Contents & Goals

Last Lecture:
• Symbolic Büchi Automata (TBA) and its (accepted) language.
• Words of a model.

This Lecture:
• Educational Objectives:
  Capabilities for following tasks/questions.
• What does this LSC mean?
• Are these UML model’s state machines consistent with the interactions?
• Please provide a UML model which is consistent with this LSC.
• What is: activation, hot/cold condition, pre-chart, etc.?

Content:
• LSC abstract syntax.
• LSC formal semantics.

Live Sequence Charts Abstract Syntax

Example

LSC:

AC:

AM: invariant

Environment:

Operational:

1. Let Θ = \{hot, cold\}. An LSC body is a tuple (I, (C, ⪯), ∼, CB, Msg, Cond, LocInv).

   • I is a finite set of instance lines,
   • C is a set of conditions,
   • ⪯ is a partial order on C,
   • ∼ is an equivalence relation on C,
   • CB is a set of control blocks,
   • Msg is a set of messages,
   • Cond is a set of conditions,
   • LocInv is a set of location invariants.

Example
Then, and

\[ l = \sim l \land l \sim \sim \]z.

Simultaneity in particular taking activation condition and activation mode into construct a TBA

Examples: Semantics.

Note: if messages in a chart are

\[ \exists l \in \forall l \text{Msg} \in l \]z. or

\[ l \in \{ l \in l \text{Msg} \in l \} \]z. i.e.

\[ \sim l \sim l \]z., or

\[ l \in \{ l \in l \} \]z.

Well-formedness
\[
\begin{align*}
&\text{Intuition} \\
&\text{The set of legs exits legally exit} \Rightarrow \text{what allows us to stay at cut} \\
&\text{loops consist of} \\bullet\text{instance heads}\quad \text{corresponding to} \quad \text{corresponding to} \\
&\text{corresponding to} \quad \text{corresponding to} \\
&\text{legs} \\
&\text{loops} \\
&\text{legs}
\end{align*}
\]
Formally let $q$ be the fired set $\text{(unique)}$.  

Progress with conditions.

Step II: Conditions and Local Invariants

Even More Helper Functions

Intuition

• When do we move from $Snd$ to $\text{cons}$ in $\text{weak mode}$?

• Even messages are distinguished messages.

• Formally and let $q$ be the fired set of $\text{(unique)}$.

• Moreover $q \ni \text{messageCut} \in$ in $\text{weak mode}$.  

• And nothing else.
Figure 14.31 - Timing Diagram with more than one Lifeline and with Messages

Figure 14.27 - Communication diagram

Figure 14.30 - Compact Lifeline with States

Step 5: Cold Conditions and Cold Local Invariants

Back to UML: Interactions

• An interaction can be (OMG claim: equivalently) a sequence of messages, or a timing diagram (reflective description of behavior is consistent with the concrete description).

• An interaction has a set of interactions, e.g.,

  – HasW/k
  – LocInv
  – Cond
  – ⪯

• When do we take a legal exit from the statechart? The exit configuration is a pair of an endstate and an exit event.

Back to UML: Interactions

Back to UML: Interactions
The Concept of History, and Other Pseudo-States

Can be formalised with protocol state machines but before shall only be called after instance, method call orders same direction:

- History and Deep History: By Example
  - Side Note: Protocol State Machines

- Time Observation
- Duration Observation
- Forbidden Scenarios
- Most Severe of these formalisms: Live Sequence Charts, standardized by the ITU in different versions, often accused to lack a formal semantics. Most Prominent of UML1.x: Message Sequence Charts. Why Sequence Diagrams?

- Activation
- Interpretation
- Drawbacks

- Live Sequence Diagrams

- Environment
- Message
- Observer
- Subject
- Acceptable
- No Card
- Idle
- Open Door
- Idle
- Use
- Wait Alarm
- Wait Access
- Message Sequence Diagrams
- A CS ystem
- Observer.range = (0 .. Subject.capacity)
- Observer.range = (0 .. Subject.capacity)
- Message Sequence Diagrams
- Observer
- Open Door
- Observer: SlidingBarIcon
- Observer : SlidingBarIcon
- Observer
- Observer
- Observer

- Interaction Constraint
- Time Observation
• Junction ("static conditional branch"): 
  \[
  \begin{array}{c}
  \text{gd}_1 / \text{act}_1 \\
  \text{gd}_2 / \text{act}_2
  \end{array}
  \]

• Choice ("dynamic conditional branch"): 

Note: not so sure about naming and symbols, e.g., I'd guessed it was just the other way round...

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.
DeferredEvents: Idea

UML state machines contain the feature of deferred events. The idea is as follows:

- Consider the following state machine:

  \[ s_1 \rightarrow E/F \rightarrow s_2 \rightarrow s_3 \]

- Assume we're stable in state \( s_1 \), and \( F \) is ready in the ether.

- In the framework of the course, \( F \) is discarded.

- But, we may find it a pity to discard the poor event and may want to remember it for later processing, e.g., in state \( s_2 \), in other words, defer it.

General options to satisfy such needs:

- Provide a pattern how to "program" this (uses 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 an event $E$ is dispatched,
  - And there is no transition enabled to consume $E$,
  - And $E$ 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 ($\varepsilon = \text{extend}$) or into the local state of the object ($\sigma = \text{extend}$))
  - 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?

[Fecher and Schönborn, 2007], e.g., claim to provide semantics for the complete Hierarchical State Machine language, including deferred events.

---

Active and Passive Objects

**[Hareland and Gery, 1997]**

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.

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.

---

Active and Passive Objects: Nomenclature

[Hareland and Gery, 1997] propose the following (orthogonal!) notions:
- A class (and thus the instances of this class) is either active or passive as declared in the class diagram.
- An active object has (in the operating systems sense) an own thread: an own program counter, an own stack, etc.
- A passive object doesn't.
Active and Passive Objects: Nomenclature

[Hareland Gery, 1997] propose the following (orthogonal!) notions:

• A class (and thus the instances of this class) is either active or passive as declared in the class diagram.

• An active object has (in the operating systems 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 understand?

- active
- passive
- reactive ✔ (∗)
- non-reactive (✔)

Passive and Reactive

• So why don’t we understand passive/reactive?

• Assume passive objects \( u_1 \) and \( u_2 \), and active object \( u \), and that there are events in the ether for all three.

• Which of them (can) start a run-to-completion step…?

• Do run-to-completion steps still interleave…?

Reasonable Approaches:

• Avoid — for instance, by

• requiring that reactive implies active for model well-formedness.

• requiring for model well-formedness that events are never sent to instances of non-reactive classes.

• Explain — here: (following [Hareland Gery, 1997])

• Delegate all dispatching of events to the active objects.

Passive and Reactive Classes

• Firstly, establish that each object \( u \) knows, via (implicit) link \( \text{itsAct} \), the active object \( u_{\text{act}} \) which is responsible for dispatching events to \( u \).

• If \( u \) is an instance of an active class, then \( u_a = u \).

• \( C_1 \rightarrow C_2 \) \( \langle \langle \text{signal} \rangle \rangle \) \( E \rightarrow C_1 \) \( \langle \langle \text{signal} \rangle \rangle \) \( E \rightarrow C_2 \) \( \langle \langle \text{signal} \rangle \rangle \) \( E \rightarrow D_{\text{dest}} \) \( C_{\text{dest}} \).
In the current setting, the (local) state of objects is only modified by methods that are directly connected to the state-transition diagram. This approach is consistent with UML's emphasis on using methods to represent behavioral features.
• The implementation of a behavioural feature can be provided by:
  • An operation. In our setting, we simply assume a transformer like $T_f$.
    It is then, e.g., clear how to admit method calls as actions on transitions:
    function composition of transformers (clear but tedious: non-termination).
    In a setting with Java as action language: operation is a method body.
  • The class' state-machine ("triggered operation").
  • Calling $F$ with $n_2$ parameters for a stable instance of $C$ creates an auxiliary event $F$ and dispatches it (bypassing the ether).
  • Transition actions may fill in the return value.
  • On completion of the RTC step, the call returns.
  • For a non-stable instance, the caller blocks until stability is reached again.

Visibility:

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

Useful properties:

• Concurrency — is thread-safe
• Concurrent — guarded — some mechanism ensures/should ensure mutual exclusion
• Sequential — is not threadsafe, 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


