Software requirements redux

Today

- Take another look at software requirements
- Two approaches
  - *Use cases* for helping effectively elicit requirements
  - Finite-state specifications of reactive systems

Use cases: a quick preview

- A use case is a description of an example behavior of the system as situated in the world
  - Jane has a meeting at 10AM; when Jim tries to schedule another meeting for her at 10AM, he is notified about the conflict
- Similar to CRC (class responsibility collaborator) and eXtreme programming "stories"

How to use them

- Teams write them down and talk through them
  - Interacting with the customers, of course
- This process raises lots of questions at first, which forces a team to think through and clarify some key issues
- Later it helps in producing the requirements document

Use cases and actors

- *Use cases* represent specific flows of events in the system
- Use cases are initiated by *actors* and describe the flow of events that these actors are involved in
- An actor can be anything that interacts with a use case; it could be a human, external hardware (like a timer) or another system

Jacobson example: recycling

The course of events starts when the customer presses the "Start-Button" on the customer panel. The panel's built-in sensors are thereby activated.

The customer can now return deposit items via the customer panel. The sensors inform the system that an object has been inserted, they also measure the deposit item and return the result to the system.

The system uses the measurement result to determine the type of deposit item: can, bottle or crate.

The day total for the received deposit item type is incremented as is the number of returned deposit items of the current type that this customer has returned...

How are the entities related?

- There are many relationships among the entities in a system – these entities are roughly the actors in the use cases
- These are often captured in a diagram usually called an *object model*

Object models

- There are many "languages" for defining object models
  - All object-oriented modeling techniques have such a language (OMT, UML, Booch, etc.)
- But the heart of these is basically Chen's entity-relationship diagrams (ERDs)
- Basically, boxes represent entities and connectors represent relationships

Trivial example

- Each of the two entities has a single attribute
  - This is similar to an instance variable
- There is a relationship (or association) named *Intersects* between the entities
  - This reads "2 or more Lines intersect in 0 or more Points"
  - Different notations do this in different ways

Line
name
2+
Intersects
Point
name

Aggregation

Microcomputer
1+
Monitor
System box
Mouse
Keyboard
Chassis
CPU
RAM
Fan

- Aggregation represents an *is-part-of* relationship

Aggregation

Whence inheritance?

- These notations indeed support the representation of the inheritance relationship
- However, it's quite unusual for a good requirements object model to include inheritance relationships
  - Ones' design documents might do so, however

Recap

- Use use cases to define instances of the behavior of a system
  - Beware: it's very hard to show completeness of a large collection of use cases

Finite state machines

- What have you used these for before?

Finite-State Specifications

- There is a large class of specification languages based on finite state machines
- Often primarily for describing the control aspects of reactive systems
- The theoretical basis is very firm
  - Lots of theory on finite-state machines, plus analysis support from theorem proving and model checking
  - There are techniques for formally checking the properties of these machines

Walkman example

(due to Alistair Kilgour, Heriot-Watt University)

- What happens when…?
- The implementers need to know, the users need to know, …

Reactive systems

- Essentially event-driven systems that responds to both external (from the environment) and internally-generated stimuli, and also provides stimuli to the external environment
- These are generally embedded systems in which we care about the behavior of the overall system, not the software per se
- As fewer and fewer complex systems are built without software — one can legitimately view this as inappropriate and, in some cases, perhaps even unethical — the pressures on properly specifying (and analyzing) reactive systems increases

Statecharts (Harel)

- A visual formalism for defining finite state machines
- A hierarchical mechanism allows for complex machines to be defined by smaller descriptions
  - Parallel states (AND decomposition)
  - Conventional OR decomposition
- The reduced size of the description is a central piece of the leverage of Statecharts

Walkman example: statechart

Communicating state machines

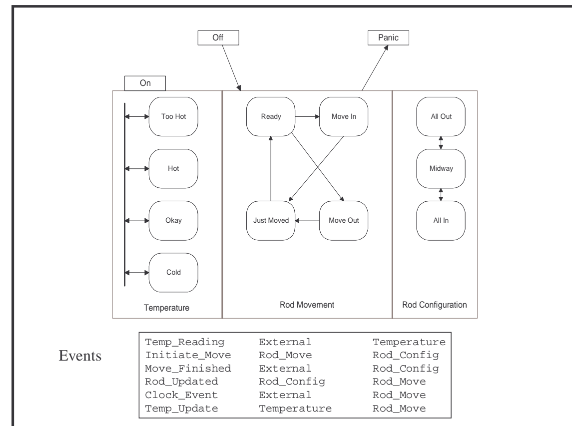
- In conventional state machines, precisely one state must be occupied at a given time
- In Statecharts, every machine in a composition must occupy one state at a given time
  - This allows (in part) the blow-up of representation to be mitigated, because now a pair of communicating state machines can represent NxM states in the overall machine using N+M states

Hierarchical state machines

- Harel's additional insight was to allow the hierarchical definition of state machines
  - It's basically an and-or tree of state machines
  - Machines separated by dotted lines are "and" machines, where each of the machines occupies exactly one state at a time; it's easy to imagine taking the cross product to create a flattened machine
  - Everything else is an "or" machine, essentially like a standard state machine (although they can in turn be nested "and" machines)

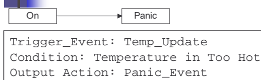
An RSML example

- The following slide shows a very rough "statechart" from RSML
  - RSML is a variant of statecharts developed specifically for the specification of TCAS (Traffic Collision Avoidance System)
- Three high-level states: on, off, and panic
- The on state is expanded and has three parallel states: temperature, rod movement, and rod configuration
- The only non-traditional statecharts feature in this description is the temperature state, which uses a bus that connects all substates (too hot, hot, okay, cold) to one another
- There are six events listed at the bottom (this is an incomplete list)
  - Each event has a name, a description of how it is generated (externally or by a specific sub-machine in the description), and a list of the sub-machines that react to that event



Temp_Reading	External	Temperature
Initiate_Move	Rod_Move	Rod_Config
Move_Finished	External	Rod_Config
Rod_Updated	Rod_Config	Rod_Move
Clock_Event	External	Rod_Move
Temp_Update	Temperature	Rod_Move

Sample transitions




```
Trigger_Event: Temp_Update
Condition: Temperature in Too Hot
Output Action: Panic_Event
```

```
Trigger_Event: Temp_Update
Condition: Rod_Movement in Ready and Temperature in Hot
Output Action: Initiate_Move
```

```
Trigger_Event: Clock_Event
Condition: Rod_Movement in Just_Moved and
           t > t(entered(Just_Moved))+ Move_Delay
```

- This slide shows three sample transitions
- Conditions on the transitions are common
- Output actions are also listed here

Events

- External—interactions with environment
- Internal events, too, to control the machine
- An external event triggers a cascade of internal events (micro steps)
  - Stability reached before next external event (there are other models, too)

Semantics

- What to do when there are multiple events available: which of the enabled transitions should be taken?
- There are literally dozens of (published) choices, with subtle distinctions
- Some of the more theoretically pleasing semantics seem, unfortunately, to be less intuitive to people
- It is, however, critical to have a well-defined semantics; after all, these are specification languages
  - The most common semantics are the "Statemate semantics", Harel and Naamad, which define the formal semantics of statecharts in terms of the operational semantics defined by the Statemate tool
- At the same time, for most "normal" examples, the differences among the semantics are not significant

Reasoning

- The definition of precise semantics allows reasoning of the meaning of statecharts
- Given an initial state
  - And a set of possible external events
  - What states can be reached?

Model checking

- In the last decade an approach called *model checking* has taken off as a way to verify properties of a state machine
- A model checker accepts two inputs
  - A state machine
  - A temporal language formula
- And produces one output
  - Either "true" (the formula is satisfied by the state machine) or "false" (with a counterexample – a trace through the machine – provided)

Model checking II

- The ability to formally and precisely check properties of machines has vastly increased their value
- Model checking is a complex technology that is becoming increasingly ubiquitous not only in research, but also in practice
  - Ex: the SLAM (or Static Driver Verifier) project at Microsoft Research, using model checking to ensure properties of device drivers

Recap

- Use cases can help elicit requirements from customers
- Statecharts can describe reactive systems
  - Precisely
  - Can be analyzed to ensure particular properties
- Lots of other approaches not covered here!