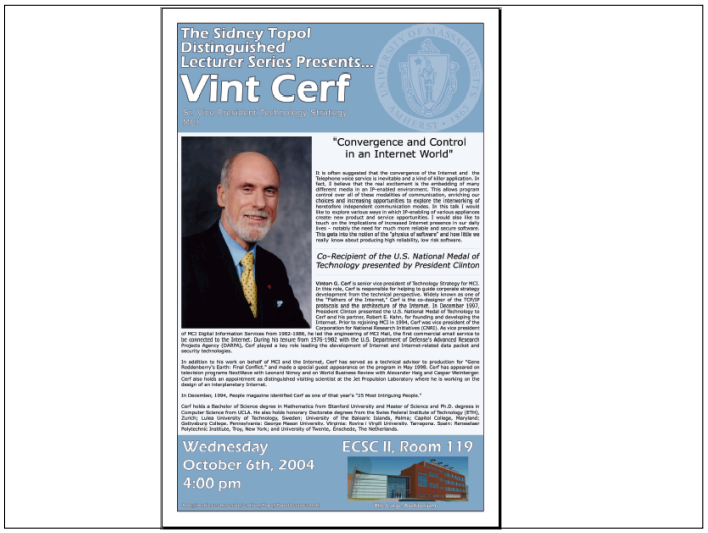
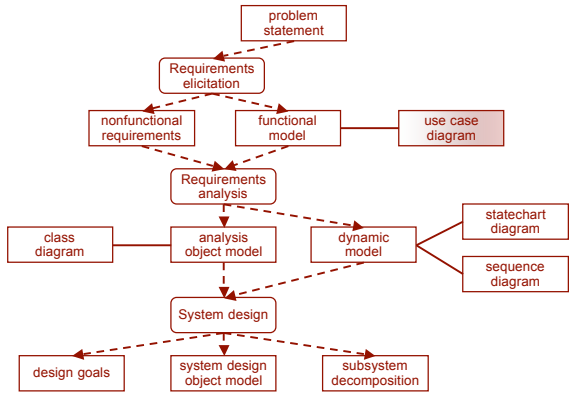


08 Use Cases & UML Overview

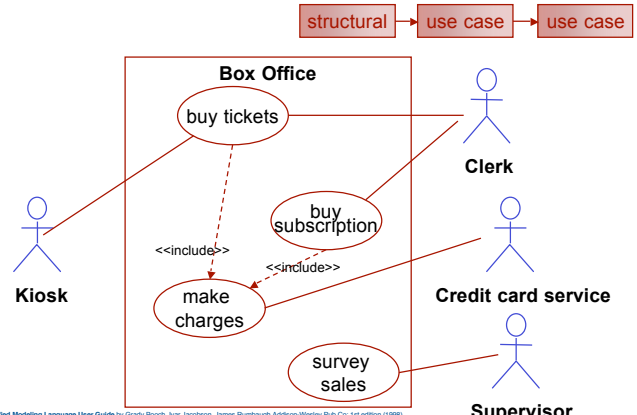
- OMG Tutorial Series
  - [cK00] Kobryn Cris, "Lecture 1: Introduction to UML: Structural and Use Case Modeling Object Modeling with OMG," UML Tutorial Series
    http://www-edlab.cs.umass.edu/cs520/OMG-Tutorials/Tut1IntroToUML.ppt
  - [OSBB00] Övergaard, Gunnar, Bran Selic, Conrad Bock and Morgan Björkander, "Lecture 2: Behavioral Modeling with UML," Object Modeling with OMG UML Tutorial Series
    http://www-edlab.cs.umass.edu/cs520/OMG-Tutorials/Tut2BehaviorModeling.ppt
  - [PSWD00] Palmkvist, Karin, Bran Selic, Jos Warmer and Nathan Dykman, "Lecture 3: Advanced Modeling with UML," Object Modeling with OMG UML Tutorial Series
    http://www-edlab.cs.umass.edu/cs520/OMG-Tutorials/Tut3AdvModeling.ppt
  - Note: This version of the tutorial series is based on OMG UML Specification v. 1.4, UML Revision Task Force recommended final draft, OMG doc# ad/01-02-13.
- Other
  - [BJR98] **The Unified Modeling Language User Guide** by Grady Booch, Ivar Jacobson, James Rumbaugh Addison-Wesley Pub Co; 1st edition (1998)
  - [dB03] Bell, Donald, "UML basics: An introduction to the Unified Modeling Language," The Rational Edge, June 2003
    http://www-106.ibm.com/developerworks/rational/library/769.html

O-O System Development



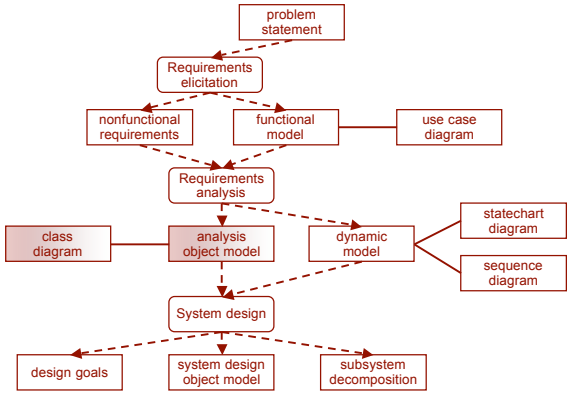
adapted from Bruegge/Dutoit O-O SW Engr

Use Case Diagram



The Unified Modeling Language User Guide by Grady Booch, Ivar Jacobson, James Rumbaugh Addison-Wesley Pub Co; 1st edition (1998)

O-O System Development



adapted from Bruegge/Dutoit O-O SW Engr

Classes

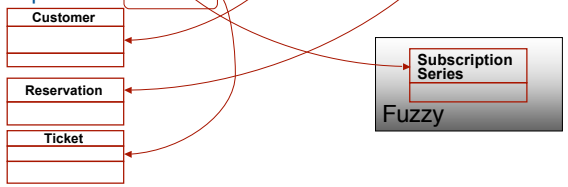
- A class is a collection of objects with common structure, common behavior, common relationships and common semantics
- Classes may be found by examining:
  - written requirements
  - objects in sequence and collaboration diagrams
- A class is drawn as a rectangle with three compartments
  - name
  - attributes
  - operations
- Classes should be named using the vocabulary of the domain
  - Naming standards should be created
  - e.g., all classes are singular nouns starting with a capital letter

Class Modeling

- Captures **system state** – the function of the system's information content at a point in time
- Class modeling and use case modeling are typically conducted in parallel
- A **class diagram** shows the existence of classes and their relationships in the logical view of a system; in UML class diagrams elements include
  - Classes and their structure and behavior
  - Association, aggregation, generalization, dependency, and inheritance relationships
  - Multiplicity and navigation indicators
  - Role names

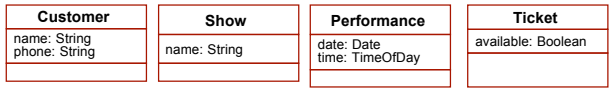
Find Classes from requirements

- Design a program for a booking office of an arts center. There are several theatres, people may reserve seats at any theatre for any future event, and people may subscribe to a series of events. People need to be able to discuss seat availability, where seats are located, and how much they cost. When people make a choice, the program should print the price, record the selection, and print out a ticket



Attributes

- The structure of a class is represented by its attributes
- Attributes may be found by examining class definitions, the problem requirements, and by applying domain knowledge
- More requirements
  - Each customer offering has a name and phone number
  - Each show has a name
  - Each performance has a date and time
  - Ticket has availability



Operations

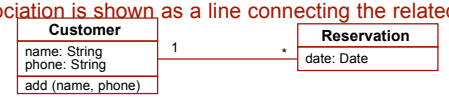
- The behavior of a class is represented by its operations
- Operations may be found by examining interaction diagrams

Relationships

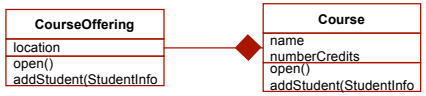
- Relationships provide a pathway for communication between objects
- Sequence and/or collaboration diagrams are examined to determine what links between objects need to exist to accomplish the behavior -- if two objects need to "talk" there must be a link between them
- Three types of relationships are:
  - Association
  - Aggregation
  - Dependency

Relationships

- An association is a bi-directional connection between classes
  - An association is shown as a line connecting the related classes

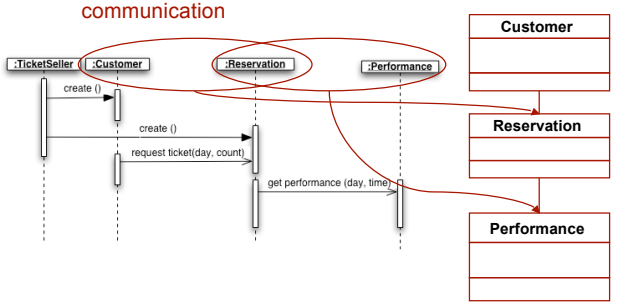


- An aggregation is a stronger form of relationship where the relationship is between a whole and its parts
- An aggregation is shown as a line connecting the related classes with a diamond next to the class representing the whole



Finding Relationships

- Relationships are discovered by examining interaction diagrams
  - If two objects must "talk" there must be a pathway for communication



Multiplicity and Navigation

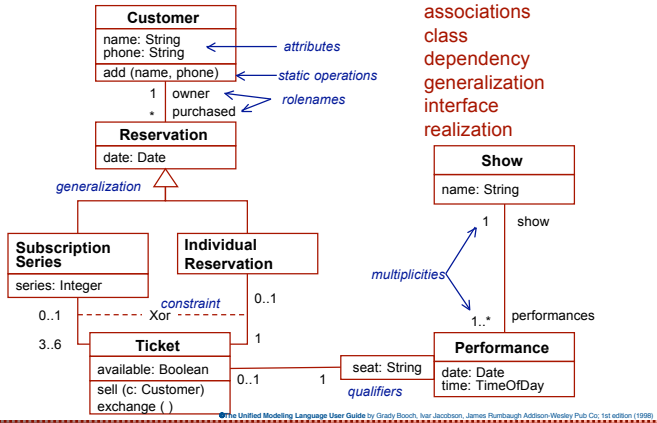
- Multiplicity defines how many objects participate in a relationships
  - Multiplicity is the number of instances of one class related to ONE instance of the other class
  - For each association and aggregation, there are two multiplicity decisions to make: one for each end of the relationship
- Although associations and aggregations are bi-directional by default, it is often desirable to restrict navigation to one direction
- If navigation is restricted, an arrowhead is added to indicate the direction of the navigation

Inheritance

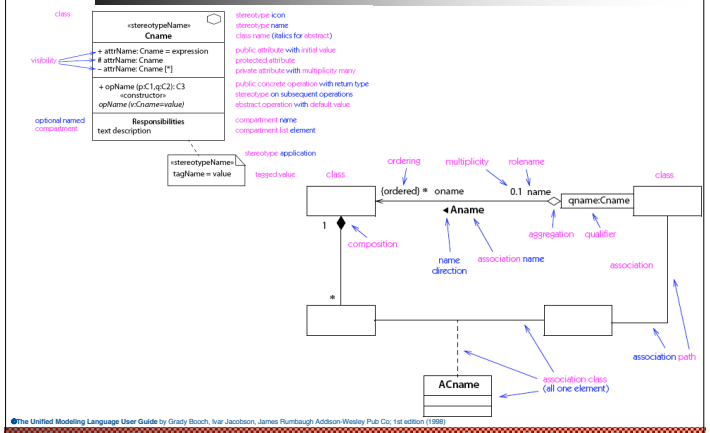
- Inheritance is a relationships between a superclass and its subclasses
- There are two ways to find inheritance:
  - Generalization
  - Specialization
- Common attributes, operations, and/or relationships are shown at the highest applicable level in the hierarchy

Class Diagram

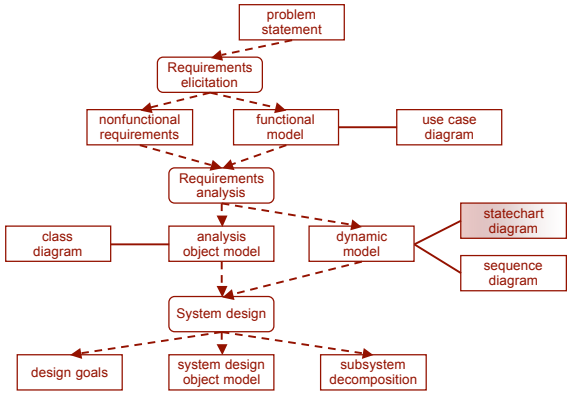
structural → static → class



Class & associations -- notation

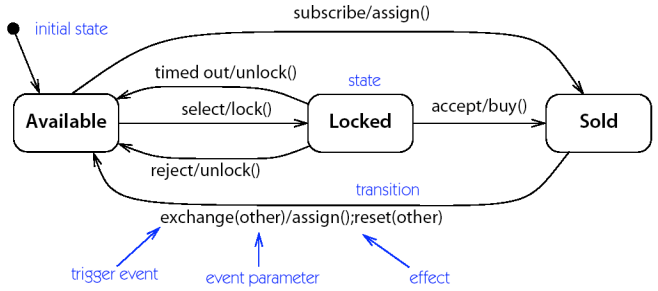


O-O System Development

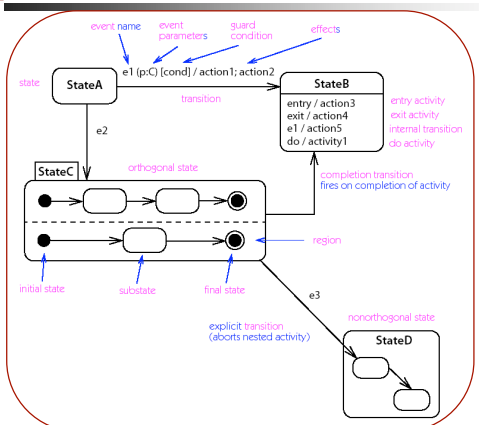


adapted from Bruegge/Dutoit O-O SW Engr

State Machine View



State Machine Notation

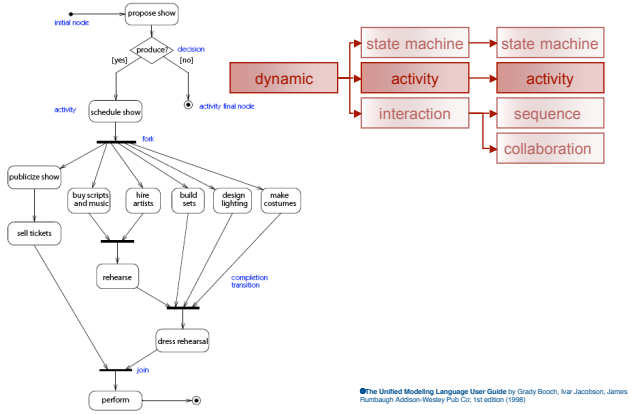


UML 2 Activity Diagrams

- object-oriented equivalent of flow charts and data flow diagrams (DFDs) from structured development
- typically used for
  - business process modeling
  - modeling the logic captured by a single use case or usage scenario
  - modeling the detailed logic of a business rule
- could potentially model
  - the internal logic of a complex operation
  - far better to simply rewrite the operation so that it is simple enough that you don't require an activity diagram

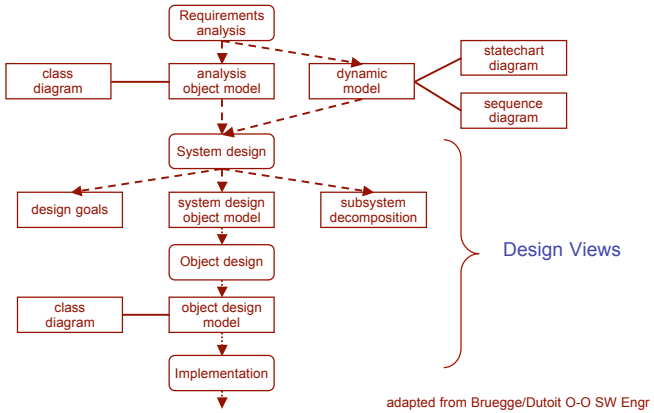
Scott W. Ambler , Copyright 2003 www.agilemodeling.com/

Activity Diagram



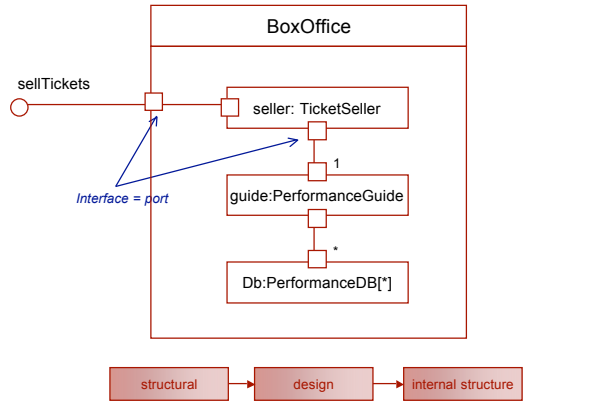
The Unified Modeling Language User Guide by Grady Booch, Ivar Jacobson, James Rumbaugh Addison-Wesley Pub Co; 1st edition (1998)

O-O System Development



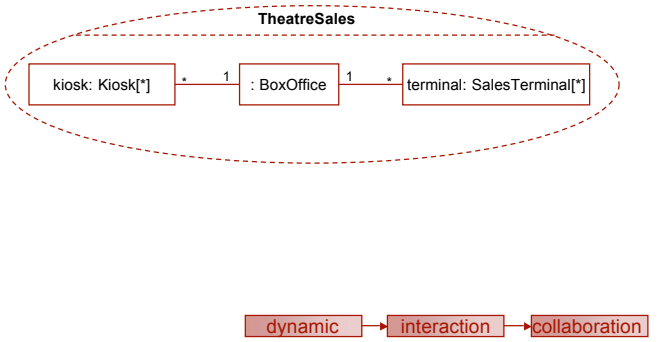
adapted from Bruegge/Dutoit O-O SW Engr

Internal Structure Diagram



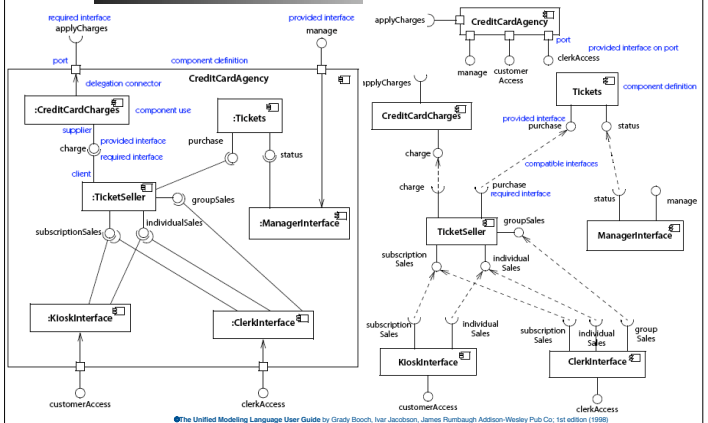
The Unified Modeling Language User Guide by Grady Booch, Ivar Jacobson, James Rumbaugh Addison-Wesley Pub Co; 1st edition (1998)

Collaboration Diagram



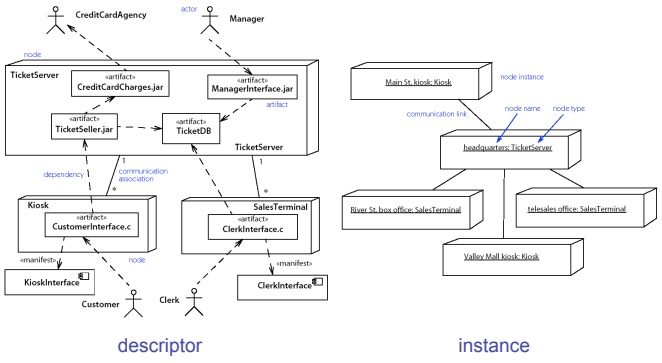
The Unified Modeling Language User Guide by Grady Booch, Ivar Jacobson, James Rumbaugh Addison-Wesley Pub Co; 1st edition (1998)

Component Definition & Diagram



The Unified Modeling Language User Guide by Grady Booch, Ivar Jacobson, James Rumbaugh Addison-Wesley Pub Co; 1st edition (1998)

Deployment views

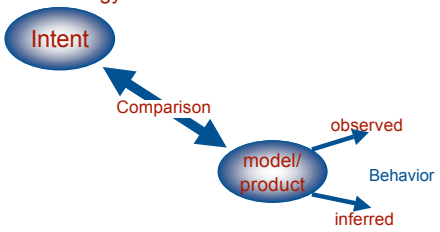


UML -> Notation

- We'll come back to UML in the RUP (design methods)
- Reading
  - **[jmW90]** Wing, J. M., "A Specifier's Introduction to Formal Methods," **IEEE Computer** , September 1990, pp.8--24.
  - [avL00] van Lamsweerde, Axel, "Formal Specification: a Roadmap," **Future of Sofware Engineering** Limerick Ireland 2000
  - **[LZ75]** Liskov, B.H. and Zilles, S.N., "Specification Techniques for Data Abstractions," **IEEE Transactions on Software Engineering** , March 1975, pp.7--19.
  - **[GHW85]** Guttag, J.V., Horning, J.J. and Wing, J.M., "The Larch family of Specification Languages," **IEEE Software** , September 1985, pp.24--36.

Overview of Formal Methods

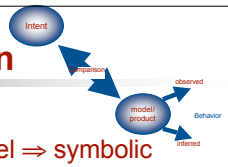
- Formal methods
  - mathematically-based languages, techniques and tools for specifying and verifying software and systems
  - specification ⇔ verification
  - basic strategy



Basic Verification Strategy

- analyze a system for desired properties, i.e., compare behavior to intent
  - intent
    - can be expressed as properties of a model (**model-based specification**)
    - can be expressed as formulas in mathematical logic (**property-based specification**)
  - behavior
    - can be observed as software executes
    - can be inferred from a model
    - can be expressed as formulas in mathematical logic
  - different representations support different sorts of inferences

finite-state verification



- model checking
  - logic spec + FSA comp model ⇒ symbolic model checking
  - FSA spec + FSA comp model ⇒ automata-theoretic model checking
- property checking
- advantages/disadvantages
  - reason about a finite model of the system
  - fast, yields counterexamples, manages partial specifications, applies to concurrency
  - state explosion!

(automated) mathematical reasoning

- theorem proving
- proof checking
- advantages/disadvantages
  - difficult, error prone
  - decidability vs. expressiveness
    - propositional calculus is decidable
    - predicate calculus is semi-decidable

Specifications

- define intent and provide a basis for formal reasoning
  - should be based on a sound mathematical theory
- criteria to evaluate specification methods (languages)
  - mathematical foundation
  - constructability (ease of use)
  - comprehensibility
  - minimality
  - general applicability
  - extensibility

What is a specification language?

- A formal specification language is a triple
  <Syn, Sem, Sat >, where Syn and Sem are **sets**
  Syn X Sem ⊃ Sat is a r**elation**.
- Given a specification language, <Syn, Sem, Sat>
  - if Sat (*syn, sem*) then *syn* is a **specification** of *sem* and *sem* is a **specificand** of *syn*
  - the **specificand set** of a specification *syn* ∈ Syn is the set of all specificands *sem* ∈ Sem, such that
    Sat (*syn,sem*)

from Wing

Properties

- a specification *syn* ∈ Syn is **unambiguous** if and only if Sat maps *syn* to exactly one specificand set.
- a specification *syn* ∈ Syn is **consistent** (or **satisifable**) if and only if Sat maps *syn* to a non-empty specificand set.
- Given <Syn, Sem, Sat >, an **implementation** *prog* ∈ Sem is **correct** with respect to a given specification *spec* ∈ Syn if and only if Sat (*spec, prog*)
- informally, a specifier who "overspecifies" is guilty of "**implementation bias**"
  - a specification has implementation bias if it specifies unobservable properties of its specificands,
  - e.g., a set specification that keeps track of the insertion order favors an ordered-list implementation over a hash table implementation

Classification

- Model-oriented (operational) specification
  - behavior described in terms of another data abstraction or mathematical model with known properties, e.g., tuples, relations, functions, sets, and sequences
- Property-oriented (descriptive) specification
  - behavior is described in terms of properties, usually stated as axioms, that the system must specify
  - or the objects and operations to define themselves implicity
- Formal vs "semi-formal" vs informal

Alternative classification

- Axiomatic specification
- Abstract models
- Set Theory
- Predicate Logic
- Programming Languages

Model-oriented examples

- Formal:
  - Abstract-data-type specification languages: Parnas' state machines, VDM, Z
  - Concurrent and distributed systems specification languages: Trace Specifications, Petri nets, CCS, CSP
- Semi-Formal
  - Diagrams
    - Behavior: FSA, Petri-Nets, StateCharts
    - Communications: DFD, activity diagrams, sequence diagrams
    - Functions: Use-Case diagrams

Semi-Formal Technques

- Communication: DFD
  - lack precise semantics
  - abstract "machine" for interpreting the operational semantics of a DFD specification is not fully defined
  - can't simulate behavior
- Behavior: FSA
  - limited memory
  - combinatorial explosion

abstract data type example

type stack is
create: ⇒ stack
pop: stack ⇒ stack
push: stack X integer ⇒ stack
top: stack ⇒ integer

Note: Because some of the specification methods are easier to apply to functions, all operations are functions

Input/Output Specification

- type definition:

  type S is record

  top: integer

  data: array [1 ...] of integers

  end record

- operational specification:

  {true} push (S_0, I) ⇒ S

  {∀ J, 1 < J ≤ S_0.top

  S_0.data [J] = S.data [J] ∧

  S.top = S_0.top + 1 ∧

  S.Data [S.top] = I }

Ordered Sets

- ordered set definition:

  X = {x_0,x_1, ... ,x_n}

  |X| = n +1

  extract(X) = {x_0,x_1, ... ,x_{n-1}}

- operational definitions:

  create = { 0 }

  push (S_0, I) = S ∧

  S_0 = extract(S) ∧

  |S| = |S_0| + 1 ∧

  $x_{|S|}$ = I

Z ("zed")

- proposed by Abrail, 1980
- developed by Hayes and Spivey
- based on typed set theory and first order logic
- provides a schema to describe a specifications state and operations
- describe systems as collections of SCHEMAS
  - inputs and outputs to functions
  - Invariants: statements whose truth is preserved by the functions

Z

- a schema groups variable declarations with a list of predicates that constrain the possible values for a variable

schema name

schema signature

schema predicate

The "Birthday Book" Example

- N… …days
- …are to be
- …date
- S… …em: the s…

Possible state of system

known = {John, Mike, Susan}
birthday = {John ↦ 25-Mar,
Mike ↦ 20-Dec,
Susan ↦ 20-Dec}

BirthdayBook
known : 𝐏 *NAME*
birthday: *NAME* ↦ *DATE*

known = **dom** *birthday*

set — elements — function — invariant

Another Schema

$$known' = known \cup \{name?\}.$$

In fact we can *prove* this from the specification of *AddBirthday*, using the invariants

stat… $known'$

$$\begin{aligned}
&= \mathrm{dom}\, birthday' && \text{[invariant after]}\\
&= \mathrm{dom}(birthday \cup \{name? \mapsto date?\}) && \text{[spec. of } AddBirthday\text{]}\\
&= \mathrm{dom}\, birthday \cup \mathrm{dom}\, \{name? \mapsto date?\} && \text{[fact about dom]}\\
&= \mathrm{dom}\, birthday \cup \{name?\} && \text{[fact about dom]}\\
&= known \cup \{name?\}. && \text{[invariant before]}
\end{aligned}$$

Another Schema

FindBirthday
Ξ*BirthdayBook*
name?: *NAME*
date!: *DATE*

name? ∈ *known*
date! = *birthday* (*name?*)

no state change — invariants — apply fn

Z Summary

- Schemas can be grouped and composed
- More notation: aimed at facilitating terse, precise communication
- Emphasis on what a system is supposed to do
- Indication of how it looks externally
- (Like Abstract Data Type specifications) basis for going on to think about HOW to implement

State machine model

- 2 types of operations
  - V-Operations (value returning)
    - Do not cause a change in state
  - O-Operations
    - Cause a change in state
- specs must show the effect of each operation on the V-operations

Example

- V-operation: TOP
  possible values: integers; initially undefined
  parameters: none
  effect:
  error call if 'DEPTH' = 0
- O-operation: PUSH(a)
  possible values: none
  parameters: integer a
  effect:
  error call if 'DEPTH' = MAX
  else (TOP =a; 'DEPTH' = 'DEPTH'+1)

Hidden Operations

- must deal with side effects and delayed effects, such as the effect of PUSH on TOP
- V-operation: DEPTH
  possible values: integer; initial value 0
  parameters: none
  effect: none
- Parnas had informal language, later hidden operations were used to support the provided O & V operations. In both cases, need to show that 0≤ Depth (S) ≤ MAX

Concurrent & distributed systems

- FSA
- Petri nets
- Trace specifications
  - a trace is a sequence of procedure or function calls and return values from those calls
    - proposed by David Parnas, 1977
    - formalized by McLean, 1984
    - further developed by Dan Hoffman, Rick Snodgrass, etc

Finite State Machines (FSM's)

- FSM's describe behavior of a system:
  - The sequence of stages/steps/conditions that the system goes through
  - FSM shows how a system acts/reacts to inputs
  - Does this by showing progress through different states
- Hypothesis:
  - The universe in which the system being described must operate can be accurately modeled as always being in exactly one of a finite number of states (situations)
  - There are only a finite number of possible system inputs

Finite State Machines (FSM)

- finite set of states S ={s_1, ... , s_n}
- finite set of inputs I = {i_1, ..., i_n}
- transition function δ: SxI $\Rightarrow$ S
  - can be a partial function
- represent as a graph
  - nodes $\Rightarrow$ states
  - edges $\Rightarrow$ inputs
  - graph $\Rightarrow$ transition function

Why Use FSM's?

- Primary appeal is visualization
- Intuitively: Can "watch" a stream of inputs "drive" the behavior of the system as a sequence of movements from state to state
- Kinds of questions FSM's seem adept at helping answer:
  - "What is a good way to think about the problem to be solved?"
  - "What is the solution approach?"
  - "How does this program work?"

Enhancements to FSM's

- Use of hierarchy
- Output annotations on edges
- Distinguished Initial and Terminal states
- Separate data definitions, local and global variables

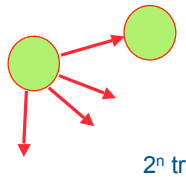
What is FSM good/not good for?

- Focus on specific issue: safety concern
  - Model unsafe state
  - Model state transitions
  - Can unsafe state be reached?
- Drawbacks
  - No sense of functionality
  - No sense of how functionality achieved
  - Difficult and generally impossible to reason about timing

FSM are limited

- Library example
- getbook: index X library ⇒ book
- $1{,}000{,}000^{+}$ books in a good library

state = books on shelf

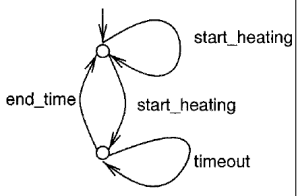


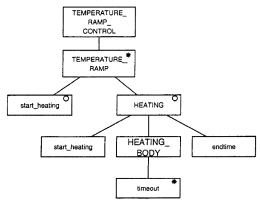
new state = libe - book

2^n transitions could be >> 10^6 states

Some FSM-based notations

- Process Graphs
  - nondeterministic
  - may have infinitely many nodes
  - from any node, infinitely many edges may depart
  - used as interpretation structures for formal specifications in process algebra and dynamic logic
- JSD Process Structure Diagrams.
  - used in Jackson Structured Programming (JSP)
    - represent the structure of files and of regular programs
  - used in JSD
    - represent the behavior of a system in a modular way
  - a visual way to represent a regular expression by means of a tree diagram





FSM

- Finite State Diagrams
  - contain only finitely many states and transitions
- Extended Finite State Diagrams.
  - number of states can be increased by introducing variables that may be tested and updated by the finite state machine
  - *global state* = *explicit state* (STD nodes) + *extended state* (variables)
  - local variables or external variables
    - local variable is declared together with the specification of the STD + scope rules for these variables (*usually the entire state machine specification*)
    - external variable is declared outside the specification of the STD but can be accessed by means of special operations that act as an interface between the specification and the variables
    - in dataflow models, data stores are external variables with respect to the control processes in the DFD
    - global state change requires communication between the state machine and the external data stores

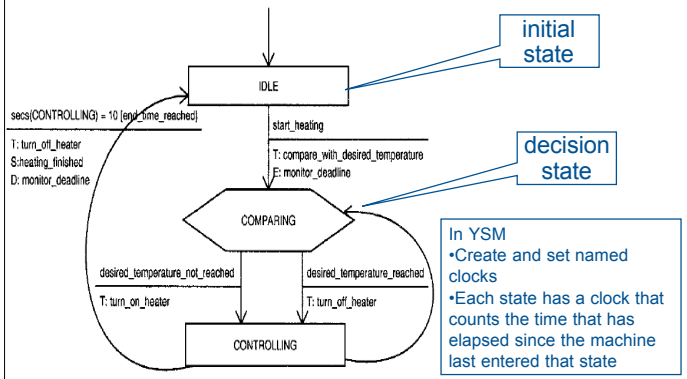
Extended FSM

- state transition may change the values of variables
- a guard may be specified for each transition that says when the transition can occur.
  - weak interpretation,
    - the transition cannot occur if the guard is false
    - guard is in this case a necessary condition of the transition
  - strong interpretation,
    - the transition can occur if and only if the guard is true
    - guard is a necessary and sufficient condition of the transition
  - usually initially specify guards with the weak semantics
    - when all conditions for a transition are specified, interpret the conjunction of all weak guards as a strong guard
    - a guard could be the conjunction of all preconditions specified for a transition
- include tests in a state machine that are used to determine the next state
  - such tests can be used to resolve nondeterminism
  - a test determines which of a set of possible transitions will occur, thus a test consists of a guard for each of the possible transitions.

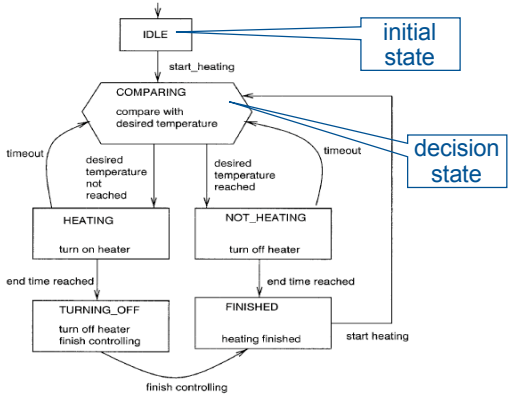
Mealy & Moore Machines

- Mealy machine
  - output actions are associated with transitions
- Moore Machines
  - outputs are associated with states

Mealy FSM for the juice plant controller

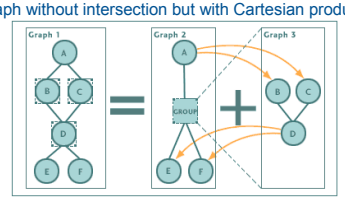


Moore FSM for the juice plant controller



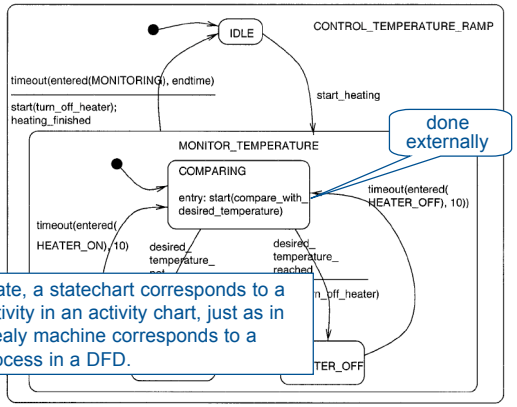
Statecharts

- higraph without intersection but with Cartesian products



- node inclusion allows us to partition a state into substates
- Cartesian products allow us to specify parallelism
- actions can be specified
  - along transitions (Mealy)
  - upon entry of states (Moore), and
  - exit of states
- local variables represent the extended state.

Statecharts



In Statemate, a statechart corresponds to a control activity in an activity chart, just as in YSM a Mealy machine corresponds to a control process in a DFD.

Statechart with parallelism