Mathematical-Based Specifications

- Queueing and Simulation Models
  - Predict and Simulate System Behavior
- Declarative Specifications:
  - Prolog and Logic Specifications
  - Logic Specifications with Z
  - Algebraic Specifications and Standard ML
  - CSE233 Programming Languages
- Writing Specifications
  - Requirements for Specification Notation
  - Building Modular Specs
  - What’s in a Spec?
- Concluding Remarks
Queueing and Simulation Techniques

- Queueing Models
  - Mathematical Formulations of Predicted System Behavior
  - Gross to Detailed Behavior
- Simulation:
  - Computerized Scenarios of Diverse Application Requirements
  - Languages Including: GPSS, Simscript, etc.
- Allow Evaluative Refinement of Specification Requirements at Earliest Stages of the Process
- Geared Towards Identifying Potential Bottlenecks, Underutilized (or Unused) Components, etc.

A Basic Queueing Model

- $r$: Probability of Job’s I/O Wait Time
- $1-r$: Probability of Job’s CPU Wait Time
- $P_i$: Probability for Waiting for $i$-th Device with Sum $P_i$’s for 1 to $n$ Equal to 1
An I/O Sub-Model of Disk System

- \( L \): Mean Arrival Rate
- Mean Bulk Size \( T/n \)
- Variance of Bulk Size: \( T/n (1-1/n) \)
- Mean Bulk Size at Each Module \( T/Mn \)

Simulation Model
Simulation Program - Simscript

PREAMBLE
\ ' THIS IS A SIMSCRIPT MODEL OF A SINGLE SERVER QUEUE. IN THIS MODEL, THERE \ ' IS ONLY ONE PROCESS CALLED CUSTOMER WHO INITIATES THE PROCESSING
NORMALLY MODE IS UNDEFINED
PROCESSES INCLUDE CUSTOMER
RESOURCES INCLUDE SERVER
ACCUMULATE UTIL.SERVER AS THE AVERAGE OF N.X.SERVER
ACCUMULATE AVE.QUEUE.LENGTH AS THE AVERAGE OF N.Q.SERVER
ACCUMULATE MAX.QUEUE.LENGTH AS THE MAXIMUM OF N.Q.SERVER
DEFINE NO.OF.ARRIVALS AND MAX.NO.ARRIVALS AS INTEGER VARIABLES
END

MAIN
PRINT 1 LINE THUS
INPUT NUMBER MAXIMUM NUMBER OF ARRIVALS
READ MAX.NO.ARRIVALS
CREATE EVERY SERVER(1)
LET U.SERVERER(1)=1
ACTIVATE A CUSTOMER IN EXPONENTIAL.F(10.0,1) MINUTES
START SIMULATION
PRINT 4 LINES WITH TIME.V*24*60, UTIL.SERVER(1),
AVE.QUERY.LENGTH(1) AND MAX.QUERY.LENGTH(1) THUS
CLOCK TIME = ********.** MINUTES
THE UTILIZATION OF THE SEVER WAS ****.**
THE AVERAGE LENGTH OF THE QUEUE WAS ****.**
THE MAXIMUM LENGTH OF THE QUEUE WAS *****
END

PROCESS CUSTOMER
ADD 1 TO NO.OF.ARRIVALS
IF NO.OF.ARRIVALS < MAX.NO.ARRIVALS
ACTIVATE A CUSTOMER IN EXPONENTIAL.F(10.0,1) MINUTES
ALWAYS
REQUEST 1 UNIT OF SERVER()
WORK EXPONENTIAL.F(8.0,2) MINUTES
RELINQUISH 1 UNIT OF SERVER(1)
END

EXECUTION/OUTPUT OF SIMULATION PROGRAM:
INPUT NUMBER MAXIMUM NUMBER OF ARRIVALS
II.5> 100

CLOCK TIME = 100008.36 MINUTES
THE UTILIZATION OF THE SEVER WAS .80
THE AVERAGE LENGTH OF THE QUEUE WAS 2.65
THE MAXIMUM LENGTH OF THE QUEUE WAS 12
Logic Specifications

- Logic Specifications Employ Mathematical Formalisms and First-Order Theory (FOT)
  - Variables and Numeric Constants
  - Predicates and Functions
  - Logical Connectors (and, or, not, implies, =)
  - Formula May Return Boolean Values (T/F)
  - May also Employ There Exists and For All
- Logical Specifications:
  - Provide Precise Means to Specify System Behavior
  - Not All Systems Can Utilize Them
- Two Pronged Presentation:
  - The Prolog Programming Language
  - Logic Specification Techniques

The Prolog Programming Language

- Prolog is a Logic Programming Language for Users to Define Facts and Inquire About them
- First Step is to Define a Relation of Facts as:
  ```prolog
  parent(john,jack).
  parent(john,jill).
  parent(peter,john).
  parent(vicky,sandi).
  parent(jim,justin).
  parent(vicky,sandi).
  parent(jim,justin).
  parent(steve,stevejr).
  parent(lois,stevejr).
  parent(steve,charlie).
  parent(lois,charlie).
  ```
- Store these Facts in File: parentinfo.pl
- File is Then Loaded into Prolog Execution Environment
- Queries Against this Environment can be Made
- Examples for this Material done Using:
  http://www.swi-prolog.org/
Selected File-Consult to Load parentinfo.pl

Note Subsequent Assertions:

- parent(steve,john) and parent(steve,stevejr)

What do the Following Queries Do?
Possible to Define Grandparent Relation as Below:

\[
\text{grandparent}(G,C) :- \text{parent}(G,P), \text{parent}(P,C)
\]

This was added to parentinfo.pl

For help, use `?- help(Topic), or `- apropos(Word).

```
1  ?- grandparent(vicky,justin).
   Yes
2  ?- grandparent(X,Y).
   X = peter
   Y = jack ;
   X = peter
   Y = jill ;
   X = vicky
   Y = steve ;
   X = vicky
   Y = charlie ;
   X = vicky
   Y = justin ;
   No
3  ?-
```

A Second Example with PLs

Define the Links that Historically Exist Among PLs

First Step is to Define a Relation of Facts as:

```prolog
link(fortran,algol60).
link(algol60,cpl).
link(cpl,bcp1).
link(bcp1,c).
link(c,cpp).
link(algol60,simula67).
link(simula67,smalltalk80).
link(simula67,cpp).
link(cpp,java).
link(smalltalk80,modula).
path(L,L).
path(L,M) :- link(L,K), path(K,M).
```

Store these Facts in File: links.pl

What Does Path Define?
Starting the Links Example

**How Does Path Rule Work?**

Path is a Recursive Rule

**TERMINATION CONDITION:**  path(L,L).

**RECURSION:**  path(L,M) :- link(L,K), path(K,M).
Two Languages with Links to Third?

X
/    /   \
Y    Z

CH5.17

What Does Path(A,B) Return?

A = fortran

B = smalltalk

X = algol60

Y = simula67

Z = algol60

CH5.18
Logic Specification

- Examples of FOT Formulae:
  - $x > y$ and $y > z$ implies $x > z$
  - $x = y$ implies $y = x$
  - $x + 1 < x - 1$
  - for all $x$ (exists $y$ ($y = x + z$))
- Free Variables: Not Quantified - $x > 3$ or $x < -6$
- Bound Variables: exists $x$ ($x < 3$)
- Closed Formula: All Variables Quantified
  - for all $x, y, z$ ($x > y$ and $y > z$ implies $x > z$)
  - Always True or False
  - Quantified by Binding Free Variables

Logic Specification

- Logic Can be Used to Express Contraints
  - Class Enrollment Between 5 and Max
    - for all $a$ in Class
      - $5 \leq$ cardinality $\{b|<a,b> \in ENROLLED\_IN\} \leq a.MAXENROLLMENT$
  - Cardinality Function Indicates No. Elements in Set
- How Do we Transition from Logic to Programming?
  - Input-Output Assertions
    - Logical Formulas Relating to a Program (Procedure or Function) Input and Output
  - Intermediate Assertions
    - Logical Formulas for Program Fragments
    - Statements About Program’s Execution at Various Points
    - Quantifies What When under Which Conditions
Logic Specifications for Programs

Let $P$ be a Sequential Program
- $<i_1, i_2, ..., i_n>$ $P$'s Input Values
- $<o_1, o_2, ..., o_m>$ $P$'s Output Values

Track Conditions of Execution
- Precondition: Conditions Satisfied before Program (Procedure/Function/Statement) Executes
- Postcondition: Conditions Satisfied after Program (Procedure/Function/Statement) Executes

Thus:
Pre: $\{<i_1, i_2, ..., i_n>\}$
$P$
Post: $\{<o_1, o_2, ..., o_m, i_1, i_2, ..., i_n>\}$

Examples

Pre: $\{A$ is an Array with Values$\}$
$P$
Post: $\{A$ is a Sorted Array$\}$

Pre: $\{\exists x (i_1 = z \cdot i_2)$
$P$
Post: $\{o_1 = \frac{i_1}{i_2}\}$

Pre: $\{true\}$
$P$
Post: $\{(o_1 = i_1 \text{ or } o_1 = i_2) \text{ and } o_1 \geq i_1 \text{ and } o_1 \geq i_2\}$

What Does $o_1$ Contain?

Compute Greatest Common Divisor:
Pre: $\{i_1 > 0$ and $i_2 > 0\}$
$P$
Post: $\{($exists $z_1, z_2 (i_1 = o_1 \cdot z_1$ and $i_2 = o_1 \cdot z_2)$ and not ($exists h$ $\{($exists $z_1, z_2 (i_1 = h \cdot z_1$ and $i_2 = h \cdot z_2)$ and $h > 0)\})\}$
Examples of Procedures

A Search for Element in Array Procedure:
{\(n > 0\)} -- \(n\) is a constant value

procedure search (table: in integer_array;
                 n: in integer;
                 element: in integer;
                 found: out Boolean);
{\(\text{found } \equiv \text{exists } i(1 \leq i \leq n \text{ and } \text{table}(i) = \text{element})\)}

A Reverse an Array Procedure:
{\(n > 0\)} -- \(n\) is a constant value

procedure reverse (a: in out integer_array;
                   n: in integer);
{\(\text{for all } i(1 \leq i \leq n \text{ implies } a(i) = \text{old} - a(n-i+1))\)}

Pre/Post Conditions with Invariants:
{\(\text{for all } i, j (1 \leq i \leq \text{length and } 1 \leq j \leq \text{length and } i \neq j)\)}
implies \(\text{IMPL}[i] \neq \text{IMPL}[j]\)
(no duplicates are stored - implements a set)

Case Study: Elevator Example

Logic Notation Can be Utilized to Model the Elevator
Example Briefly Seen with Petri Nets

First, we Define:
- \(E\) – Elevator; \(F\) – Floor; \(T\) – Time;
- \(EB\) – Elevator Button

Next, we Define Predicates:
- \((E, F, T)\) – \(E\) is at Floor \(F\) at Time \(T\)
- \((E, F, T, \text{up})\) – \(E\) left Floor \(F\) at Time \(T\) Moving Up
- \((E, F, T, \text{down})\) – similar but Moving Down
- \((EB, F, T)\) – \(EB\) Pushed for Floor \(F\) at Time \(T\)

Also, we Need to have Rules, for example:
- \((\text{at } (E, F, T) \text{ and on } (EB, F_1, T) \text{ and } F_1 > F) \text{ Implies Start } (E, F, T, \text{up})\)
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
- Events
  - Instantaneous (Caused State Change occurs at same Time)
  - Represented by Predicates that hold only at a Particular Time Instant --- arrived (E, F, T)

Assumptions:
- Zero Time to make a Decision
- No Simultaneous Events or States Allowed

States and Events (Informally)

States
- moving (E, F, D, T₁, T₂) – D can also be U
- standing (E, F, T₁, T₂)
- list (E, L, T₁, T₂) – of floors for time interval

Events
- arrival (E, F, T)
- departure (E, F, D, T) – D can also be U
- stop (E, F, T)
- new_list (E, L, T) – floors where E will stop
- call (F, D, T)
- request (E, F, T)
### Events - Formally

- **arrival** $(E, F, T)$
  - $E$ in $[1..n]$, $F$ in $[1..m]$, $T \geq t_0$, ($t_0$ 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 {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

- **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

### 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 empty. If the floor to serve is higher, it moves upward; otherwise it moves downward.

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

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, 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.

stop (E, F, T_a) and
list (E, L, T, T_a + Dts) and
first (L) > F,
implies
departure (E, F, up, T_a + Dts)
R5: If the elevator has no floors to service, it stops until its list becomes nonempty.

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

R6: Assume that the time to move from one floor to the next is known and fixed. The rule describes movement.

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

R7: The event of stopping initiates standing for at least \( Dts \).

\[ \text{stop (} E, F, T \text{) implies}
\]
\[ \text{standing (} E, F, T, T + Dts \text{)} \]
Rules Relating Events and States

R8: At the end of the minimum stop interval Dts, E remains standing if there are no floors to service.

\[
\text{stop (E, F, T_s) and} \\
\text{list (E, empty, T_s + Dt, T)} \\
\text{implies} \\
\text{standing (E, F, T, T)}
\]

R9: Departure causes moving.

\[
\text{departure (E, F, D, T)} \\
\text{implies} \\
\text{moving (E, F, D, T, T + Dt)}
\]

Control Rules

Express the Scheduling Strategy by Describing “new_list” Events and “list” States

Button calls are Treated Differently:

- 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 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

\[
\text{request (E, F, T_r) and not (standing (E, F, T, T_r)) and} \\
\text{list (E, L, T, T_r) and LF = insert_in_order(L, F, E)} \\
\text{implies} \\
\text{new_list (E, LF, T_r)}
\]
Remaining Control Rules

R11: Effect of arrival of E at floor F

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

R12: How list changes

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

Logic Specifications with Z

- Z Has Been Used to Specify Software Architectures
  - Formal Basis for Describing System
  - Mathematically Sound
- Using Z Schemas:
  - Systems are Specified by Describing State Space
  - Properties of State Space Described by Invariant Predicates
  - Predicates written in First-Order Logic
- Operations Define State Transformations
The Elevator Example in Z

\[ \begin{align*}
SWITCH & ::= \text{on} \mid \text{off} \\
MOVE & ::= \text{up} \mid \text{down} \\
FLOORS & : \mathbb{N} \\
FLOORS & > 0
\end{align*} \]

\[ \begin{align*}
\text{IntButtons} & \\
\text{IntReq} : 1 \ldots \text{FLOORS} & \to \text{SWITCH} \\
\text{FloorButtons} & \\
\text{ExtReq} : 1 \ldots \text{FLOORS} & \to \mathbb{P} \text{ MOVE} \\
\text{down} & \notin \text{ExtReq}(1) \\
\text{up} & \notin \text{ExtReq}(\text{FLOORS})
\end{align*} \]

\[ \begin{align*}
\text{Scheduler} & \\
\text{NextFloorToServe} & : 0 \ldots \text{FLOORS}
\end{align*} \]

\[ \begin{align*}
\text{Elevator} & \\
\text{CurFloor} & : 1 \ldots \text{FLOORS} \\
\text{CurDirection} & : \text{MOVE}
\end{align*} \]

A First Look at System Schema

\[ \begin{align*}
\text{System} & \\
\text{Elevator} & \\
\text{IntButtons} & \\
\text{FloorButtons} & \\
\text{Scheduler} & \\
\text{NextFloorToServe} \neq 0 & \Rightarrow \\
\text{IntReq}(\text{NextFloorToServe}) & = \text{on} \lor \text{ExtReq}(\text{NextFloorToServe}) \neq \emptyset \\
\text{NextFloorToServe} = 0 & \Rightarrow \\
(\forall f : 1 \ldots \text{FLOORS} \bullet (\text{IntReq}(f) = \text{off} \land \text{ExtReq}(f) = \emptyset))
\end{align*} \]
The Complete System Schema

System
Elevator
InitButtons
Floor Buttons
Scheduler

\[ \begin{align*}
2 \text{ Pri}_{1}, \text{ Pri}_{2}, \text{ Pri}_{3} : \mathbb{P} \mathbb{N}_{2} \rightarrow & \quad \\
\text{CurDirection} = \text{up} \Rightarrow & \quad \\
\{ \text{Pri}_{1} = \{ f : 1 \ldots \text{Floors} \mid f \geq \text{CurFloor} \land (\text{InitReq}[f] = \text{on} \lor \text{up} \in \text{EstReq}(f)) \} \land & \\
\text{Pri}_{2} = \{ f : 1 \ldots \text{Floors} \mid \text{down} \in \text{EstReq}(f) \lor (f < \text{CurFloor} \land \text{InitReq}[f] = \text{on}) \} \land & \\
\text{Pri}_{3} = \{ f : 1 \ldots \text{Floors} \mid f < \text{CurFloor} \land \text{up} \in \text{EstReq}(f) \} \land & \\
((\text{Pri}_{1} \neq \emptyset \land \text{NextFloorToServe} = \text{min}(\text{Pri}_{1})) \lor & \\
(\text{Pri}_{1} = \emptyset \land \text{Pri}_{2} \neq \emptyset \land \text{NextFloorToServe} = \text{max}(\text{Pri}_{2}) ) \lor & \\
(\text{Pri}_{1} = \emptyset \land \text{Pri}_{2} = \emptyset \land \text{Pri}_{3} \neq \emptyset \land \text{NextFloorToServe} = \text{min}(\text{Pri}_{3}) ) \lor & \\
(\text{Pri}_{1} = \emptyset \land \text{Pri}_{2} = \emptyset \land \text{Pri}_{3} = \emptyset \land \text{NextFloorToServe} = 0)) \} \land \\
\text{CurDirection} = \text{down} \Rightarrow & \\
\{ \text{Pri}_{1} = \{ f : 1 \ldots \text{Floors} \mid f \leq \text{CurFloor} \land & \\
(\text{InitReq}[f] = \text{on} \lor \text{down} \in \text{EstReq}(f)) \} \land & \\
\text{Pri}_{2} = \{ f : 1 \ldots \text{Floors} \mid \text{up} \in \text{EstReq}(f) \} \land & \\
(\text{ Pri}_{1} \neq \emptyset \land \text{NextFloorToServe} = \text{max}(\text{Pri}_{1}) ) \lor & \\
(\text{Pri}_{1} = \emptyset \land \text{Pri}_{2} \neq \emptyset \land \text{NextFloorToServe} = \text{min}(\text{Pri}_{2}) ) \lor & \\
(\text{Pri}_{1} = \emptyset \land \text{Pri}_{2} = \emptyset \land \text{Pri}_{3} \neq \emptyset \land \text{NextFloorToServe} = \text{max}(\text{Pri}_{3}) ) \lor & \\
(\text{Pri}_{1} = \emptyset \land \text{Pri}_{2} = \emptyset \land \text{Pri}_{3} = \emptyset \land \text{NextFloorToServe} = 0)) \} \\
\end{align*} \]

Supporting Operations

MoveToNextFloor
\[ \Delta \text{System} \]
\[ \text{NextFloorToServe} \neq \emptyset \]
\[ \text{CurFloor} \neq \text{NextFloorToServe} \]
\[ \text{CurFloor} > \text{NextFloorToServe} \rightarrow \]
\[ \text{CurFloor}' = \text{CurFloor} + 1 \land \text{CurDirection}' = \text{up} \]
\[ \text{CurFloor} < \text{NextFloorToServe} \rightarrow \]
\[ \text{CurFloor}' = \text{CurFloor} - 1 \land \text{CurDirection}' = \text{down} \]
\[ \text{InitButtons}' = \text{InitButtons} \]
\[ \text{FloorButtons}' = \text{FloorButtons} \]
\[ \text{ExternalPush} \]
\[ \Delta \text{System} \]
\[ f' : 1 \ldots \text{Floors} \]
\[ \text{dir}' : \text{MOVE} \]
\[ \text{InitReq}' = \text{InitReq} \sqcup \{ f' \rightarrow \text{on} \} \]
\[ \text{Elevator}' = \text{Elevator} \]
\[ \text{FloorButtons}' = \text{FloorButtons} \]

ServeInRequest
\[ \Delta \text{System} \]
\[ \text{NextFloorToServe} = \text{CurFloor} \]
\[ \text{InitReq}(\text{CurFloor}') = \text{on} \]
\[ \text{InitReq} = \text{InitReq} \sqcup \{ (\text{CurFloor} \rightarrow \text{off}) \} \]
\[ \text{ExtReq}' = \text{ExtReq} \]
\[ \text{CurFloor}' = \text{CurFloor} \]
\[ \text{CurDirection}' = \text{CurDirection} \]
Supporting Operations

Serve ExtRequestSameDir

\[ \text{System} \]

NextFloorToServe = CurFloor
IntReq(CurFloor) = off
CurDirection \notin ExtReq(CurFloor)
IntReq' = ExtReq
ExtReq' = ExtReq \oplus \{\{CurFloor \leftrightarrow ExtReq(CurFloor) \setminus \{CurDirection}\}\}
CurFloor' = CurFloor
CurDirection' = CurDirection

Serve ExtRequestOtherDir

\[ \text{System} \]

NextFloorToServe = CurFloor
IntReq(CurFloor) = off
CurDirection \notin ExtReq(CurFloor)
IntReq' = ExtReq
ExtReq' = ExtReq \oplus \{\{CurFloor \leftrightarrow \emptyset\}\}
CurFloor' = CurFloor
CurDirection' = CurDirection

Usage of Z in Specification Process

- Work of Prof. María Cecilia Bastarrica, Dept. of Computer Science, Univ. of Chile
  - UConn Ph.D. Student (Graduated 1999)
  - Co-Advisors: S. Demurjian and A. Shvartsman
- A Distributed Application is a Set of Components Deployed Over a Network That Communicate.
- Components May Be:
  - Data Types
  - Executable Programs
    - Legacy or COTS
    - Clients/Servers
  - Databases, Etc.
Network

- A Network is a Set of Computers Connected So They Can Communicate
- Computers & Connections May Have Different Characteristics That Affect Their Usage
  - Speed
  - Storage
  - Bandwidth

Best Deployment

- A Distributed System is Optimally Deployed if it Yields the Best Performance
- Performance: Efficient Use of Resources
  - Throughput
  - Response Time
  - Number of Messages
The Complete Cycle

- **Graphical Specification**
  - Transformation Procedures

- **Textual Specification**
  - Usage Patterns Information

- **Extended Textual Specification**
  - BIP Model

- **Optimal Deployment**

The Five Levels of $I^5$

- **Interface (I1)** - Types of Components, Nodes and Connectors
- **Implementation (I2)** - Classes of Components, Nodes and Connectors
- **Integration (I3)** - Dependencies Between Component and Node Classes
- **Instantiation (I4)** - Instances of Each Class Definition
- **Installation (I5)** - Deployment of Each Instance (Requirements and Complete Deployment)
Dependencies Between Levels

Component Types  
Node Types

Component Classes  
Node Classes

Implementation Dependencies

Inst. Components  
Inst. Nodes

System Instantiation

Installation Req.  
Installation Req. (fix location)

(together, separated)

Complete Installation

Modeling a System in $I^5$

- Conceptually Based on an Essentially Serializable Data Service (ESDS)
- Demonstration of
  - Modeling Capabilities of $I^5$
  - Verification of Work Utilizing Genetic Algorithm
  - Joint Research with Bastarrica and Cabellero
  - Published in Software Engineering/Knowledge Engineering Conference 01
  - See Course web page for paper
**What is ESDS?**

- ESDS is a Replicated Data Service Dealing with
  - Replicated Objects Allowing clients of Service to Relax Consistency Requirements for Improved Responsiveness
  - Simultaneously Provide Guarantees of Eventual Consistency of Replicated Data
- ESDS Implementation Includes
  - Parameterizable Number of Client, Front-End, and Replicated Components
  - Each can have Different Communication Patterns
  - Each can be Individually Deployed in a Target Network

**Modeling ESDS using \( f^5 \)**

- ESDS Deals with Replicated Objects and Trades Short-Term Consistency (lack of) for Performance
- ESDS Guarantees Eventual Consistency of Replicas
- In ESDS Replicas
  - Keep Order of Known Data Operations
  - Implemented as Individually Deployable Components
  - Periodically Exchange Knowledge in Gossip Messages
- Clients Interact with Users of the Service
- Front-Ends Broker Communication Between Clients and Replicas
Software Interface: I1S

comp_types = \{Client, FrontEnd, Replica\}

Components Types
- Types/Supertypes
- Associated Interfaces
- Calls

Properties
- Types are Unique
- Supertypes Part of I1S
- Calls Satisfied in I1S

[COMP_TYPE][INTERFACE]

```
Comp_Type
  type : COMP_TYPE
  supertypes : P Comp_Type
  local_int, interfaces : P INTERFACE
  local_calls, calls : Comp_Type ↔ INTERFACE

self \notin supertypes
interfaces = supertypes.interfaces \cup local_int
calls = supertypes.calls \cup local_calls
```
**Hardware Interface: I1H**

- **Node Types**: {Sun, Intel Pentium}
- **Connector Types**: {MPI, Sockets}
- **Connections**: 
  - (MPI ⊢ {Sun, Sun})
  - (Sockets ⊢ {Sun, Intel Pentium})

---

**Interface - Hardware: I1H**

- **Node Types**
  - Properties: Node Types Must Be Connected
  - Defined Node and Connector Types Take Part in Connections

**[NODE_TYPE]**

- **Node_Type**
  - type: NODE_TYPE

**[CONN_TYPE]**

- **Conn_Type**
  - type: CONN_TYPE

**I1_H**

- node_types : P Node_Type
- conn_types : P Conn_Type
- connects : Conn_Type ↔ P Node_Type

(dom connects) = conn_types

∀ n_set ∈ (ran connects) • n_set ⊆ node_types

∀ nt ∈ node_types • ∃ (ct ⊩ nt_set) ∈ connects | nt ∈ nt_set
Software Implementation :I2S

c_classes = {PCtrCl, XCtrCl, XFrontEnd, Counter}

Hardware Implementation: I2H

node_classes = {SUN OS 4.1.4, Win95}
conn_classes = {MPI_Impl, CSockets}
connects_cl = {(MPI_Impl ∪ {SUN OS 4.1.4, SUN OS 4.1.4}),
               (CSockets ∪ {SUN OS 4.1.4, Win95})}
Integration: I3

supports = \{(\text{Counter} \land \text{Sun OS 4.1.4}), (\text{XFrontEnd} \land \text{Sun OS 4.1.4}), (\text{XCtrCl} \land \text{Sun OS 4.1.4}), (\text{PCCtrCl} \land \text{Win95})\}

Software Instantiation: I4S

i\_comp = \{c1:\text{PCCtrCl}, c2:\text{PCCtrCl}, c3:\text{PCCtrCl}, c4:\text{PCCtrCl}, fe1:\text{XFrontEnd}, fe2:\text{XFrontEnd}, ct1:\text{Counter}, ct2:\text{Counter}, ct3:\text{Counter}, ct4:\text{Counter}, ct5:\text{Counter}, ct6:\text{Counter}, \}
nodes = {pc1: WIn95, pc2: WIn95, pc3: WIn95, pc4: WIn95, sun1: SunOS4.1.4, sun2: SunOS4.1.4, sun3: SunOS4.1.4, sun4: SunOS4.1.4, sun5: SunOS4.1.4, sun6: SunOS4.1.4, sun7: SunOS4.1.4, sun8: SunOS4.1.4, sun9: SunOS4.1.4, sun10: SunOS4.1.4 }
conns = {mpi, sock1, sock2, sock3, sock4}

separated = {ct1: Counter, ct2: Counter, ct3: Counter, ct4: Counter, ct5: Counter, ct6: Counter}
Algebraic Specifications

- Algebra is an Underlying Mathematical Formulatism
- Heterogeneous Algebra:
  - Collection of Different Sets on Which Operations are Defined
  - Represents Theory that Underlies ADTs
- Example Homogeneous Algebra
  - Integer – Data Set
  - Add, Subtract, Multiple, Divide – Operations
- Example Heterogeneous Algebra
  - Strong
  - Concat, Length, Search, Compare, Copy, …
Algebraic Specifications

- Algebra Consists of:
  - Signature – Collection of Sets that Form the Algebra
  - Operations – Allow the Set to be Managed
- Syntax:
  - Signature + Operations
- Semantics:
  - Essential Properties that Must Always be True When Operations are Applied
  - Axioms or Equations
- Defining Algebras:
  - In Concept, Building our own ADTs

Semi-Formal Example

- A system for strings, with operations for
  - Creating new, empty strings (operation **new**)
  - Concatenating strings (operation **append**)
  - Adding a new character at the end of a string (operation **add**)
  - Checking the length of a given string (operation **length**)
  - Checking whether a string is empty (operation **isEmpty**)
  - Checking whether two strings are equal (operation **equal**)
Consider the Algebra Below:

- Note the Definition of the Data (Sorts)
- Operations are Functions with Domains/Ranges

```plaintext
algebra StringSpec;
  introduces
  sorts String, Char, Nat, Bool;
  operations
    new: () → String;
    append: String, String → String;
    add: String, Char → String;
    length: String → Nat;
    isEmpty: String → Bool;
    equal: String, String → Bool
end StringSpec.
```

Properties Track State of the Algebra

constrains new, append, add, length, isEmpty, equal so that
for all $s, s1, s2: String; c: Char$

- isEmpty (new ()) = true;
- isEmpty (add (s, c)) = false;
- length (new ()) = 0;
- length (add (s, c)) = length (s) + 1;
- append (s, new ()) = s;
- append (s1, add (s2, c)) = add (append (s1, s2), c);
- equal (new (), new ()) = true;
- equal (new (), add (s, c)) = false;
- equal (add (s, c), new ()) = false;
- equal (add (s1, c), add (s2, c)) = equal (s1, s2);

end StringSpec.
Second Example – An Editor

- **newF**
  - creates a new, empty file

- **isEmptyF**
  - states whether a file is empty

- **addF**
  - adds a string of characters to the end of a file

- **insertF**
  - inserts a string at a given position of a file (the rest of the file will be rewritten just after the inserted string)

- **appendF**
  - concatenates two files

An Editor – Syntax

```
algebra TextEditor;
Introduces
  sorts Text, String, Char, Bool, Nat;
operations
  newF: () → Text;
  isEmptyF: Text → Bool;
  addF: Text, String → Text;
  insertF: Text, Nat, String → Text;
  appendF: Text, Text → Text;
  deleteF: Text → Text;
  lengthF : Text → Nat;
  equalF : Text, Text → Bool;
  addFC: Text, Char → Text;
  {This is an auxiliary operation that will be needed to define addF and other operations on files.}
```
An Editor – Properties

constrains newF, isEmptyF, addF, appendF, insertF, deleteF so that TextEditor generated by [newF, addFC]
for all [f, f1, f2: Text; s: String; c: Char; cursor: Nat]

isEmptyF (newF ()) = true;
isEmptyF (addFC (f, c)) = false;
addF (f, newS ()) = f;
addF (f, addS (s, c)) = addFC (addF (f, s), c);
lengthF (newF ()) = 0;
lengthF (addFC (f, c)) = lengthF (f) + 1;
appendF (f, newF ()) = f;
appendF (f1, addFC (f2, c)) =
    addFC (appendF (f1, f2), c);
equalF (newF (), newF ()) = true;
equalF (newF (), addFC (f, c)) = false;
equalF (addFC (f, c), new ()) = false;
equalF (addFC (f1, c1), addFC (f2, c2)) =
equalF (f1, f2) and equalC (c1, c2);
insertF (f, cursor, newS ()) = f;
((equalF (f, appendF (f1, f2)) and (lengthF (f1) = cursor - 1))
implies
    equalF (insertF (f, cursor, s), appendF (addF (f1, s), f2))) = true;
end TextEditor.

The ML Programming Language

- ML (Like Lisp) is a Functional Programming Language
  - Structure: Similar to Imperative Structure
  - Signature: Information on Structure at Compile Time
- ML is a Very Strongly Typed Programming Language
  - Strict Limits on What can be Defined
  - Compilation Very Difficult
  - If Compile – Most Likely Execute w/o Errors
- Unlike Lisp, ML has Rich Type Structure
  - Again – Akin to Defining ADTs
Examples in ML

- Structure Defines the Private Implementation
- Signature Represents the Public Interface
- What Does S define Below via Function f?

```ml
structure S =
  struct
    val y = z
    fun f(x) = if x = 0 then 1 else x * f(x - 1)
  end;

signature S =
  sig
    val y: int
    val f: int -> int
  end;
```

Examples in ML

- What Does the Following Represent?

```ml
signature L =
  sig
    type 'a list
    val Nil: 'a list
    val Cons: 'a x 'a list -> 'a list
    val Append: 'a list x 'a list -> 'a list
  end;

structure L =
  struct
    datatype 'a list = Nil | Cons of 'a x 'a list
    fun Append(x, Nil) = x
    (Append(x, Cons(h, t)) = Cons(h, Append(x, t))
  end;
```
**Requirements of Specification Notation**

- All SWE Principles Must be Applied to the Construction of a Specification
- Rigor and Formality as Discussed in Chapter 3
- Separation of Concerns
  - Functional vs. Performance vs. GUI vs. DB vs. …
  - Different Notations for Different Parts of System
    - DFD, ER, UML Activity, Statechart, Collaboration, etc.
    - Different Stakeholders See Design for their Perspective
- Incrementality
  - Not All SW Requirements Understood at Start
  - Apply to Construction of Specification
  - Add Details and Functionality Over Time
  - Specification Evolves Like a Design

---

**Recall Different Views – Data Flow**

- Predefined Text skeletons
- Predefined Formats
- Formatting options
- Document production
- Customer data (name, type of document)
- Print Document
- User
- Customers
- Formatting options

---
Recall Different Views – Control Flow

Get user name

Search in Customers

Get other data from the database

Get other relevant data from user interaction

Get appropriate text skeletons from predefined text library

Compose the document by choosing formatting options (this involves interaction with the user and access to the Formats data base)

Print document

Building Modular Specifications

- Specification Document is Combination of Multiple Formal and Semi-Formal Notations
  - Bound Together with Prose
  - Organized into Sections
- Fully Formal Specs Needed for Critical Applications
  - Formalisms Not Understood by All
  - Mathematics Used to Verify Correctness
- Tools Aid in Construction of Specification
  - Word Processors, Visual (PPT, Visio, etc.)
  - Analyzers (e.g., Logic)
  - Design Tools (Together Architect/Rhapsody)
- What’s in a Specification?
  - Diverge to Web Page for Additional Presentation
  - Accompanying Document
Return from “What’s in a Specification”

- 30 Pages of Main text
- Supplemental Information:
  - Conceptual, High Level, Detailed Designs
  - 25 Page Object Model with Explanation
  - 7 Pages of ER Diagrams
  - 200 Pages of DFDs
- Object Model Underestimated Requirements by Factor of 10!
  - 15 Classes Estimated, 150 Classes Designed/Built
  - First Very OO Project by PB
- Specification Not Updated Over Time

What Did their Specification Contain?

- Consulting Project in 1995 with Pitney Bowes
  - Investigate the Ability to Sell Postage over Internet
  - C++/OO Windows 95/Add-on Hardware Device
- Their Specification Contained
  - Introduction: Purpose, Scope, Terms, Abbr, etc.
  - Models of Operation: System Ops + High-Level
  - System Functions w.r.t. Performance/Reliability, Security/Confidentiality, Other Domain Issues
  - Details from User Perspective: Interactions and Behavior
  - Details from System Perspective: The way the System Works + Components + User Interactions
  - Business Related Considerations: Feasibility of Idea, Marketability, etc.
## Concluding Remarks

### Specifications Describe
- What the Users Need from a System (Requirements Specification)
- Design of a Software System (Design and Architecture Specification)
- Features Offered by a System (Functional Specification)
- Performance Characteristics of a System (Performance Specification)
- External Behavior of a Module (Module Interface Specification)
- Internal Structure of a Module (Internal Structural Specification)

### Descriptions are given via Suitable Notations
- There is no “Ideal” Notation
- Notations, Diagrams, Approach, on Application-by-Application Basis

### Three Keys Issues:
- Specifications must be Modular
- Specifications support Communication and Interaction between Designers and Users
- Specification Process Itself Must be Managed
  - Versions of Specification (CMS)
  - Assigning Specification Parts to D&D Team
  - Updating Specification as Changes Occur
  - Etc.