Declarative specifications

ER diagrams: semiformal specs
Logic specifications
Algebraic specifications
ER diagrams

- Often used as a complement to DFD to describe conceptual data models
- Based on entities, relationships, attributes
- They are the ancestors of class diagrams in UML
Example
Relations

- Relations can be partial
- They can be annotated to define
  - one to one
  - one to many
  - many to one
  - many to many
Non binary relations
Fast Food Restaurant System

- Order Taking unit
- Assembly unit
- Food Preparation unit
- Inventory unit
- Management unit

Fast-food restaurant system:
- General scope
- Network view
E-R diagram of the Restaurant System

- **Direction of relation:**
  - Left-right
  - Bottom-up

- **1 to 0, 1, many**
- **1 to 1, many**
- **1 to 1**

- **Selection**

- **Order attr.:**
  - order-number
  - date/time
  - status (completed, stored, paid)
  - total price/tax
  - number of items

- **Menu-item attr.:**
  - name
  - list of ingredients
  - recipe
  - min. level quantity
  - stock quantity
  - price
  - anticipated chute-items

- **Raw-material attr.:**
  - name
  - stock quantity
  - price

- **Cash attr.:**
  - cash balance
  - float
  - skin

- **Stock on order**
- **Supplies**
- **Raw materials in stock**

- **Restaurant menu**
- **Consists of**
- **Consists of**
- **Consists of**

- **Order in OT station**
- **Unpaid orders**
- **Completed orders**

- **Order in Aam station**
- **Customer's order**

- **Cooking items**
- **Satisfies**

- **Supplies**

- **Fast-food restaurant system:**
  - Owner's perspective
  - Data view
Logic Specifications

Slides from Other Sources
Mathematical Fundamentals

• Chapter 3 from book:
  – By: Daniel Hoffman, Paul A. Strooper
  – Publisher: Thomson Computer Press
Propositional Logic

Slides:

Sets

Functions

First Order Predicate Logic

Logic specifications

A logical formula is a statement whose truth can be determined, e.g., 5<7 is true and 7<5 is false.

A formula in first order logic is an expression involving variables, numeric constants, functions, predicates, parentheses, logical connectivity (not/and/or/imply/equivalent) and quantifiers (exists, for all)

Examples of first-order logic formulas:
• \( x > y \) and \( y > z \) implies \( x > z \)
• \( x = y \equiv y = x \)
• for all \( x, y, z \) (\( x > y \) and \( y > z \) implies \( x > z \))
• \( x + 1 < x - 1 \)
• for all \( x \) (exists \( y \) (\( y = x + z \)))
• \( x > 3 \) or \( x < -6 \)
Specifying complete programs

A \textit{property}, or \textit{requirement}, for P is specified as a formula of the type

\[
\{\text{Pre } (i_1, i_2, \ldots, i_n) \} \\
\text{P} \\
\{\text{Post } (o_1, o_2, \ldots, o_m, i_1, i_2, \ldots, i_n)\}
\]

\textit{Pre}: precondition
\textit{Post}: postcondition
\textit{i}'s and \textit{o}'s are free variables
Example

• Program to compute greatest common divisor

PRE \{i1 > 0 \text{ and } i2 > 0\}

\[ P \]

POST \{(\exists z_1, z_2 \ (i1 = o \times z_1 \text{ and } i2 = o \times z2) \text{ and not } (\exists h \ (\exists z_1, z2 \ (i1 = h \times z_1 \text{ and } i2 = h \times z2) \text{ and } h > o))\}\}
### Specifying procedures

**Preconditions (PRE)**

{n > 0} -- n is a constant value

**Procedure `search`**

<table>
<thead>
<tr>
<th>Parameter(s)</th>
<th>Type(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>table</code></td>
<td><code>in</code> integer_array</td>
</tr>
<tr>
<td><code>n</code></td>
<td><code>in</code> integer</td>
</tr>
<tr>
<td><code>element</code></td>
<td><code>in</code> integer</td>
</tr>
<tr>
<td><code>found</code></td>
<td><code>out</code> Boolean</td>
</tr>
</tbody>
</table>

**Postconditions (POST)**

{found \equiv (\exists i \ (1 \leq i \leq n \text{ and } table(i) = element))}

**Preconditions (PRE)**

{n > 0}

**Procedure `reverse`**

<table>
<thead>
<tr>
<th>Parameter(s)</th>
<th>Type(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a</code></td>
<td><code>in out</code> integer_array</td>
</tr>
<tr>
<td><code>n</code></td>
<td><code>in</code> integer</td>
</tr>
</tbody>
</table>

**Postconditions (POST)**

{for all i \ (1 \leq i \leq n) \ implies (a(i) = old--a(n - i + 1))}
Specifying classes

• Invariant predicates and pre/post conditions for each method
• Example of invariant specifying an array implementing ADT set

for all i, j (1 ≤ i ≤ length and 1 ≤ j ≤ length and i ≠ j) implies IMPL[i] ≠ IMPL[j]
(invariant: no duplicates are stored)
Specifying non-terminating behaviors

- Example: producer+consumer+buffer
- **Invariant** specifies that whatever has been produced is the concatenation of what has been taken from the buffer and what is kept in the buffer

\[
\text{input\_sequence} = \text{append} (\text{output\_sequence}, \text{contents}(\text{CHAR\_BUFFER}))
\]
A case-study using logic specifications

• We outline the elevator example

• Elementary predicates
  – at (E, F, T)
    • Elevator E is at floor F at time T
  – start (E, F, T, up)
    • Elevator E left floor F at time T moving up

• Rules
  – (at (E, F, T) and on (EB, F₁, T) and F₁ > F) implies start (E, F, T, up)
States and events

• Elementary predicates are partitioned into:
  – **states**, having non-null duration
    – standing(E, F, T1, T2)
      » assumption: closed at left, open at right: [T1..T2]
  – **events**
    • instantaneous (causes state change occurs at the same time)
    • represented by predicates that hold only at a particular time instant
      – arrived (E, F, T)

• **For simplicity, we assume**
  • zero decision time
  • no simultaneous events
Events (1)

• **arrival** \((E, F, T)\)
  - \(E \in [1..n]\), \(F \in [1..m]\), \(T \geq t_0\), \((t_0 \text{ initial time})\)
    - does not say if it will stop or will proceed, nor where it comes from

• **departure**\((E, F, D, T)\)
  - \(E \in [1..n]\), \(F \in [1..m]\), \(D \in \{\text{up, down}\}\), \(T \geq t_0\)

• **stop** \((E, F, T)\)
  - \(E \in [1..n]\), \(F \in [1..m]\), \(T \geq t_0\)
    - specifies stop to serve an internal or external request
Events (2)

- **new_list** \((E, L, T)\)
  - \(E\) in \([1..n]\), \(L\) in \([1.. m]^*\), \(T \geq t_0\)
  - \(L\) is the list of floors to visit associated with elevator
    (scheduling is performed by the control component of the system)

- **call** \((F, D, T)\)
  - external call (with restriction for 1, N)

- **request** \((E, F, T)\)
  - internal reservation
States

- **moving** (E, F, D, T1, T2)
- **standing** (E, F, T1, T2)
- **list** (E, L, T1, T2)

  - We implicitly assume that state predicates hold for any sub-interval (i.e., the rules that describe this are assumed to be automatically added)
    - Nothing prevents that it holds for larger interval
Rules relating events and states

\[ R_1 : \text{When E arrives at floor } F, \text{ it continues to move if there is no request for service from } F \text{ and the list is empty. If the floor to serve is higher, it moves upward; otherwise it moves downward.} \]

\[
\text{arrival } (E, \ F, \ T_a ) \text{ and } \\
\text{list } (E, \ L, \ T, \ T_a ) \text{ and } \\
\text{first } (L) > F \\
\text{implies} \\
\text{departure } (E, \ F, \ \text{up}, \ T_a )
\]

A similar rule describes downward movement.
Rules relating events and states

$R_1$: When $E$ arrives at floor $F$, it continues to move if there is no request for service from $F$ and the list is not empty. If the floor to serve is higher, it moves upward; otherwise it moves downward.

\[
\text{arrival (E, F, T_a) and list (E, L, T, T_a) and first (L) > F implies departure (E, F, up, T_a)}
\]

A similar rule describes downward movement.
**R2**: Upon arrival at F, E stops if F must be serviced (F appears as first of the list)

arrival \((E, F, T_a)\) and  
list \((E, L, T, T_a)\) and  
first \((L) = F\)  
implies  
stop \((E, F, T_a)\)

**R3**: E stops at F if it gets there with an empty list

arrival \((E, F, T_a)\) and  
list \((E, \text{empty}, T, T_a)\)  
implies  
stop \((E, F, T_a)\)
R4: Assume that elevators have a fixed time to service a floor. If the list is not empty at the end of such interval, the elevator leaves the floor immediately.

\[
\text{stop (E, F, } T_a) \text{ and list (E, L, T, } T_a + Dt_s) \text{ and first (L) > F,}
\]

implies

\[
\text{departure (E, F, up, } T_a + Dt_s)
\]

R5: If the elevator has no floors to service, it stops until its list becomes nonempty.

\[
\text{stop (E, F, } T_a) \text{ and list (E, L, } T_p, T) \text{ and } T_p > T_a + Dt_s \text{ and list (E, empty, } T_a + Dt_s, T_p) \text{ and first (L) > F}
\]

implies

\[
\text{departure (E, F, up, } T_p)
\]
**R6**: Assume that the time to move from one floor to the next is known and fixed. The rule describes movement.

\[
\text{departure } (E, F, \text{up}, T) \\
\text{implies} \\
\text{arrival } (E, F + 1, T + Dt)
\]

**R7**: The event of stopping initiates standing for at least Dts.

\[
\text{stop } (E, F, T) \\
\text{implies} \\
\text{standing } (E, F, T, T + Dt_s)
\]
**R8**: At the end of the minimum stop interval $D_t$, $E$ remains standing if there are no floors to service.

$$\text{stop (}E, F, T_s\) \text{ and}$$

$$\text{list (}E, \text{ empty}, T_s + D_{t}, T)$$

implies

$$\text{standing (}E, F, T_s, T)$$

**R9**: Departure causes moving.

$$\text{departure (}E, F, D, T)$$

implies

$$\text{moving (}E, F, D, T, T + Dt)$$
Control rules

Express the scheduling strategy (by describing “new_list” events and “list” states)

Internal requests are inserted in the list from current floor to top if the elevator is moving up.

External calls are inserted in the list of the closest elevator that is moving in the correct direction, or in a standing elevator.
R10: Reserving F from inside E, which is not standing at F, causes immediate update of L according to previous policy

\[ \text{request} \left( E, F, T_R \right) \text{ and not } \left( \text{standing} \left( E, F, T_a, T_R \right) \right) \text{ and } \]
\[ \text{list} \left( E, L, T_a, T_R \right) \text{ and } LF = \text{insert}\_\text{in}\_\text{order}(L, F, E) \]
\[ \text{implies} \]
\[ \text{new}\_\text{list} \left( E, LF, T_R \right) \]
**R11:** Effect of arrival of E at floor F

\[
\text{arrival (E, F, T_a) and list (E, L, T, T_a) and } \\
F = \text{first (L) and } L_t = \text{tail (L) implies } \\
\text{new_list (E, L_t, T_a)}
\]

**R12:** How list changes

\[
\text{new_list (E, L, T_1) and not (new_list (E, L, T_2) and } \\
T_1 < T_2 < T_3) \quad \text{implies } \\
\text{list (E, L, T_1, T_3)}
\]
Verifying specifications

• The system can be simulated by providing a state (set of facts) and using rules to make deductions

  \textit{standing} (2, 3, 5, 7) \textit{elevator} 2 at floor 3 at least from instant 5 to 7
  
  \textit{list}(2, \textit{empty}, 5, 7)
  
  \textit{request}(2, 8, 7)
  
  \textit{new_list}(2, \{8\}, 7)

  \Rightarrow \text{(excluding other events)}
  
  \textit{departure} (2, \textit{up}, 7 + \textit{Dt}_s)
  
  \textit{arrival} (2, 8, 7 + \textit{Dt}_s + \textit{Dt}_a * (8-3))
Verifying specifications

- Properties can be stated and proved via deductions

\[
\text{new\_list (E, L, T) and } F \in L \implies \\
\text{new\_list (E, L_1, T_1) and } F \not\in L_1 \text{ and } T_1 > T
\]

(all requests are served eventually)
Formal Specification

Z Schema
Z Schema
by: Mike Woolbridge

http://www.cas.mcmaster.ca/~sartipi/course/se3km4/f07/slides/FORMAL/lect11-Z.pdf
Z Schema
Example: Order Invoicing

http://www.cas.mcmaster.ca/~sartipi/course/se3km4/f07/slides/FORMAL/Z-ch01-example.doc
Z Schema
Example: elevator

Specified in the following slides
Modularizing logic specifications: Z

- Z is a formal specification language based on the sets, functions, and first-order predicate logic
- The system entities are defined using types
- System is specified by describing state space, using Z schemas
- Properties of state space described by invariant predicates that are required to be maintained as the system moves from one state to another state
- Predicates written in first-order logic
- Operations define state transformations
Modularizing logic specifications: Z

- State transitions are described by providing the relationships between inputs and outputs of the operations, and predicates specifying the resulting state changes.

- Z schemas are modularization constructs used to define states and how they are affected by the operations.

- To simplify the problem:
  - Only one elevator is considered
  - No timing property is considered.
The elevator example in Z

\[
\text{SWITCH ::= on | off}
\]

\[
\text{MOVE ::= up | down}
\]

\[
\text{FLOORS : } \mathbb{N}
\]

\[
\text{FLOORS > 0}
\]

- **IntButtons**
  
  \[
  \text{IntReq : } 1 \ldots \text{FLOORS} \rightarrow \text{SWITCH}
  \]

- **FloorButtons**
  
  \[
  \text{ExtReq : } 1 \ldots \text{FLOORS} \rightarrow \exists \text{MOVE}
  \]

  \[
  \text{down } \notin \text{ ExtReq(1)}
  \]

  \[
  \text{up } \notin \text{ ExtReq(\text{FLOORS})}
  \]

- **Scheduler**
  
  \[
  \text{NextFloor ToServe : } 0 \ldots \text{FLOORS}
  \]

- **Elevator**
  
  \[
  \text{CurFloor : } 1 \ldots \text{FLOORS}
  \]

  \[
  \text{CurDirection : MOVE}
  \]
Complete state space

This invariant is weak, since the invariant is satisfied even if NextFloorToServe is 0 and there is a pending request for the elevator.
Complete state space

This schema resolves the drawback of the above schema, however, it is still too weak, since any floor for which a request is pending can be chosen as the next floor to serve.
Priorities in serving requests

Complex invariant is defined using existential predicate that deal separately with the cases where the elevator is moving up or down:

- **Priority 1** (if elevator is going up):
  - Serves floors higher than the current floor for which there are internal requests to stop or external request for up rides. The next floor to serve is the lowest floor towards up.

- **Priority 2** (if P1 is empty):
  - Serves to lower floors (i.e., int / ext request exist). The elevator goes down and the first to serve is the highest floor.

- **Priority 3** (if P2 is empty)
  - Serves to lowest floor whose external pending requests towards up.

- **P3 is empty**
  - There is no next floor to visit; elevator stays in the current floor.
Complete state space: final

\[ \exists \text{ Pri1, Pri2, Pri3 : } \mathbb{PN}_1 \]  

**System**

- Elevator
- IntButtons
- Floor Buttons
- Scheduler

\[ \exists \text{ Pri1, Pri2, Pri3 : } \mathbb{PN}_1 \]  

**CurDirection = up ⇒**  

\[ \begin{align*}
\text{Pri1} &= \{ f : 1 \ldots \text{FLOORS} \mid f \geq \text{CurFloor} \land (\text{IntReq}(f) = \text{on} \lor \text{up} \in (\text{ExtReq}(f))) \} \lor \\
\text{Pri2} &= \{ f : 1 \ldots \text{FLOORS} \mid \text{down} \in \text{ExtReq}(f) \lor (f < \text{CurFloor} \land \text{IntReq}(f) = \text{on}) \} \lor \\
\text{Pri3} &= \{ f : 1 \ldots \text{FLOORS} \mid f < \text{CurFloor} \land \text{up} \in \text{ExtReq}(f) \} \lor \\
((\text{Pri1} \neq 0 \land \text{NextFloorToServe} = \min(\text{Pri1})) \lor \\
(\text{Pri1} = 0 \land \text{Pri2} \neq 0 \land \text{NextFloorToServe} = \max(\text{Pri2})) \lor \\
(\text{Pri1} = 0 \land \text{Pri2} = 0 \land \text{Pri3} \neq 0 \land \text{NextFloorToServe} = \min(\text{Pri3})) \lor \\
(\text{Pri1} = 0 \land \text{Pri2} = 0 \land \text{Pri3} = 0 \land \text{NextFloorToServe} = 0)) \lor \\
\end{align*} \]

**CurDirection = down ⇒**  

\[ \begin{align*}
\text{Pri1} &= \{ f : 1 \ldots \text{FLOORS} \mid f \leq \text{CurFloor} \land \\
(\text{IntReq}(f) = \text{on} \lor \text{down} \in \text{ExtReq}(f)) \} \lor \\
\text{Pri2} &= \{ f : 1 \ldots \text{FLOORS} \mid \text{up} \in \text{ExtReq}(f) \lor \\
(f > \text{CurFloor} \land \text{IntReq}(f) = \text{on}) \} \lor \\
\text{Pri3} &= \{ f : 1 \ldots \text{FLOORS} \mid f > \text{CurFloor} \land \text{down} \in \text{ExtReq}(f) \} \lor \\
((\text{Pri1} \neq 0 \land \text{NextFloorToServe} = \max(\text{Pri1})) \lor \\
(\text{Pri1} = 0 \land \text{Pri2} \neq 0 \land \text{NextFloorToServe} = \min(\text{Pri2})) \lor \\
(\text{Pri1} = 0 \land \text{Pri2} = 0 \land \text{Pri3} \neq 0 \land \text{NextFloorToServe} = \max(\text{Pri3})) \lor \\
(\text{Pri1} = 0 \land \text{Pri2} = 0 \land \text{Pri3} = 0 \land \text{NextFloorToServe} = 0)) \lor \\
\end{align*} \]
### Operations

**MoveToNextFloor**

\(\Delta\text{System}\)

NextFloorToServe \(\neq\) 0  
CurFloor \(\neq\) NextFloorToServe  
CurFloor > NextFloorToServe  
CurFloor' = CurFloor - 1 ∧ CurDirection' = down  
CurFloor < NextFloorToServe  
CurFloor' = CurFloor + 1 ∧ CurDirection' = up  
ιIntButtons' = ιIntButtons  
ιFloorButtons' = ιFloorButtons

**InternalPush**

\(\Delta\text{System}\)

\(f? : 1 \ldots FLOORS\)

IntReq' = IntReq \(\oplus\) \{\(f? \mapsto\) on\}  
ιElevator' = ιElevator  
ιFloorButtons' = ιFloorButtons

**ExternalPush**

\(\Delta\text{System}\)

\(f? : 1 \ldots FLOORS\)

dir? : MOVE

ExtReq' = ExtReq \(\oplus\) \{(f? \mapsto (ExtReq (f?) \cup \{dir?\}))\}  
ιElevator' = ιElevator  
ιIntButtons' = ιIntButtons

**ServeIntRequest**

\(\Delta\text{System}\)

NextFloorToServe = CurFloor  
IntReq(CurFloor) = on  
IntReq' = IntReq \(\oplus\) \{(CurFloor \mapsto\) off\}  
ExtReq' = ExtReq  
CurFloor' = CurFloor  
CurDirection' = CurDirection
Operations (2)

ServeExtRequestSameDir

\[ \begin{align*}
\Delta \text{System} \\
\text{NextFloorToServe} &= \text{CurFloor} \\
\text{IntReq}(\text{CurFloor}) &= \text{off} \\
\text{CurDirection} &\in \text{ExtReq}(\text{CurFloor}) \\
\text{IntReq}' &= \text{IntReq} \\
\text{ExtReq}' &= \text{ExtReq} \oplus \{(\text{CurFloor} \rightarrow (\text{ExtReq}(\text{CurFloor}) \setminus \{\text{CurDirection}\}))\} \\
\text{CurFloor}' &= \text{CurFloor} \\
\text{CurDirection}' &= \text{CurDirection}
\end{align*} \]

ServeExtRequestOtherDir

\[ \begin{align*}
\Delta \text{System} \\
\text{NextFloorToServe} &= \text{CurFloor} \\
\text{IntReq}(\text{CurFloor}) &= \text{off} \\
\text{CurDirection} &\notin \text{ExtReq}(\text{CurFloor}) \\
\text{IntReq}' &= \text{IntReq} \\
\text{ExtReq}' &= \text{ExtReq} \oplus \{(\text{CurFloor} \rightarrow 0 )\} \\
\text{CurFloor}' &= \text{CurFloor} \\
\text{CurDirection}' &= \text{CurDirection}
\end{align*} \]

SystemInit

\[ \begin{align*}
\forall i : 1 \ldots \text{FLOORS} \cdot \text{IntReq}'(i) &= \text{off} \land \text{ExtReq}'(i) = 0 \\
\text{NextFloorToServe}' &= 0 \\
\text{CurFloor}' &= 1 \\
\text{CurDirection}' &= \text{up}
\end{align*} \]
Control Diagrams

State Transition Diagram
&
Statechart
Requirements for a notation

- Ability to support separation of concerns
  - e.g., separate functional specs from
    - performance specs
    - user-interface specs
    - ...
- Support different views
Example of views

document production

data flow view (1)
Control flow view (2)

1. Get user name
2. Search in Customers
3. Get other data from the data base
4. Get other relevant data from user interaction
5. Get appropriate text skeletons from predefined text library
6. Compose the document by choosing formatting options (this involves interaction with the user and access to the Formats data base)
7. Print document
Structured Design: Data Flow + Control Flow

Case study of Restaurant System using Statemate I-CASE Toolkit

http://www.cas.mcmaster.ca/~sartipi/course/se3km4/f07/slides/statechart.pdf
UML notations

• Class diagrams
  – describe static architecture in terms of classes and associations
  – dynamic evolution can be described via Statecharts (see later)

• Activity diagrams
  – describe sequential and parallel composition of method executions, and synchronization
An activity diagram
Sequential decomposition
-- chemical plant control example--

Pressure signal
Pressure action
Successful recovery
Normal
Successful recovery
Temperature signal
Temperature action
Unsuccessful recovery
Off
Unsuccessful recovery

Normal
RecoverySuccess
AnomalyDetection
Recovery
Press
Temperature
Identification
Done
Done
Pressure Action
Temperature Action
RecoveryFailure
Parallel decomposition
Object state diagram using Statecharts

Empty

NotEmpty

Push(item)

Pop[stack contains more than 1 item]

Pop[stack contains 1 item]

Top

Push(item)
Specifications for the end-user

- Specs should be used as common reference for producer and user
- They help removing ambiguity, incompleteness, ...
- Can they be understood by end-user?
  - They can be the starting point for a prototype
  - They can support some form of animation (e.g., see Petri nets)
Conclusions (1)

• Specifications describe
  – what the users need from a system (requirements specification)
  – the design of a software system (design and architecture specification)
  – the features offered by a system (functional specification)
  – the performance characteristics of a system (performance specification)
  – the external behavior of a module (module interface specification)
  – the internal structure of a module (internal structural specification)
Conclusions (2)

• Descriptions are given via suitable notations
  – There is no “ideal” notation
• They must be modular
• They support communication and interaction between designers and users