08- Notation-Formal

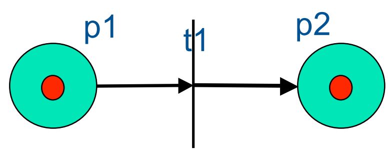
Rick Adrion

Engineering and Computer Job Fair

- Campus Center on October 1 from 10 am - 3 pm
- Microsoft, Mitre, GE, FAA and BAE
- seeking Computer Science students for permanent, summer and co-op positions

Petri Nets

- Petri nets are “marked” graphs
  - two node types: places & transitions
  - tokens mark the nodes
  - transitions are enabled (“fire”) if all connected places contain tokens

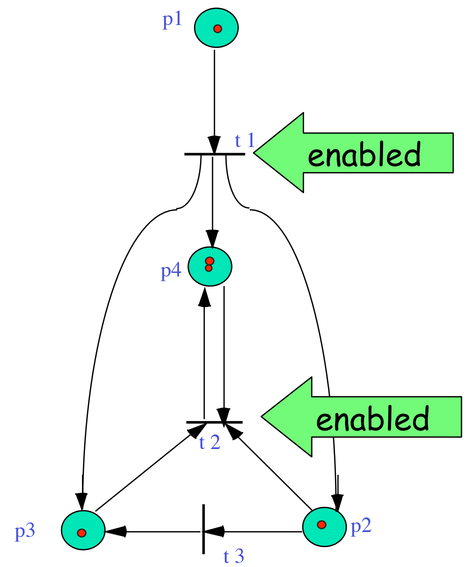


  - Options: simultaneous or asynchronous

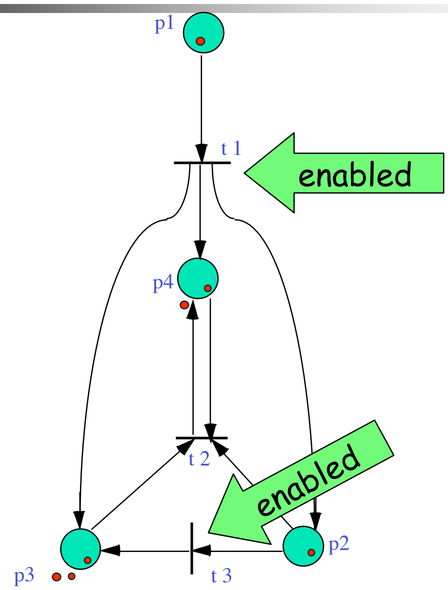
Petri Nets: Informal Definition

- Designed specifically for modeling systems with interacting concurrent components.
- Consists of a set of places and a set of transitions
  - Edges connect places and transitions.
  - Only transition $\rightarrow$ place and place $\rightarrow$ transition links are allowed.
- Each place can have a finite number of tokens.
- A transition is enabled if each of its input places has at least one token.
  - An enabled transition can fire: one token is taken from each input place and one token is put into each output place.

Petri Net example

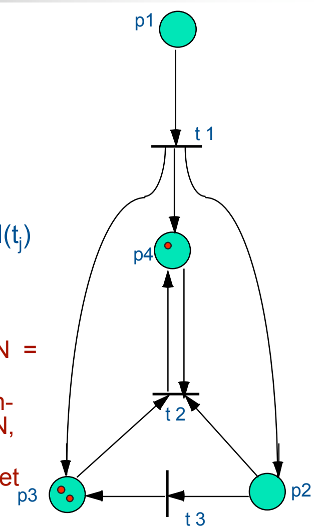


Petri Net example



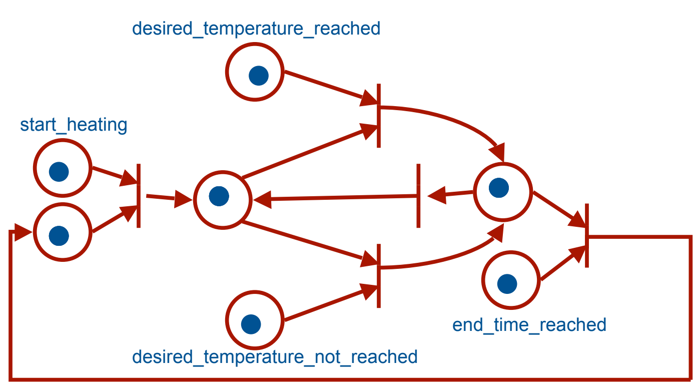
Petri Nets: Formal Definition

- A Petri Net is a four-tuple, C=(P,T,I,O)
- $P = \{p_1, p_2, ..., p_n\}$, $n \geq 0$ is a finite set of places.
- $T = \{t_1, t_2, ..., t_m\}$, $m \geq 0$ is a finite set of transitions.
  - I: $T \rightarrow P$ is the input function.
  - O: $T \rightarrow P$ is the output function.
- p_i is an input place of a transition t_j if $p_i \in I(t_j)$
- p_i is an output place of a transition t_j if $p_i \in O(t_j)$
- Petri Net markings
  - A marking m is a mapping $P \rightarrow N$ where N = 0, 1, 2, ...
  - The marking m can be represented as a n-vector m = $(m_1, m_2, ...m_n)$, n = |P |, $m_i \in N$, $1 \leq i \leq n$
  - A marked Petri net M = (C, m) is a Petri net C and a marking m .



marking (0,0,2,1)

Petri Net for Heating Controller



UNIVERSITY OF MASSACHUSETTS AMHERST • DEPARTMENT OF COMPUTER SCIENCE • CMPSCI 520/620 FALL 2003

More on Petri nets

- if there exists a marking which is reachable from the initial marking where no transitions are enabled, such a transition is called a "deadlock"
- a PN with no possible deadlock is said to be live, called the "liveness property"
- in simplest PN, tokens are uninterpreted
  - in general, a selection policy can not be specified
  - have no "policy" for resolving conflicts, potential "starvation"
- many extensions:
  - Hierarchical Petri Nets
  - Colored tokens
  - "Or" transitions
  - Queues at places

Petri Nets vs. FSA

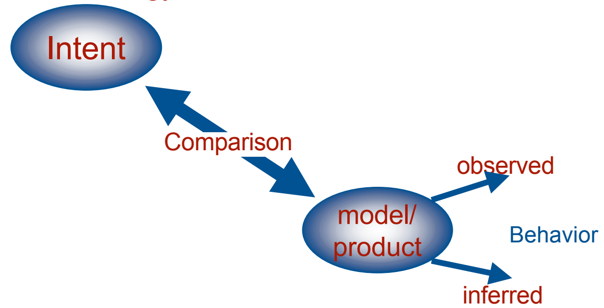
- For any finite state machine, a Petri net can be built that models the finite state machine
  - Petri nets are as powerful as finite state machines
- Petri nets advantages:
  - net composition (in different forms) can be found easier than the cross-product of finite state machines
  - parallelism and nondeterminism are represented in a more understandable way
- FSA advantage:
  - simpler graph structure for some applications (e.g. parsers)

How to write it down?

- natural language
- structured natural language
- pictorial notation
  - Charts, Diagrams, Box-and-Arrow Charts
  - Graphs
    - Flowgraphs
    - Parse Trees
    - Call graphs
    - Dataflow graphs
- **formal language(s)**
  - state-oriented
  - function-oriented
  - object-oriented

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 $\supset$ Sat is a **relation**.
- 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* $\in$ Syn is the set of all specificands *sem* $\in$ Sem, such that
    Sat (*syn,sem*)

from Wing

Properties

- a specification *syn* $\in$ Syn is **unambiguous** if and only if Sat maps *syn* to exactly one specificand set.
- a specification *syn* $\in$ 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* $\in$ Sem is **correct** with respect to a given specification *spec* $\in$ 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: DF, 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: $\Rightarrow$ stack
pop: stack $\Rightarrow$ stack
push: stack X integer $\Rightarrow$ stack
top: stack $\Rightarrow$ 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) $\Rightarrow$ S
  {$\forall$ J, 1 < J $\leq$ S_0.top
  S_0.data [J] = S.data [J] $\wedge$
  S.top = S_0.top + 1 $\wedge$
  S.Data [S.top] = I }

Ordered Sets

- ordered set definition:
  - $X = \{x_0, x_1, \ldots, x_n\}$
  - $|X| = n + 1$
  - $extract(X) = \{x_0, x_1, \ldots, x_{n-1}\}$
- operational definitions:
  - create = { 0 }
  - push $(S_0, I) = S \wedge$
    - $S_0 = extract(S) \wedge$
    - $|S| = |S_0| + 1 \wedge$
    - $x_{|S|} = I$

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

- M … ndays … 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* (set; elements)
birthday: *NAME* ↪ *DATE* (function)

known = **dom** *birthday* (invariant)

Another Schema

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

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

stat

r

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

Another Schema

FindBirthday
ΞBirthdayBook
name?: NAME
date!: DATE

name? ∈known
date! = birthday (name?)

no state change

apply fn

invariants

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

Trace specifications

NAME
label
SYNTAX
name: __type ... __type ⇒ return_value_type
SEMANTICS
assertions of the form:
L(T) -- asserts that T is a legal trace
V(T) = *value* -- is the value returned if T ends in a function call

- operator precedence

≡ < “ = ≥ >

¬

& ~ |

⇒ ⇔

Trace specifications

T1 ≡ T2 ⇒

(∀T) ((L(T1·T) ⇒ L(T2·T)) &

(T is not empty ⇒ (

(T_1·T has a value ⇔ T_2·T has a value) &

(T_1·T has a value ⇒V(T_1·T)= V(T_2·T))))

- note (∀S,T) (L(S·T) ⇒ L(S))

-

Example

NAME

stack

SYNTAX

push: integer;

pop: ;

top: ⇒ integer;

SEMANTICS

/*1*/ (∀T,i) (L(T) ⇒ L(T·push(i))

/*2*/ (∀T) (L(T·top) ⇔ L(T·pop)

/*3*/ (∀T,i) (T ≡T·push(i)·pop)

/*4*/ (∀T) (L(T·top) ⇒ T ≡ T·top)

/*5*/ (∀T,i) (L(T) ⇒V(T·push(i)·top)=i)

Interpretation

/*1*/ (∀T,i) (L(T) ⇒ L(T·push(i))

/*1*/ unbounded stack

/*2*/ (∀T) (L(T·top) ⇔ L(T·pop)

/*2*/ top cause no error iff pop causes no error

/*3*/ (∀T,i) (T ≡T·push(i)·pop)

/*3*/ push followed by pop does not affect the future behavior

/*4*/ (∀T) (L(T·top) ⇒ T ≡ T·top)

/*4*/ top does not affect the behavior

/*5*/ (∀T,i) (L(T) ⇒V(T·push(i)·top)=i)

/*5*/ how to compute the value of top

Example - using /*3*/ and /*5*/

note: push(i)·push(j)·push(k)·pop·pop·top ⇒ top= i

By /*3*/ (∀T,i) (T ≡T·push(i)·pop)

By /*5*/ (∀T,i) (L(T) ⇒V(T·push(i)·top)=i)

Heuristics

- define normal forms
- structure semantics
- use predicates
- develop specs incrementally
- use macros

Comparison

- trace specifications
  - based on call sequence
  - no "hidden functions"
  - natural application to inter-process communication
  - universal & existential quantifiers
- algebraic specifications
  - based on "type of interest," therefore maybe in terms of objects not visible to user
  - requires "hidden functions"
  - cannot handle concurrency
  - no existential quantification

Property-oriented techniques

- Abstract-data-type specification languages
  - Axiomatic: Hoare, OBJ, Anna, Larch, and algebraic, e.g., Clear, ActOne, Aspeque
  - Concurrent and distributed systems specification languages: temporal logic, Lamport, LOTOS
- Semi-formal
  - ER diagrams

Logic Specifications

- Expressed using formulas under a first order logic theory (usually with quantification), e.g.,
  - $\exists$ j [1 $\leq$ j $\leq$ s.top| t.data[j]=s.data[j]]
- Typically expressed as pre- and post-conditions, e.g.,
  - Let P be a sequential program
  - with inputs $(i_0,i_1, \ldots ,i_n)$ and outputs $(o_0,o_1, \ldots ,o_m)$
  - Pre $(i_0,i_1, \ldots ,i_n)$ P Post$(o_0,o_1, \ldots ,o_m,i_0,i_1, \ldots ,i_n)$ is a property

UNIVERSITY OF MASSACHUSETTS AMHERST • DEPARTMENT OF COMPUTER SCIENCE • CMPSCI 520/620 FALL 2003

"Hoare" example

type stack =

record top: integer

data:array [1 ... 100] of integer

end

t:= push(s, i)

true{t:= push(s, i)} ∃ j [1≤ j≤s.top| t.data[j]=s.data[j]

∧ t.data[t.top] = i

∧ t.top =s.top +1]

precondition

"program"

post condition

"Hoare" example

Logic specification:

true {t:= push(s, i)} ∃ j [1 ≤ j ≤ s.top| t.data[j]=s.data[j]

∧ t.data[t.top] = I ∧ t.top =s.top +1]

Operational specification

{true} push (S_0, I) {∀ J, 1 < J ≤ S_0.top

S_0.data [J] = S.data [J] ∧

S.top = S_0.top + 1 ∧

S.Data [S.top] = I }

Algebraic Specification

Stack (S) $\wedge$ Integer (I) …

(1) Top (Push (S, I)) = I

(2) Top (Create) = Integer Error

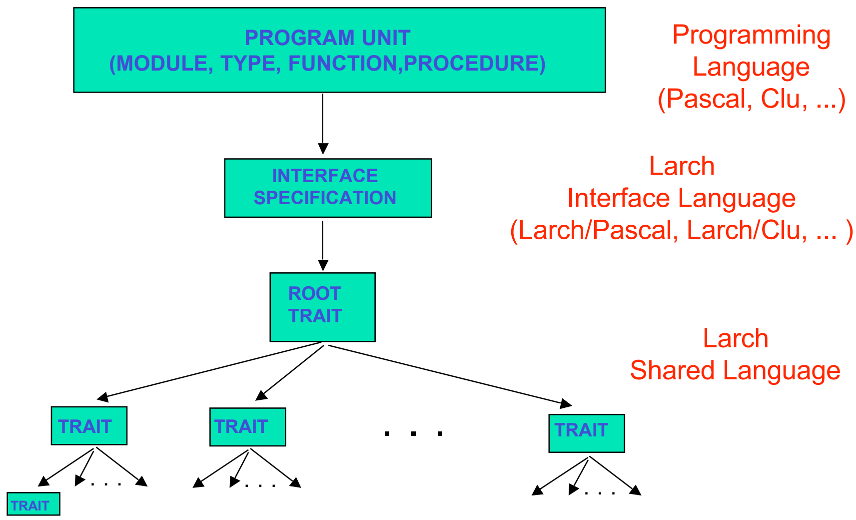
(3) Pop (Push (S, I)) = S

(4) Pop (Create) = Stack Error

Larch

- The Larch Family of Specification Languages
  - John Guttag, James Horning, Jeannette Wing IEEE Software, 1985
- Larch Shared Language
  - Common language for formally representing models
- Larch Interface Language
  - Interface between the shared language and the target programming language
    - Larch/Pascal
    - Larch/CLU
- Specific implementation language

Larch



Terminology

SPECIFICATION TERM	PROGRAMMING LANGUAGE TERM
Operator	Function
Sort	Type
Term	Expression
Trait	Module (ADT), Function, Procedure type

Goals of Larch

- Composability
  - Common specifications from existing specifications
  - Library or handbook
- Readability
- Localize programming language dependence
  - General model is very complex so use different language specific models
- Automated Support
  - Construction tool
  - Syntactic checking
  - Semantic checking
  - Support incompleteness

Trait

Introduces — signature of the operation (sort checking)

Constrains — constrains the operations & relations among the operators

theory - set of theorems that can be proved about the operator done by substitution, using rules of first order predicate calculus with equality

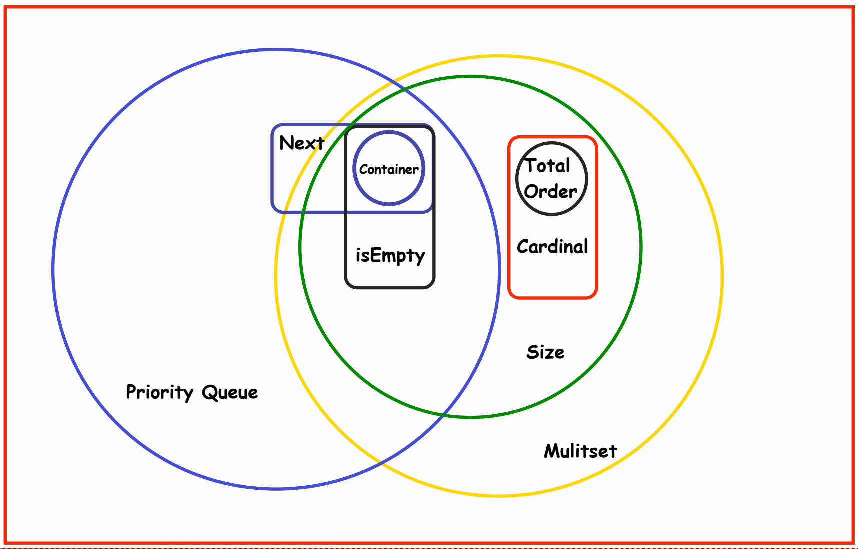
Examples

Container: trait
introduces
new: → C
insert: C, E → E
constrains C **so that**
C **generated by** [new, insert]

Examples

IsEmpty: **trait**
assumes Container
introduces
isEmpty: C → Bool
constrains isEmpty, new, insert
so that for all [*c* :C, *e* :E]
isEmpty(new) = true
isEmpty(insert(c,e)) = false
implies converts [isEmpty]

Constructing traits



Interface Languages

- “bridge” between shared language and implementation language
- “Two-tiered” specification approach: principal innovation of Larch w/r/t algebraic specification languages

Interface Languages

- Larch/L incorporates "flavor" of L
  - semantics, keywords
  - makes it easier for those who know L to write provable specs
  - just need to adapt existing shared traits from Library (in theory...)
- Larch/L languages designed to support data abstraction, even if language L doesn't directly support it (Pascal, C, C++)

Larch/Pascal specification

```
type Bag exports bagInit, bagAdd, bagRemove, bagChoose
  based on sort Mset from MultiSet with [integer for E]
  procedure bagInit(var b:Bag)
          modifies at most [ b ]
          ensures bpost = { }
  procedure bagAdd(var b:Bag; e; integer)
          requires numElements(insert(b,e )) ≤ 100
          modifies at most [ b ]
          ensures bpost = insert(b,e )
  procedure bagRemove(var b:Bag; e; integer)
          modifies at most [ b ]
          ensures bpost = delete(b,e )
  procedure bagChoose(var b:Bag; e; integer): boolean
          modifies at most [ b ]
          ensures if ~ isEmpty (b )
           then bagChoose & count (b, epost)>0
           else ~ bagChoose & modifies nothing
End Bag
```

UNIVERSITY OF MASSACHUSETTS AMHERST • DEPARTMENT OF COMPUTER SCIENCE • CMPSCI 520/620 FALL 2003

Pascal implementation of BagAdd

```
prodedure bagAdd(var B:Bag;e:integer);
  var i, lastEmpty: 1...MaxBagSize
  begin
    i:= 1;
    while ((i < MaxBagSize) and (b.elems[i]<>e)) do
      begin
        if b.counts[i] = 0 then LastEmpty:=i;
        i:= i+1;
      end;
    if b.elems[i] = e
      then b.counts[i]:= b.counts[i]+1;
      else begin
        if b.counts[i]=0 then LastEmpty:=i;
        b.elems[LastEmpty]:=e;
        b.counts[LastEmpty]:=1;
      end;
end[bagAdd];
```


Conclusions

- Interesting attempt to address:
  - readability/writability of formal specs
  - large, multi-lingual environment issues
- Relationship between shared and interface languages complex and unclear
- Relationship between interface and implementation languages not as strong as one would like
- “Software tool support needed” (syntax-directed editors, browsers, theorem-provers, etc.)

Current Status

- Strong theoretical foundation
- Some practical use, especially in Europe
- Current Languages trying to be more practical

How effective are these methods?

- Wing's study of the Library Problem
  - a small library database
  - transactions
    - checkout/return book
    - add/remove book
    - get a list of books
      - author
      - subject
      - borrower
    - get date/borrower for book
  - users
    - staff
    - borrowers
  - restrictions
    - availability
    - no book available & checked out
    - # books borrowed ≤max

Analysis

- Specification approaches
  - informal
  - AI
  - logic
  - executable/non-executable
- Comparisons
  - formality
  - life-cycle phase
  - operational vs. behavioral
  - modularity
  - readability
  - completeness
- Not considered
  - concurrency
  - reliability
  - fault-tolerance
  - security
- initialization
  - what's the initial state of the library?
- missing operations
  - need more transactions?
- error handling
  - what to do with errors?
  - checkout, return, add, remove, "type errors"
- missing constraints
  - more than one copy in library, checked out
- state
  - what to record, change?
- "non-functional" specification
  - human factors, liveness, time

UNIVERSITY OF MASSACHUSETTS AMHERST • DEPARTMENT OF COMPUTER SCIENCE • CMPSCI 520/620 FALL 2003

Conclusions

- methods do not differ radically
- style
  - most use pre- and post-conditions for specifying behavior
  - algebraic & set-theoretic most common for specifying data (operational)
  - model-oriented (operational) most common approach
- formal specs can
  - identify diff in informal specs
  - handle simple, small problems
  - specify sequential functional behavior
- Challenges
  - scaling
  - non-functional behavior
  - combining techniques
  - tools
  - integrating specification into the lifecycle

UNIVERSITY OF MASSACHUSETTS AMHERST • DEPARTMENT OF COMPUTER SCIENCE • CMPSCI 520/620 FALL 2003

