices such that \( u_{k_i} = u \) for \( 1 \leq i \leq N \). Then we call the sequence

\[
(\sigma_0(u) =) \sigma_{k_1}(u), \sigma_{k_2}(u) \ldots, \sigma_{k_N}(u) \quad (= \sigma_{n-1}(u))
\]

a (!) **run-to-completion computation** of \( u \) (from (local) configuration \( \sigma_0(u) \)).
Divergence

We say, object $u$ can diverge on reception $cons$ from (local) configuration $\sigma_0(u)$ if and only if there is an infinite, consecutive sequence

$$(\sigma_0, \varepsilon_0) \xrightarrow{(cons_0, Snd_0)} (\sigma_1, \varepsilon_1) \xrightarrow{(cons_1, Snd_1)} \ldots$$

such that $u$ doesn’t become stable again.

- **Note**: disappearance of object not considered in the definitions.
  By the current definitions, it’s neither divergence nor an RTC-step.
Run-to-Completion Step: Discussion.

What people may dislike on our definition of RTC-step is that it takes a global and non-compositional view. That is:

- In the projection onto a single object we still see the effect of interaction with other objects.
- Adding classes (or even objects) may change the divergence behaviour of existing ones.
- Compositional would be: the behaviour of a set of objects is determined by the behaviour of each object “in isolation”.

Our semantics and notion of RTC-step doesn’t have this (often desired) property.

Can we give (syntactical) criteria such that any global run-to-completion step is an interleaving of local ones?

Maybe: Strict interfaces. (Proof left as exercise...)

- (A): Refer to private features only via “self”.
  (Recall that other objects of the same class can modify private attributes.)
- (B): Let objects only communicate by events, i.e. don’t let them modify each other’s local state via links at all.
References


