The RHAPSODY Semantics of Statecharts (aka On the Executable Core of the UML)

Raphael Mannadiar

March 20, 2008

References

Harel D. and Kugler H.
The rhapsody semantics of statecharts (or, on the executable core of the uml), 2004.

Intro

- In 1996, the RHAPSODY tool provided one of the first executable semantics for OO statecharts
- RHAPSODY did not introduce OO statecharts but provide one of the only fully specified implementations
- To accommodate OO behaviour, the RHAPSODY semantics differs here and there from that of STATEMATE

Outline

The Basics

Basic System Reaction

Compound Transitions

History

Transition Scope

Conflicting Transitions

Basic Step Algorithm

Multi-Threaded Systems

Outline

The Basics

Basic System Reaction

Compound Transitions

History

Transition Scope

Conflicting Transitions

Basic Step Algorithm

Multi-Threaded Systems

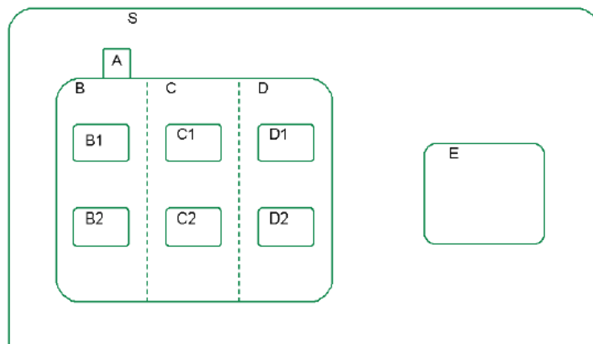
Classes

- An OO system is composed of classes
- A statechart describes how a class reacts to messages it receives by defining the actions taken and the new mode entered.
- A class can but need not have a statechart (e.g. data-driven class)
- A new independently running statechart is created of each instance of a class

States

As in STATEMATE, there are 3 types of states:

- OR-states: substates are related to each other by XOR
- AND-states: have orthogonal components related by AND
- basic states: have no substates



Transitions

The general syntax of an expression labelling a transition is $m[c]/a$ where

- m : the message that triggers the transition
- c : is the guard
- a : an action carried out if and when the transition is taken

Labels are optional.

Transitions

The general syntax of an expression labelling a transition is *m[c]/a* where

- *m* : the message that triggers the transition
- *c* : is the guard
- *a* : an action carried out if and when the transition is taken

Labels are optional.

In RHAPSODY,

- the message/trigger can be a timeout, an event or a trigger (more later...)
- triggerless transitions are allowed : they're called *null transitions*
- guards and actions are written in the implementation language (i.e. C/C++, Java, Ada)
- the action can be an event or trigger generation, a primitive operation (i.e. method call in underlying language) or a sequence of actions

Actions

Actions can be associated to states as well as transitions...

- Entry actions are carried out during the step in which state S is entered...same as if on incoming transition
- Exit actions are carried out during the step in which state S is exited...same as if on outgoing transition

There are also special actions called *static reactions*. They have the same syntax ($m[c]/a$) as transition labels but don't entail a state transition.

Steps

- The execution of a statechart consists of a series of *steps*
- The execution of a step consists of a series of microsteps (more later)
- Microsteps can take non-zero time to complete (e.g. some action in the implementation language). As a result, a transition from $S1$ to $S2$ can take non-zero time.
- Changes that occur in a step can be sensed in the same step
- A step always leads from one legal state configuration to another

Outline

The Basics

Basic System Reaction

Compound Transitions

History

Transition Scope

Conflicting Transitions

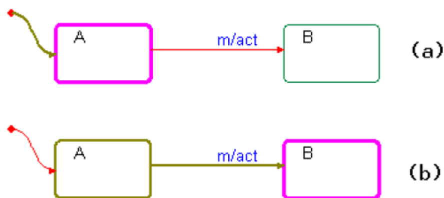
Basic Step Algorithm

Multi-Threaded Systems

A simple step

"Statecharts can react to messages by performing a transition from one active configuration to another and possibly performing an action"

The object is in state A and message m is dispatched.



1. Exit action of state A is performed
2. *act* is performed
3. Entry action of state B is performed
4. The object is placed in state B

Events

Statecharts can communicate asynchronously via *events*.

- Each class defines the set of events it can receive.
- Events are nice because the sender object can continue its work without waiting for the receiver object to consume the event

Events

Statecharts can communicate asynchronously via *events*.

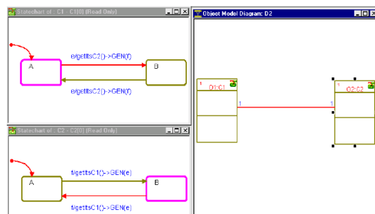
- Each class defines the set of events it can receive.
- Events are nice because the sender object can continue its work without waiting for the receiver object to consume the event

Events are sent by applying the *GEN* method to the destination object:
$O \rightarrow GEN(event(p_1, p_2, ..., p_N))$ where

- O is some reference to the destination object obtained by navigating the model relations
- $(p_1, p_2, ..., p_N)$ are event parameters

The GEN method creates events and queues them in the event queue of O's thread (multi-threading later)

Events...

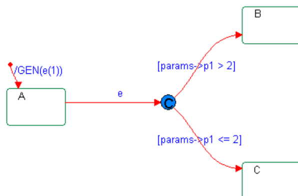


1. Object $O1$ receives event e
2. The transition from state A to state B is taken involving sending event f to $O2$
3. Once the transition of $O1$ to state B is complete, f is removed from the event queue and dispatched to $O2$
4. $O2$ the transits from state A to state B...
5. ...
6. The system ends up looping forever

Events...

Events have two more interesting properties

- they can be internal, in which case the destination object is omitted.
- they are independent entities of the model and can be sub-classed like objects.



- Internal event *e* is sent when the statechart performs the default transition to state *A*
- This triggers the transition from *A* to the condition connector...
- BUT, if event *e2(1)* derived from *e* had occurred, the same behaviour would have occurred... an event triggers all the transitions of its super-events

Triggered Operations

Statecharts can also communicate synchronously via *triggered operations*.

Triggered operations

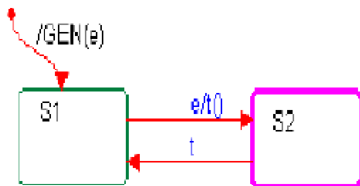
- are services provided by a class i.e. callable class methods
- are a means of synchronous communication between a client and a server object
- the operation can return a value to the client

Triggered operations were integrated into RHAPSODY to model systems that are not event-driven.

Triggered Operations...

- A triggered operation is invoked like a primitive operation in the underlying implementation language: $result = O \rightarrow to(p_1, p_2, ..., p_N)$
- The return value of a triggered operation is set within the transition. It is specified via the reply method: $t/reply(ret_val)$

One concern is the reaction to an object to an invocation of a triggered operation when it is the midst of performing a transition



Event *e* is generated while transiting to *S1*, this entails a call to *t()* on this statechart...

Triggered Operations...

Possible alternatives to deal with this situation are

- Design problem → Deadlock
- Complete transition to *S2* then process *t()*
- Ignore the invocation → `RHAPSODY` choice

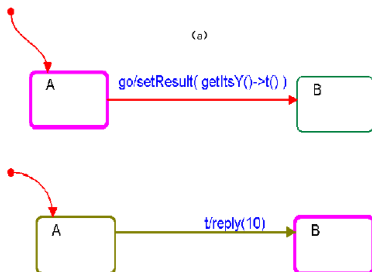
Triggered Operations...

Possible alternatives to deal with this situation are

- Design problem → Deadlock
- Complete transition to *S2* then process *t()*
- Ignore the invocation → RHAPSODY choice

You may remember the earlier example that showed and infinite loop, replacing the events by triggered operations would break that loop.

Triggered Operations...



1. Object X receives event *go*
2. The transition from state A to state B is taken involving a call to *Y.t()*
3. X does not transit to state B until t is processed causing object Y to transit from state A to state B and to return 10.
4. The X' transition is completed (i.e. X moves to state B) after *setResult()* is called with parameter value 10 and returns.

Outline

The Basics

Basic System Reaction

Compound Transitions

History

Transition Scope

Conflicting Transitions

Basic Step Algorithm

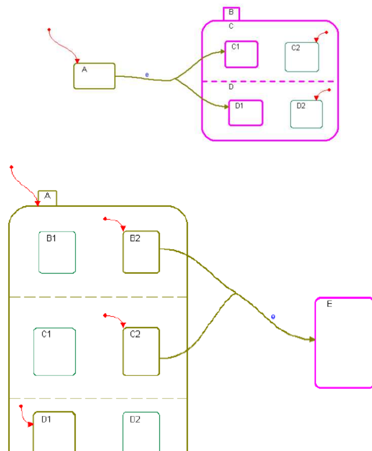
Multi-Threaded Systems

Compound Transitions

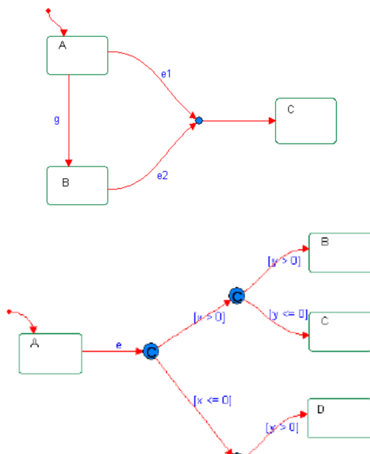
You might recall from your experience with AToM3 that statecharts admit hyperedges...
In the RHAPSODY semantics, hyperedges are called connectors and come in 2 types:

- AND connectors: e.g. forks and joins → all transition segments participate in the transition
- OR connectors: e.g. junctions and conditions → one input and one output transition segment participate in the transition

AND connectors



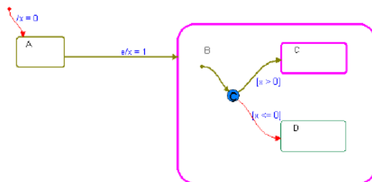
OR connectors



Defaults

"A statechart can't be in a non-basic state without the ability to enter appropriate substates"

- To satisfy this property, OR states with more than one substate must specify a *default connector*
- Taking a default transition is considered to be a microstep.
- Recall that actions taken in previous microsteps are taken into account at each microstep...



Hence, in the above example, x takes value 1 before the default transition from B is taken and the statecharts ends up in state C

Outline

The Basics

Basic System Reaction

Compound Transitions

History

Transition Scope

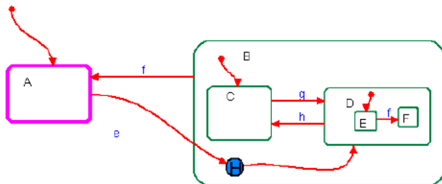
Conflicting Transitions

Basic Step Algorithm

Multi-Threaded Systems

History

- A history connector is used to store the most recent active configuration of a state
- The semantics of this connector is that when it is the source of a transition, the statechart transitively restores its previous configuration...RHAPSODY only supports deep history.
- If there is no history, the history connector is ignored
- History is saved before the "owner" state is exited



Outline

The Basics

Basic System Reaction

Compound Transitions

History

Transition Scope

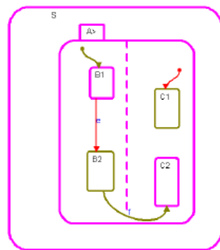
Conflicting Transitions

Basic Step Algorithm

Multi-Threaded Systems

Transition Scope

"The scope of a compound transition is the lowest OR-state in the hierarchy of states that is a proper common ancestor of all the source and target states. Taking the transition will result in a change of the active configuration involving only substates in the scope. [...] Thus, the scope is the lowest state in which the system stays without exiting and reentering when taking the transition."



The system is in states *B2* and *C1* and receives message *f*. By the above definition, the scope of the entailed transition is *S*.

Outline

The Basics

Basic System Reaction

Compound Transitions

History

Transition Scope

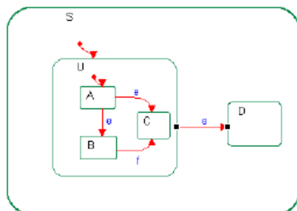
Conflicting Transitions

Basic Step Algorithm

Multi-Threaded Systems

Conflicting Transitions

"Two transitions are in conflict if there is some common state that would be exited if either of them were taken"



- The 2 outgoing transitions from state *A* are in conflict
- The transition from *U* to *D* is in conflict with all the others

Conflicting Transitions...

Two conflicting transitions cannot be taken in the same step. Only one will be chosen...

In the first case, we're faced with non-determinism. RHAPSODY detects this during code generation and blocks it. The motivation for this is that RHAPSODY figures the modeller probably made a mistake.

In the second case, priority is given to lower level states. RHAPSODY justifies this break with the STATEMATE semantics by arguing it fits the OO subclass overriding philosophy.

Finally, priority between compound transitions and static reactions is given to the one with the lowest source state or to the compound transition in case of a tie.

Outline

The Basics

Basic System Reaction

Compound Transitions

History

Transition Scope

Conflicting Transitions

Basic Step Algorithm

Multi-Threaded Systems

Basic Step Algorithm

```
procedure StepCycle ()
begin
  loop forever
    while Event-Queue ≠ empty do
      ev ← Get-Event-From-Queue
      dest ← Get-Destination-Of-Event
      if dest still exists then
        dest → takeEvent(ev)
      else
        Ignore ev
      end if
    end while
  end loop
end
```

Basic Step Algorithm...

The *takeEvent* procedure goes as follows

1. Determine the compound transitions / static reactions that will fire in response to the message
2. Perform the compound transitions / static reactions that we determined should fire (update histories, perform actions, ...)
3. Deal with null transitions
   - take null transitions until a stable state (no more enable null transitions) is reached
   - RHAPSODY will detect –infinite – loops of null transitions and halt or wait for some user-specified maximum

Outline

The Basics

Basic System Reaction

Compound Transitions

History

Transition Scope

Conflicting Transitions

Basic Step Algorithm

Multi-Threaded Systems

Multi-Threaded Systems

Up til now, all objects have shared the same thread. RHAPSODY supports the development of multi-threaded systems.

- Instances of class C will run in their own thread if class C is defined as an *active class*
- Instances of non-active classes will run on the unique system thread used by the main program
- An object's thread can also be set via the *setThread()* command
- The step algorithm consists of performing the basic step algorithm for each thread.

Multi-Threaded Systems

Up til now, all objects have shared the same thread. RHAPSODY supports the development of multi-threaded systems.

- Instances of class C will run in their own thread if class C is defined as an *active class*
- Instances of non-active classes will run on the unique system thread used by the main program
- An object's thread can also be set via the *setThread()* command
- The step algorithm consists of performing the basic step algorithm for each thread.

You can imagine than these added features introduce some delicate issues regarding thread creation/destruction, inter-thread communication and synchronization...

Communication and Synchronization

- The asynchronous case is made quite simple by the fact that each thread has its own synchronized event queue
- The synchronized event queue implies that sending an event to an object can block in a multi-threaded system
- As for the synchronous triggered operations, the sending object is blocked until the receiving statechart completes its response.
- Deadlocks and starvation are possible

Communication and Synchronization

- The asynchronous case is made quite simple by the fact that each thread has its own synchronized event queue
- The synchronized event queue implies that sending an event to an object can block in a multi-threaded system
- As for the synchronous triggered operations, the sending object is blocked until the receiving statechart completes its response.
- Deadlocks and starvation are possible

- Synchronization in RHAPSODY is achieved by defining *guarded* primitive operations.
- A class' guarded operations are all mutually exclusive.
- Unguarded operations can run in parallel.

Questions ?