Lecture 6: Requirements Modeling II

Last Week:
Modeling Enterprises
General Modeling Issues
Modeling Human Activity,
i* etc.

This Week:
Modelling Information and Behaviour
Information Structure
Information Flow
Behaviour

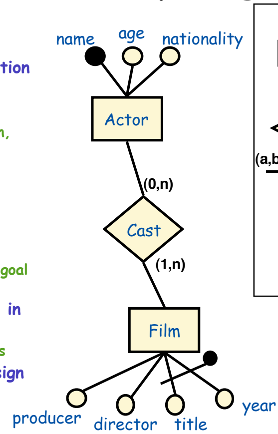
Next Week:
Non-functional requirements
Modelling NFRs
Analysis techniques for NFRs

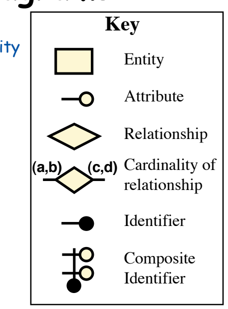
What to Model

- **Information Structure**
  - Entity Relationship Models
  - Class Diagrams (& OO Analysis)
- **Processes and Information Flow**
  - Dataflow diagrams (& Structured Analysis)
  - UML Activity Diagrams
- **System Behaviour**
  - Statecharts
  - Message Sequence Charts
  - Tabular specifications (e.g. SCR)

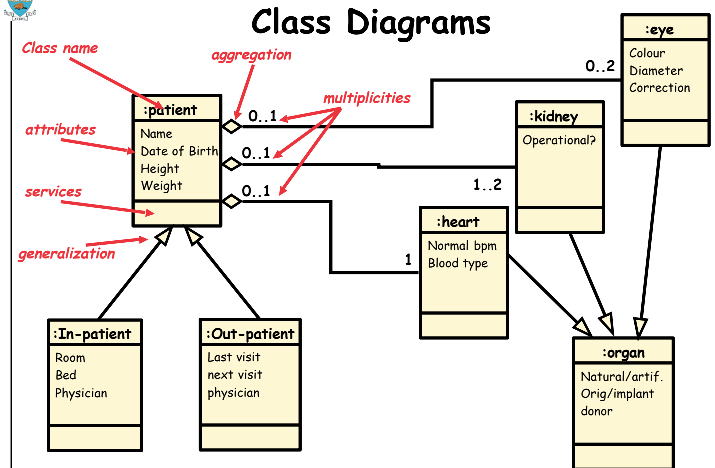
Entity Relationship Diagrams

- **ER diagrams**
  - widely used for information modeling
  - simple, easy to use
    - Note: this is a notation, not a method!
- **Used in many contexts:**
  - domain concepts
    - objects referred to in goal models, scenarios, etc.
  - Data to be represented in the system
    - for information systems
  - Relational Database design
  - Meta-modeling





Class Diagrams



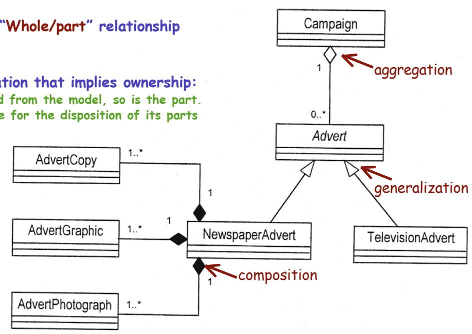
Generalization vs Aggregation

Source: Examples from Bennett, McRobb & Farmer, 2002

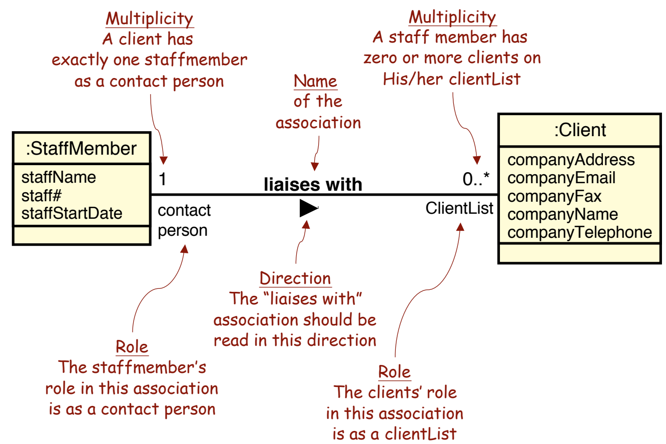
→ **Generalization**
- Subclasses inherit attributes, associations, & operations from the superclass
- A subclass may override an inherited aspect

→ **Aggregation**
- This is the "Has-a" or "Whole/part" relationship

→ **Composition**
- Strong form of aggregation that implies ownership:
  - if the whole is removed from the model, so is the part.
  - the whole is responsible for the disposition of its parts

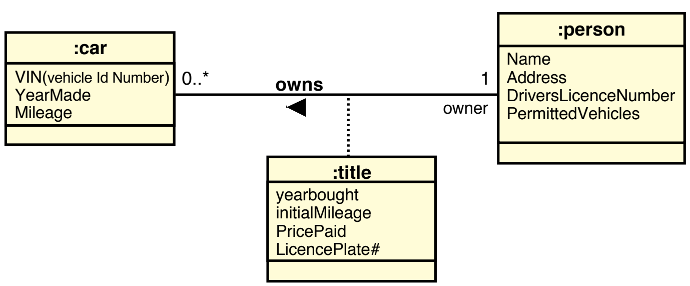


Class associations



Association Classes

→ **Sometimes the association is itself a class**
- …because we need to retain information about the association
- …and that information doesn't naturally live in the classes at the ends of the association
  - E.g. a "title" is an object that represents information about the relationship between an owner and her car



Object Oriented Analysis

→ **Background**
- Model the requirements in terms of objects and the services they provide
- Grew out of object oriented design
  - But applied to modelling the application domain rather than the program

→ **Motivation**
- OO is (claimed to be) more 'natural'
  - As a system evolves, the functions (processes) it performs tend to change, but the objects tend to remain unchanged
  - Hence a model based on functions/processes will get out of date, but an object oriented model will not…
  - …hence the claim that object-oriented designs are more maintainable
- OO emphasizes importance of well-defined interfaces between objects
  - compared to ambiguities of dataflow relationships

NOTE: OO applies to requirements engineering because it is a modeling tool. But we are modeling domain objects, not the design of the new system

Nearly anything can be an object…

Source: *Adapted from Pressman, 1994, p242*

- External Entities
  - …that interact with the system being modeled
    - E.g. people, devices, other systems
- Things
  - …that are part of the domain being modeled
    - E.g. reports, displays, signals, etc.
- Occurrences or Events
  - …that occur in the context of the system
    - E.g. transfer of resources, a control action, etc.
- Roles
  - played by people who interact with the system
- Organizational Units
  - that are relevant to the application
    - E.g. division, group, team, etc.
- Places
  - …that establish the context of the problem being modeled
    - E.g. manufacturing floor, loading dock, etc.
- Structures
  - that define a class or assembly of objects
    - E.g. sensors, four-wheeled vehicles, computers, etc.

Some things cannot be objects:
- *procedures (e.g. print, invert, etc)*
- *attributes (e.g. blue, 50Mb, etc)*

Variants

- Coad-Yourdon
  - Developed in the late 80's
  - Five-step analysis method
- Shlaer-Mellor
  - Developed in the late 80's
  - Emphasizes modeling information and state, rather than object interfaces
- Fusion
  - Second generation OO method
  - Introduced use-cases
- Unified Modeling Language (UML)
  - Third generation OO method
  - An attempt to combine advantages of previous methods

Example method: Coad-Yourdon

Source: *Adapted from Pressman, 1994, p242 and Davis 1990, p98-99*

- Five Step Process:
  1. Identify Objects & Classes *(i.e. 'is_a' relationships)*
  2. Identify Structures *(i.e. 'part_of' relationships)*
  3. Define Subjects
     - A more abstract view of a large collection of objects
     - Each classification and assembly structure become one subject
     - Each remaining singleton object becomes a subject *(although if there a many of these, look for more structure!)*
     - Subject Diagram shows only the subjects and their interactions
  4. Define Attributes and instance connections

  5a. Define services - 3 types:
     - Occur (create, connect, access, release) *These are omitted from the model as every object has them*
     - Calculate (when a calculated result from one object is needed by another)
     - Monitor (when an object monitors for a condition or event)

  5b. Define message connections
     - These show how services of one object are used by another
     - Shown as dotted lines on object and subject diagrams
     - Each message may contain parameters

Unified Modeling Language

- Third generation OO method
  - Booch, Rumbaugh & Jacobson are principal authors
    - Still in development
    - Attempt to standardize the proliferation of OO variants
  - Is purely a notation
    - No modeling method associated with it!
  - But has been accepted as a standard for OO modeling
    - But is primarily owned by Rational Corp. (who sell lots of UML tools and services)
- Has a standardized meta-model
  - Use case diagrams
  - Class diagrams
  - Message sequence charts
  - Activity diagrams
  - State Diagrams (uses Harel's statecharts)
  - Module Diagrams
  - Platform diagrams

Evaluation of OOA

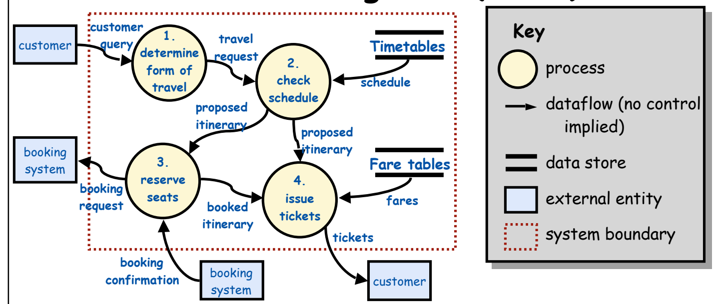
→ Advantages of OO analysis for RE

- Fits well with the use of OO for design and implementation
  - Transition from OOA to OOD 'smoother' than from SA to SD (but is it?)
- Removes emphasis on functions as a way of structuring the analysis
- Avoids the fragmentary nature of structured analysis
  - object-orientation is a coherent way of understanding the world

→ Disadvantages

- Emphasis on objects brings an emphasis on static modeling
  - although later variants have introduced dynamic models
- Not clear that the modeling primitives are appropriate
  - are objects, services and relationships really the things we need to model in RE?
- Strong temptation to do design rather than problem analysis
- Fragmentation of the analysis
  - E.g. reliance on use-cases means there is no "big picture" of the user's needs
- Too much marketing hype!
  - and false claims - e.g. no evidence that objects are a more natural way to think

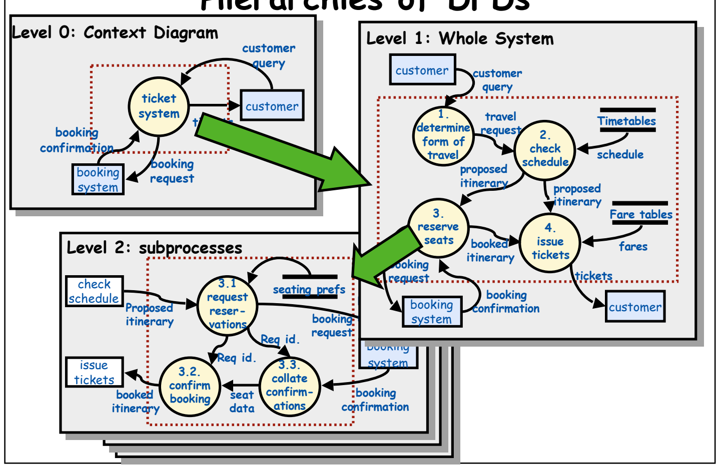
Dataflow Diagrams (DFDs)



→ Notes:

- every process, flow, and datastore must be labeled
- representation is hierarchical
  - each process will be represented separately as a lower level DFD
- processes are normally numbered for cross reference
- processes transform data
  - can't have the same data flowing out of a process as flows into it

Hierarchies of DFDs

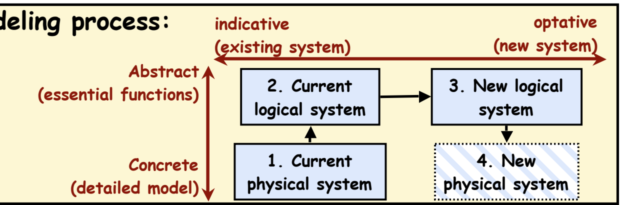


Structured Analysis

→ Definition

- Structured Analysis is a data-oriented approach to conceptual modeling
- Common feature is the centrality of the dataflow diagram
- Mainly used for information systems
  - variants have been adapted for real-time systems

→ Modeling process:

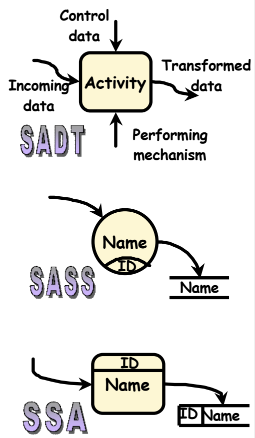


- Model of current physical system only useful as basis for the logical model
- Distinction between indicative and optative models is very important:
  - Must understand which requirements are needed to continue current functionality, and which are new with the updated system

Variants

Source: Adapted from Svoboda, 1990, p264-5

- **Structured Analysis and Design Technique (SADT)**
  - Developed by Doug Ross in the mid-70's
  - Uses activity diagrams rather than dataflow diagrams
  - Distinguishes control data from processing data
- **Structured Analysis and System Specification (SASS)**
  - Developed by Yourdon and DeMarco in the mid-70's
  - 'classic' structured analysis
- **Structured System Analysis (SSA)**
  - Developed by Gane and Sarson
  - Notation similar to Yourdon & DeMarco
  - Adds data access diagrams to describe contents of data stores
- **Structured Requirements Definition (SRD)**
  - Developed by Ken Orr in the mid-70's
  - Introduces the idea of building separate models for each perspective and then merging them



Example method: SASS

Source: Adapted from Davis, 1990, p83-86

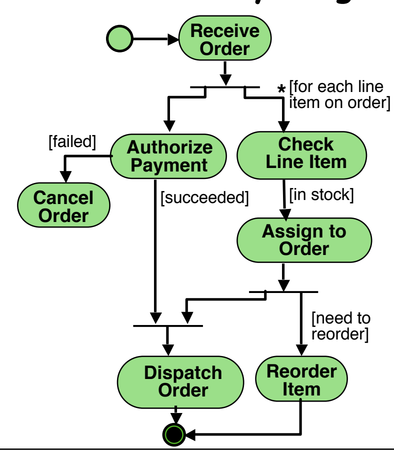
1. **Study current environment**
   - draw DFD to show how data flows through current organization
   - label bubbles with names of organizational units or individuals
2. **Derive logical equivalents**
   - replace names (of people, roles,…) with action verbs
   - merge bubbles that show the same logical function
   - delete bubbles that don't transform data
3. **Model new logical system**
   - Modify logical DFD to show how info will flow once new system is in place
     - …but don't distinguish (yet) which components will be automated
4. **Define a number of automation alternatives**
   - document each as a physical DFD
   - Analyze each with cost/benefit trade-off
   - Select one for implementation
   - Write the specification

Evaluation of SA techniques

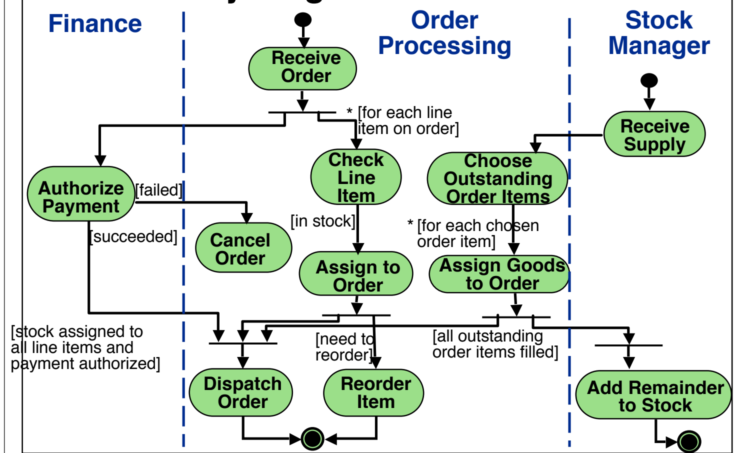
Source: Adapted from Davis, 1990, p174

- **Advantages**
  - Facilitates communication.
  - Notations are easy to learn, and don't require software expertise
  - Clear definition of system boundary
  - Use of abstraction and partitioning
  - Automated tool support
    - e.g. CASE tools provide automated consistency checking
- **Disadvantages**
  - Little use of projection
    - even SRD's 'perspectives' are not really projection
  - Confusion between modeling the problem and modeling the solution
    - most of these techniques arose as design techniques
  - These approaches model the system, *but not its application domain*
  - Timing issues are completely invisible

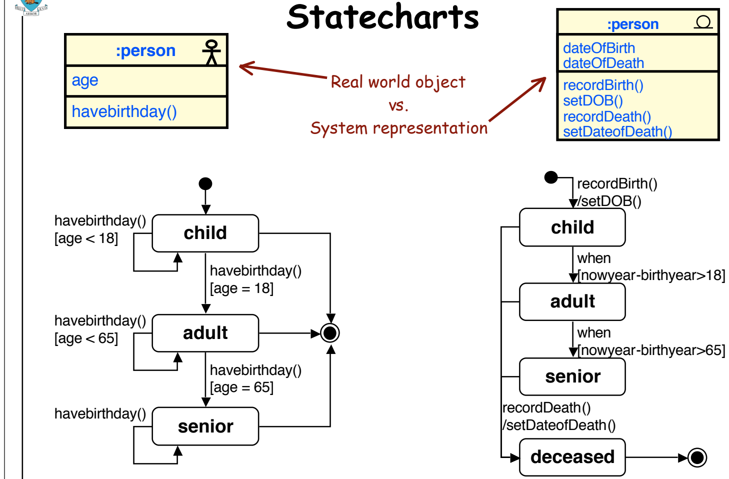
UML Activity Diagrams



Activity Diagram with Swimlanes



Statecharts



States and Transitions

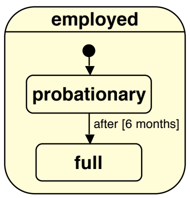
- **A state represents a time period during which**
  - A predicate is true
    - e.g. (budget - expenses) > 0,
  - An action is being performed, or an event is awaited:
    - e.g. checking inventory for order items
    - e.g. waiting for arrival of a missing order item
- **A state can be "on" or "off".**
  - When a state is "on", all its outgoing transitions are eligible to fire.
  - Transitions take the form:
    - event(parameters) [guard] / action
    - For a transition to fire, its event must occur and its guard must be true.
    - When a transition fires, its action is carried out.
- **States can have associated activities:**
  - do/activity
    - carries out some activity for as long as the state is "on"
  - entry/action and exit/action
    - carry out the action whenever the state is entered (exited)
  - include/stateDiagramName
    - "calls" another state diagram, allowing state diagrams to be nested

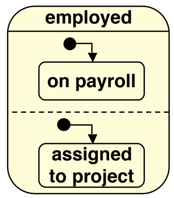
Events

- **Events are happenings the system needs to know about**
  - Must be relevant to the system (or object) being modelled
  - Must be modellable as an instantaneous occurance (from the system's point of view)
    - E.g. completing an assignment, failing an exam, a system crash
  - Are implemented by message passing in an OO Design
- **In UML, there are four types of events:**
  - *Change events* occur when a condition becomes true
    - denoted by the keyword 'when'
    - e.g. when[balance < 0]
  - *Call events* occur when an object receives a call for one of its operations to be perfomed
  - *Signal events* occur when an object receives an explicit (real-time) signal
  - *Elapsed-time events* mark the passage of a designated period of time
    - e.g. after[10 seconds]

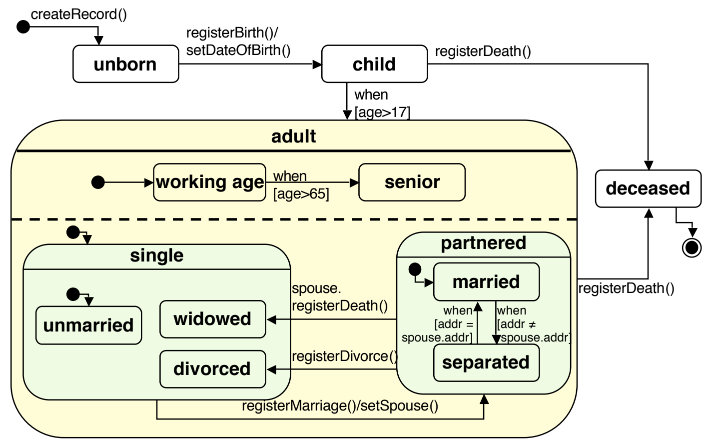
Superstates

- →States can be nested, to make diagrams simpler
  - ✎A superstate consists of one or more states.
  - ✎Superstates make it possible to view a state diagram at different levels of abstraction.
- → OR superstates
  - ✎ when the superstate is "on", only one of its substates is "on"
- → AND superstates (concurrent substates)
  - ✎ When the superstate is "on", all of its states are also "on"
  - ✎ Usually, the AND substates will be nested further as OR superstates





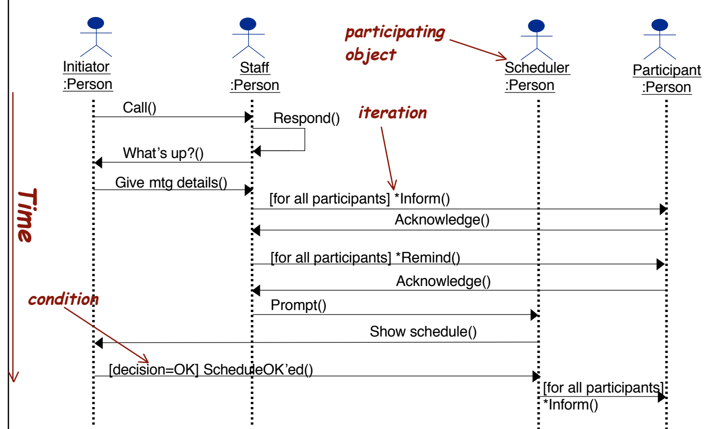
Hierarchical Statecharts



Checking your Statecharts

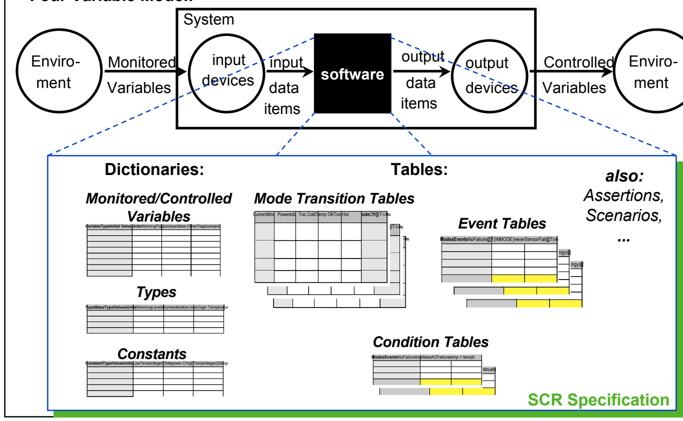
- → Consistency Checks
  - ✎ All events in a statechart should appear as:
    - ➢ operations of an appropriate class in the class diagram **and**
    - ➢ incoming messages for this object on a collaboration/sequence diagram
  - ✎ All actions in a statechart should appear as:
    - ➢ operations of an appropriate class in the class diagram and
    - ➢ outgoing messages for this object on a collaboration/sequence diagram
- → Style Guidelines
  - ✎ Give each state a unique, meaningful name
  - ✎ Only use superstates when the state behaviour is genuinely complex
  - ✎ Do not show too much detail on a single statechart
  - ✎ Use guard conditions carefully to ensure statechart is unambiguous
    - ➢ Statecharts should be deterministic (unless there is a good reason)
- → You probably shouldn't be using statecharts if:
  - ✎ you find that most transitions are fired "when the state completes"
  - ✎ many of the trigger events are sent from the object to itself
  - ✎ your states do not correspond to the attribute assignments of the class

UML Sequence Diagrams



Tabular Specifications: SCR

Four Variable Model:



SCR basics

→ Modes and Mode classes

- A mode class is a finite state machine, with states called *system modes*
  - Transitions in each mode class are triggered by *events*
- Complex systems are described using a number of mode classes operating in parallel

→ System State

- A (system) state is defined as:
  - the system is in exactly one mode from each mode class...
  - ...and each variable has a unique value

→ Events

- An event occurs when any system entity changes value
  - An *input event* occurs when an *input* variable changes value
  - Single input assumption - only one input event can occur at once
  - Notation: @T(c) means "c changed from false to true"
- A conditioned event is an event with a predicate
  - @T(c) WHEN d means: "c became true when c *was* false and d *was* true"

SCR Tables

→ Mode Class Tables

- Define the set of *modes* (states) that the software can be in.
- A complex system will have many different modes classes
  - Each mode class has a mode table showing the conditions that cause transitions between modes
- A mode table defines a *partial function* from modes and events to modes

→ Event Tables

- An event table defines how a term or controlled variable changes in response to input events
- Defines a *partial function* from modes and events to variable values

→ Condition Tables

- A condition table defines the value of a term or controlled variable under every possible condition
- Defines a *total function* from modes and conditions to variable values

Example: Temp Control System

Mode transition table:

Current Mode	Powered on	Too Cold	Temp OK	Too Hot	**New Mode**
Off	@T	-	t	-	**Inactive**
	@T	t	-	-	**Heat**
	@T	-	-	t	**AC**
Inactive	@F	-	-	-	**Off**
	-	@T	-	-	**Heat**
	-	-	-	@T	**AC**
Heat	@F	-	-	-	**Off**
	-	-	@T	-	**Inactive**
AC	@F	-	-	-	**Off**
	-	-	@T	-	**Inactive**

Failure modes

Mode transition table:

Current Mode	Powered on	Cold Heater	Too Cold	Warm AC	Too Hot	**New Mode**
NoFailure	t	@T	t	-	-	**HeatFailure**
	t	-	-	@T	t	**ACFailure**
HeatFailure	t	@F	t	-	-	**NoFailure**
ACFailure	t	-	-	@F	t	**NoFailure**

Event table:

Modes		
NoFailure	@T(INMODE)	never
ACFailure, HeatFailure	never	@T(INMODE)
Warning light =	**Off**	**On**

Source: *Adapted from Heitmeyer et. al. 1996.*

Consistency Checks in SCR

- **Syntax**
  - did we use the notation correctly?
- **Type Checks**
  - do we use each variable correctly?
- **Disjointness**
  - is there any overlap between rows of the mode tables?
    - ensures we have a deterministic state machine
- **Coverage**
  - does each condition table define a value for all possible conditions?
- **Mode Reachability**
  - is there any mode that cannot ever happen?
- **Cycle Detection**
  - have we defined any variable in terms of itself?