Announcements

- Wednesday, October 20th
  - **There will be class** (live)
  - **Extreme Programming (XP)** -- Matt Cornell & Agustin Schapira, *Knowledge Discovery Laboratory*
- PEEAS Projects
  - Project 1 & due dates posted …

10 Notation [Formal Methods]

- 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.

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 ⇒ S
  - can be a partial function
- represent as a graph
  - nodes ⇒ states
  - edges ⇒ inputs
  - graph ⇒ transition function
- enhancements to FSM
  - Use of hierarchy
  - Output annotations on edges
  - Distinguished Initial and Terminal states
  - Separate data definitions, local and global variables

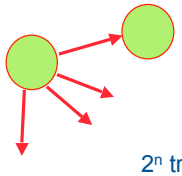
Why Use FSM's?

- Primary appeal is visualization
  - Intuitively: "watch" a stream of inputs "drive" the behavior of the system as a sequence of movements from state to state
- What is FSM good/not good for? (Example: safety concern)
  - Advantages:
    - 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

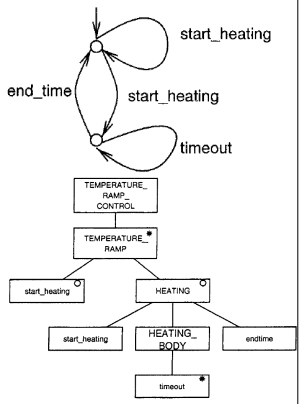


new state = libe - book

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

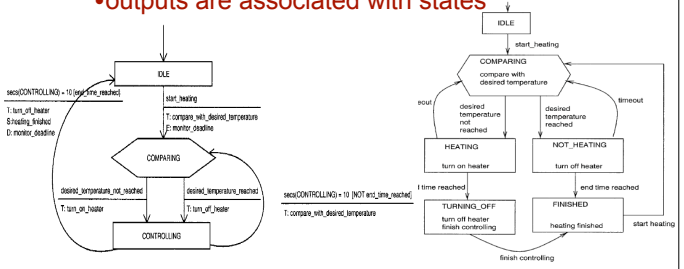
- Finite State Diagrams
  - contain only finitely many states and transitions
- Extended Finite State Diagrams.
  - number of states can be increased by introducing variables that may be tested and updated by the finite state machine
  - *global state* = *explicit state* (STD nodes) + *extended state* (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 non-determinism
  - 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.

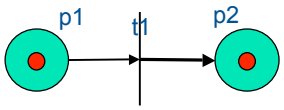
Mealy & Moore Machines

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



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

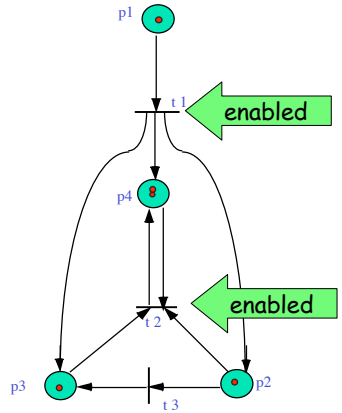


- Options: simultaneous or asynchronous

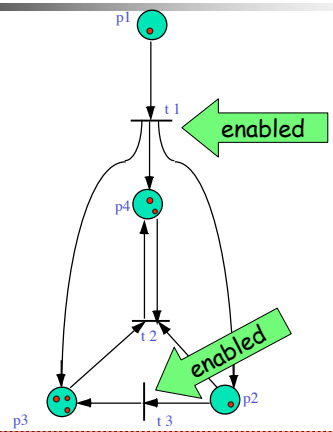
Petri Nets: Informal Definition

- 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 → place and place → 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

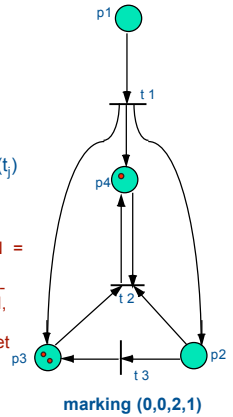


Petri Net example



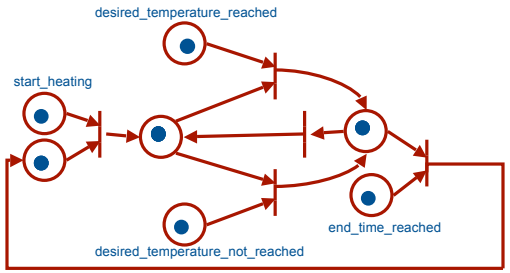
Petri Nets: Formal Definition

- A Petri Net is a four-tuple, C=(P,T,I,O)
- P = $\{p_1, p_2, ..., p_n\}$, n ≥ 0 is a finite set of places.
- T = $\{t_1, t_2, ..., t_m\}$, m ≥ 0 is a finite set of transitions.
  - I: T → P is the input function.
  - O: T → 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 → 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 |, $m_i \in N$, $1 \le i \le n$
  - 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



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)

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)

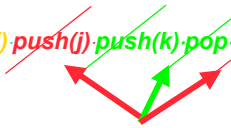
-

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.,
  - $\exists\ j\ [1 \le j \le s.top |\ t.data[j]=s.data[j]]$
- 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

"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

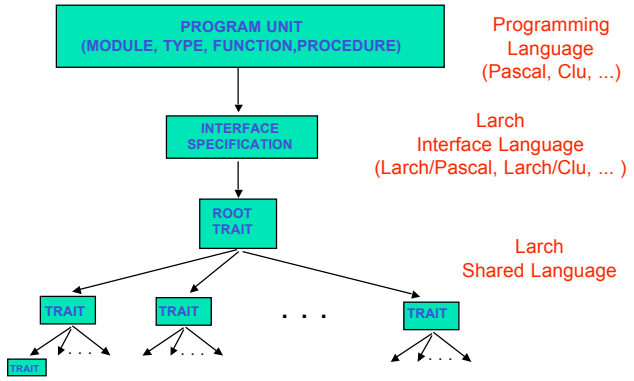
Stack (S) ∧ Integer (I) …

(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

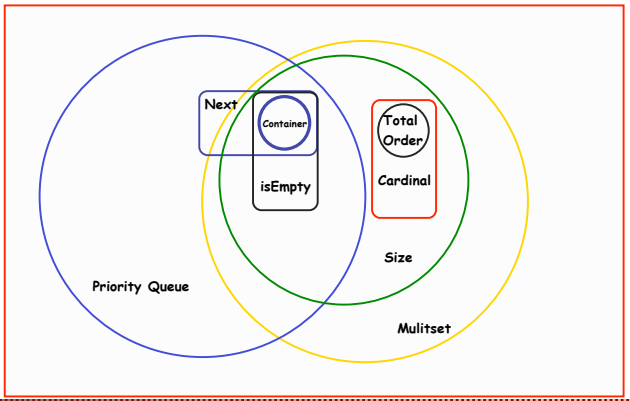
Examples

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
```


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 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

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

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