Software Design, Modelling and Analysis in UML

Lecture 15: Hierarchical State Machines I

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
### 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>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 : τ : D`, `v : atr(D)`
- `atr(C) = \{ (v_i : τ_i, expr_{i}^0) | 1 \leq i \leq n \}`

**Semantics**

... (omitted)

**Observables**

... (omitted)

**Error Conditions**

`I[expr_{i}^0](σ, u_x)` not defined for some `i`.

---

**Example**

**So Not:**

```
x = (new C).x + (new C).y;
```

**If Needed:**

```
tmp_1 = new C;
tmp_2 = new C;
x = tmp_1.x + tmp_2.y;
```

---

**So Not:**

```
new C(0.5);
```

**If Needed:**

```
tmp = new C(0.5);
tmp."
```
Transformer: Create

<table>
<thead>
<tr>
<th>abstract syntax</th>
<th>concrete syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{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**

\[
\begin{align*}
\text{expr} : \tau_D, v \in \text{atr}(D), \\
\text{atr}(C) = \{ \langle v_i : \tau_i, \text{expr}_i \rangle \mid 1 \leq i \leq n \}
\end{align*}
\]

**semantics**

\[
\ldots
\]

**observables**

\[
\ldots
\]

**(error) conditions**

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

- 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 (~ parameters of constructor). Adding them is straightforward (but somewhat tedious).

---

Create Transformer Example

\[
S_M C:
\]

\[
\begin{array}{c}
s_1 \quad \text{\( n := \text{new } C \) } \quad s_2
\end{array}
\]

\[
\text{create}(C, \text{expr}, v)
\]

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

\[
\sigma:
\]

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

\[
\varepsilon:
\]

\[
\begin{array}{c}
\text{\( x = 0 \)} \\
\text{\( y = 3 \)}
\end{array}
\]

\[
\begin{array}{c}
\text{\( x = 0 \)} \\
\text{\( y = 3 \)}
\end{array}
\]

\[
\begin{array}{c}
\text{\( x = 0 \)} \\
\text{\( y = 3 \)}
\end{array}
\]
Create Transformer Example

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

\[
\text{create}(C, \text{expr}, v) \\
\text{t}_{\text{create}}(C, \text{expr}, v)[\sigma][\varepsilon] = \ldots
\]

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

\[
\varepsilon: \\
\begin{array}{c}
\exists D(x) \\setminus \text{dom } \sigma \\
\\text{by init. value expression}
\end{array}
\]

---

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**

### Abstract Syntax

\[ \text{create}(C, \text{expr}, v) \]

### Concrete Syntax

\[ \text{create}(C, \text{expr}, v) \]

### Intuitive Semantics

Create an object of class \( C \) and assign it to attribute \( v \) of the object denoted by expression \( \text{expr} \).

### Well-Typedness

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

### Semantics

\[ (\sigma, \varepsilon), (\sigma', \varepsilon') \in I \]

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

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

\[ u_0 = I[\text{expr}](\sigma, u_x); \quad d_i = I[\text{expr}_i](\sigma, u_x) \text{ if } \text{expr}_i \neq \epsilon \]

and \( d_i \in \mathcal{D}(\tau_i) \) otherwise (non-determinism).

### Observables

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

### (Error) Conditions

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

**Transformer: Destroy**

### Abstract Syntax

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

### Concrete Syntax

\[ \text{delete expr} \]

### Intuitive Semantics

Destroy the object denoted by expression \( \text{expr} \).

### Well-Typedness

\[ \text{expr} : \tau_C, \quad C \in \mathcal{G} \]

### Semantics

\[ \ldots \]

### Observables

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

### (Error) Conditions

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

**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>destroy(expr)</code></td>
</tr>
</tbody>
</table>

#### Intuitive Semantics

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

#### Well-typedness

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

#### Semantics

\[ t[u_x](\sigma, \varepsilon) = (\sigma', \varepsilon) \]

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

#### Observables

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

#### Error Conditions

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

---

**Step and Run-to-completion Step**
Notions of Steps: The Step

**Note:** we call one evolution \((\sigma, \varepsilon) \xrightarrow{\text{cons, } Snd_u} (\sigma', \varepsilon')\) 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:**

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

\[
\varepsilon:\begin{array}{c}
\begin{array}{c}
\text{\{private\}}
\end{array}
\end{array}
\begin{array}{c}
\begin{array}{c}
E\text{ for } u
\end{array}
\end{array}
\]

\[
E[x > 0]/\quad \exists x: 0 < x = x - 1
\]
Notions of Steps: The RTC Step Cont’d

Proposal: Let

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

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. \(\text{cons}_0 = \{(u, \vec{v} \mapsto \vec{d})\}\),
- there are no receptions by \(u\) in between, i.e.
  \[\text{cons}_i \cap \{u\} \times \text{Ev}(\mathcal{E}, \mathcal{D}) = \emptyset, \quad i > 1,\]
- \(u_{n-1} = u\) and \(u\) is stable only in \(\sigma_0\) and \(\sigma_n\), i.e.
  \(\sigma_0(u)(\text{stable}) = \sigma_n(u)(\text{stable}) = 1\) and \(\sigma_i(u)(\text{stable}) = 0\) for \(0 < i < n\),

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

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

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

Divergence

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

\[(\sigma_0, \varepsilon_0) \xrightarrow{\text{cons}_0, \text{Snd}_0} (\sigma_1, \varepsilon_1) \xrightarrow{\text{cons}_1, \text{Snd}_1} \ldots\]

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


