Contents & Goals

Last Lecture:
• Legal state configurations
• Legal transitions
• Rules (i) to (v) for hierarchical state machines

This Lecture:
• Educational Objectives:
  • How do entry / exit actions work? What about do-actions?
  • What is the effect of shallow / deep history pseudo-states?
  • What about junction, choice, terminate, etc.?
  • What is the idea of deferred events?
  • How are passive reactive objects treated in Rhapsody's UML semantics?
  • What about methods?

• Content:
  • Entry / exit / do actions, internal transitions
  • Remaining pseudo-states; deferred events
  • Passive reactive objects
  • Behavioural features

Entry/Do/Exit Actions

In general, with each state $s \in S$ there is associated
• an entry, a do, and an exit action (default: skip)
• a possibly empty set of trigger/action pairs called internal transitions (default: empty).

Note: 'entry', 'do', 'exit' are reserved names; $E_1, \ldots, E_n \in E$.

Recall: each action is supposed to have a transformer; assume $t_{act\ entry_1}, t_{act\ exit_1}, \ldots$.

Taking the transition above then amounts to applying $t_{act\ entry_2} \circ t_{act\ exit_1}$ instead of just $t_{act} \Rightarrow$ adjust Rules (ii), (iii), and (v) accordingly.

Internal Transitions

Taking an internal transition, e.g. on $E_1$, only executes $t_{act\ E_1}$.

Intuition: The state is neither left nor entered, so: no exit, no entry action.

Note: internal transitions also start a run-to-completion step.

Note: the standard seems not to clarify whether internal transitions have priority over regular transitions with the same trigger at the same state. Some code generators assume that internal transitions have priority!

Alternative View: Entry / Exit / Internal as Abbreviations

Can be viewed as abbreviation for . . .

• That is: Entry / Internal / Exit don't add expressive power to Core State Machines.
  • If internal actions should have priority, $s_1$ can be embedded into an OR-state.
  • Abbreviation view may avoid confusion in context of hierarchical states.
• Intuition: after entering a state, start its do-action.
  • If the do-action terminates, then the state is considered completed (like final state), otherwise, if the state is left before termination, the do-action is stopped.
  • Recall the overall UML State Machine philosophy: "An object is either idle or doing a run-to-completion step."
  • Now, what is it exactly while the do action is executing...?

The Concept of History, and Other Pseudo-States

History and Deep History: By Example

Junction and Choice

Hierarchical states can be "folded" for readability. (but: this can also hinder readability.) Can even be taken from a different state-machine for re-use.
Hierarchical states can be "folded" for readability. (but: this can also hinder readability.)

Can even be taken from a different state-machine for re-use.

Entry/exit points, provide connection points for finer integration into the current level, than just via initial state.

Semantically a bit tricky:

First the exit action of the exiting state,
then the actions of the transition,
then the entry actions of the entered state,
then action of the transition from the entry point to an internal state,
and then that internal state's entry action.

Terminate Pseudo-State

When a terminate pseudo-state is reached, the object taking the transition is immediately killed.

UML distinguishes the following kinds of states:

Example

Simple state

Entry/act

Do/act

Exit/act

E...E

Final state

Composite state

OR

S 1  S 2  S 3

AND

S 1  S 2  S 3  S' 1  S' 2  S' 3

Deferred Events in State-Machines

Deferred Events: Idea

The idea is as follows:

Consider the following state machine:

Assume we're stable in S 1, and F is ready in the ether.

In the framework of our course, F is discarded.

But we may find it a pity to discard the poor event and we may want to remember it for later processing, e.g. in S 2, in other words: defer it.

General options to satisfy such needs:

Provide a pattern how to "program" this (use self-loops and helper attributes).

Turn it into an original language concept.

(← OMG's choice)

Deferred Events: Syntax and Semantics

Syntactically, each state has (in addition to the name) a set of deferred events.

Default: the empty set.

The semantics is a bit intricate, something like:

if Rule (i) (discard) would apply, but E is in the deferred set of the current state configuration, then stuff E into some "deferred events space" of the object, (e.g. into the ether (extend ε) or into the local state of the object (extend σ)) and turn attention to the next event.

Not so obvious:

Is there a priority between deferred and regular events?

Is the order of deferred events preserved?

...
What about non-active objects?

Recall:

• We’re still working under the assumption that all classes in the class diagram (and thus all objects) are active.
• That is, each object has its own thread of control and is (if stable) at any time ready to process an event from the ether.

→ Steps of active objects can interleave.

But the world doesn’t consist of only active objects. For instance, in the crossing controller from the exercises we could wish to have the whole system live in one thread of control. So we have to address questions like:

• Can we send events to a non-active object?
• And if so, when are these events processed?
• etc.

Passive and Reactive / Rhapsody Style: Nomenclature

Harel and Gery (1997) propose the following (orthogonal!):

• A class (and thus the instances of this class) is either active or passive as defined by the class diagram.
• An active object has (in the operating system sense) an own thread: an own program counter, an own stack, etc.
• A passive object doesn’t.
• A class is either reactive or non-reactive.
• A reactive class has a (non-trivial) state machine.
• A non-reactive one hasn’t.

Which combinations do we (not) understand yet?

<table>
<thead>
<tr>
<th>Active</th>
<th>Passive</th>
<th>Reactive</th>
<th>Non-Reactive</th>
</tr>
</thead>
<tbody>
<tr>
<td>✔</td>
<td>✔</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>

WANTED:

• In each class, add (implicit) link itsAct and use it to make each object u know the active object ua which is responsible for dispatching events to u.
• If u is an instance of an active class, then ua = u.
• Equip all signals with (implicit) association dest and use it to point to the destination object. For each signal F, have a version F\textsubscript{C} with an association dest: C\textsubscript{0}, 1, C \in C (no inheritance yet).
And What About Methods?

In UML, the former is called "feature" and can (roughly) be implemented by actions of transitions, which we abstract to transformers. In a setting with Java as action language: operation is a method body.

In our setting, we simply assume a transformer like a "triggered operation". (i.e. there are pictures that mean the same thing to all three communities)

Observation: the ether only has events for active objects.

Say an event is ready in the ether for a certain object, and use it to point to the destination object. That's a connection. In each class, add (implicit) link to the "owner" of the class.

Equip all signals with (implicit) association. This is passive reactive (narrowly).

And Where About Declared?

Exercise: Search the standard for "semantical variation point". They are legion...

- tools exist with complete and consistent code generation.
- true concurrency is enforced determinism.
- allow etc. etc.

Rhapsody state machines — the bottom line is: none is the subset of another — and waits until completion of corresponding RTC.

-ilicit a semantic difference.

Discussion: tools exist with partial syntactical variation point. Search the central database and set the database to contain both version.

In general, there are also methods.

Parameter declarations is an action of a transition.

And What About Methods?
Visibility:

- Extend typing rules to sequences of actions such that a well-typed action sequence only calls visible methods.

Useful properties:

- concurrency
- concurrent — is thread safe
- guarded — some mechanism ensures/should ensure mutual exclusion
- sequential — is not thread safe, users have to ensure mutual exclusion
- isQuery — doesn’t modify the state space (thus thread safe)

For simplicity, we leave the notion of steps untouched, we construct our semantics around state machines. Yet we could explain pre/post in OCL (if we wanted to).

References


