Coordination of Activity and Behaviour

Herman Bruyninckx
University of Leuven, Belgium
http://people.mech.kuleuven.be/~bruyninc/ecs/

March 4, 2015

Objectives of this lecture

▶ define the complementarities of “Behaviour” & “Activity”
▶ explain how to realise Coordination via state machines, and other means
▶ particular case: the Life Cycle State Machine
▶ common execution policy of Activities

Behaviour & Activity

Behaviour: “reacting to stimuli”
▶ what is “seen” from the outside
▶ software: events and data
▶ hardware: mechanical, electrical, ..., interactions

Activity: “executing the code”
▶ how behaviour is realised “internally”
▶ software: CPU + RAM + bus
▶ hardware: mechanical, electrical, ..., impedance

Coordination:
▶ decides what Behaviour an Activity should have at a particular time
▶ realised by: Finite State Machine; set of rules; if-then-else algorithm; ...
Life Cycle State Machine (LCSM)

Model to start up a Behaviour in an Activity:
- first configure resources needed for Behaviour
- then make Behaviour visible (“active”) to outside world as a new resource with specific capabilities
- switching of Behaviours possible in active super-state

Examples: ABS brake; combustion motor control; lane changing;...

Life Cycle State Machine (cont’d)

Examples of hierarchical Behaviours

Coordination of Activity and Behaviour
Herman Bruyninckx
March 4, 2015
Single-threaded execution of a Composition Pattern Activity

Common (but not absolute!) policy to serialize the execution of the Activity as follows:

```plaintext
when triggered do {
  communicate() % get latest events
  coordinate() % react to them
  configure() % possibly requiring reconfiguration
  schedule() % now do one's Behaviour
  coordinate() % execution could trigger new events
  communicate() % that others might want to know about
  log()
}
```

Concurrent execution of multiple Composition Pattern activities

Do not:
- if state(other_LCSM) == XYZ, then ...
  → too much coupling of internals, not portable

Do:
- outermost Configurator: configures innermost one to translate “contextual” names of event to the latter's LCMS events.
- add Monitor on outermost behaviour to generate events that innermost expects.

Problem with traditional “StateCharts”

(UML) Statecharts = set of states with transitions “e[g]/a”:
- e = event name
- [g] = guard condition
- a = action

Upon receiving event e, if there is a transition from current stateA on e with [g] evaluating to true, then the transition is triggered, executing a and moving the Statechart to the target stateB

But:
- evaluation of guard g can be time-consuming function, not linked one-to-one to one transition, and/or executed in concurrent activity;
- execution of action a can be time-consuming, realised by concurrent activities.
Coordination = logic event processing

Separate action execution and guard evaluation from event computation:
- no “data” entering/leaving Coordinator, only events
- action execution: separate “control” Component(s)
- guard evaluation: separate “monitoring” Component(s)

Best Practice: pure Coordination (2)

Coordinator-Configurator pattern:
- guarantees that other components consistently see only specified behaviour:
  1. specified behaviour before configuration
  2. specified “non-reactive” behaviour during configuration
  3. specified behaviour after configuration
- Configurator shields Coordinator from Computation:
  Coordinator does not have to know itself how to configure component
- Configurator shields Computation from Coordinator: the latter does not have to be aware of the computational state when making decisions about changes in the desired Computation behaviour

Best Practice: pure Coordination (3)

- behaviour of Coordination component = discrete function: first order logic, linear logic, probabilistic state machine, ...
- input: event that “something has happened in the system”
- output: event to trigger other components to (re)configure themselves
  - to distribute
  - easiest to reuse
  - to Verify & Validate formally