administrivia

automated formal methods for design
- Z/np today checked by ladybug
- temporal logic/smv, CSP/fdr on monday

friday project meetings in class
- another status report due next monday

any questions

brief word from our sponsor ...

registration is opening up in the next few days

two courses of possible interest

Human Computer Interaction CS296
- available to all CS juniors+

Tractable Software Analysis
- grad students only

model checking

on to automated tools
- phewww...

model checking is beginning to be used to check code
- developed to check hardware
- then used for protocol and design analysis

takes system description and claim
- produces counter example if claim false

model checking/static analysis

model checking varies from traditional static analysis
- static analysis more like traditional formal methods
- but completely automated

static analysis proves code “correct”
- no type errors for example

automated proving limited
- can’t verify rich properties

what is done is easy though

spin

- spin is model checker targeted at code
  - developed by gerard holzmann (bell labs)
  - development started in 1980
  - released publicly in 1992
  - won ACM System Software award in 2002
- similar logic to smv
  - plus some ideas from CSP
  - very different syntax from either

formal notations

- discussing 3 formal design notations
  - Z/np focus on structural properties
  - temporal logic focus on simple dynamic
  - csp focus on multiple process dynamics
- want you need understand
  - what each can do
  - why bother with formal methods
- don't need to be able to write specs

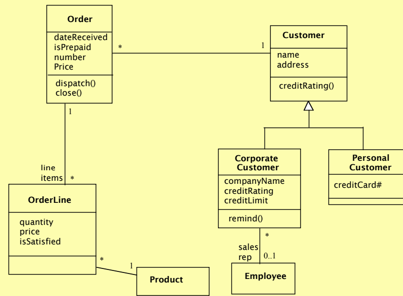
Z

- today talk about Z (and np)
- Z
  - most widely used formal notation
  - still miniscule usage
  - developed at Oxford
- focuses on structural issues
  - can this structure have a cycle?
  - is this field unique?

nitpick/ladybug

- three tools to check Z-like specs
  - nitpick written by Jackson (MIT) and me
  - ladybug written by me
  - alloy written by Jackson
- nitpick, ladybug use np language
  - almost subset of Z
  - first-order (more on this later)
  - uses ascii only

UML example



np types

3 kinds of things in np
- scalar objects (think enum)
- sets of scalar objects
- relations over scalar objects

schemas provide structuring
- `name = [ declarations | constraints]`
- almost class-like
- `| constraints` is optional

np types

np model
- instances represented by scalar objects
- types are sets of instances

top level types declared specially

```
[Order,Customer,OrderLine,Product,Employee]
```

enumerated types declared specially

```
Rating == { poor, … }
```

subtypes declared as subsets

```
OrderSystem = [
    CorporateCustomer : set Customer
    PersonalCustomer : set Customer
```


associations/attributes

associations and attributes
- usually given as functions
- from class defining attribute
- to class/type of attribute or association

simple multiplicity represented with
- total vs. partial, functional, injective

```
creditRating : tot Customer -> Rating
customerOrder : tot Order -> Customer
orderLines : tot OrderLine -> Order
product : tot OrderLine -> Product
salesRep : CorporateCustomer -> Employee
```

other attributes

boolean fields modelled as sets

- set of type defining field
- inclusion means attribute is true
- exclusion means attribute is false
- { x : Class | x.field == true}

```
isPrepaid : set Order
```

many attributes abstracted away

declarations

declaration parts define structure

- akin to what UML class diagram shows

describe attributes/associations over groups of instances a la classes

“backwards” descriptions

- describe the entire relationship at once
- OO from viewpoint of single instance

important distinction

- different way of thinking
- some things become easier to describe

constraints

constraints limit allowable cases

- akin to OCL constraints

included in constraints section

```
name = [ declarations | constraints]
```


relational expressions

constraints are relational expressions

- operate on scalars, sets and relations

traditional operations

- (some odd notation)
- U & /   union, intersection, difference
- ~ +   inverse, transitive closure
- *   reflexive transitive closure
- dom ran   actual domain, range
- = <= <   equality, subset, proper subset

and not or => <=>

more operators

domain restriction

s <| r

all pairs in r with first element in s

also range restriction

also negative domain, range restriction

f.o function application

r.{s} relational image

fun, tot, inj, bij

is relation functional, total, injective, bijective

examples

```
CorporateCustomer & PersonalCustomer = {}
```

```
ran (PersonalCustomer <| creditRating) = {
poor}
```


properties

schemas can be used to describe collections of states

- referred to as properties
- describe desirable or undesirable states

include other schemas in declarations

- grants visibility to all its state

```
Safe =
[
    OrderSystem
|
   customerOrder~.{creditRating~.{poor}} <=
isPrepaid
]
```


operations

only described static structure so far

- every interesting system changes

operation is special kind of schema

add parameter list to schema name

- may be empty
- parameter values are constant during op
- other variables have pre and post values
- post-state is primed

operation example

```
addOrder(o:Order, ol:OrderLine) =
[
    OrderSystem
|
   orderLines'.ol = o
   orderLines < orderLines'
   dom orderLines U { ol } = dom
orderLines'
]
```


claims

claims are the goals

- replace = with :: in any schema

```
isSafe(o:Order, c:Customer)::
[
    OrderSystem
|
   Safe and addOrder(o,c) => Safe'
]
```


common claims

two common categories of claims

invariance

```
property and operation => property'
```

implication

```
property => property2
```


checking

ladybug will check that claim holds for all possible instantiations

limited by scope

- number of instances of each type
- limited to small numbers (5±)

ladybug returns counterexamples

- if any
- limited to roughly 10^{120} total states
- usually finishes in seconds

example

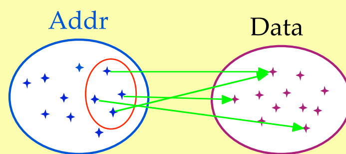
consider memory allocator (malloc)

model in terms of functions, sets

key concepts in allocator

- usage maps addresses to data
- used set of addresses in use
- newAddr new address allocated

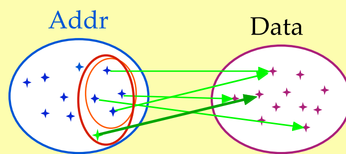
allocator example



usage maps addresses to data

used includes addresses in use

allocator example



usage′ maps addresses to data

used′ includes addresses in use

newAddr is the newly allocated address

formalizing allocator

model imposes restrictions

addresses in use map data

$used = dom\ usage$

$used' = dom\ usage'$

allocation does not change mapping of existing memory

$used \lhd usage' = usage$

formalizing allocator

newly allocated address now in use

used′ = used ∪ { newAddr }

goal: newAddress not already in use

not newAddr ∈ used

target formula

convert claim to single formula

system and not goal

for allocator

used = dom usage and used′ = dom usage′ and
used ◁ usage′ = usage and
used′ = used ∪ { newAddr } and
newAddr ∈ used

counterexamples

in less than one second:

```
Found Counterexample to Claim uniqueAddrAlloc:
a : Addr =
    a0
inUse : set Addr =
    { a0 }
inUse' : set Addr =
    { a0 }
usage : Addr->Value =
    { a0 -> v0 }
usage' : Addr->Value =
    { a0 -> v0 }
Found 1 Counterexample
```


experiences

ladybug still research tool

has several success stories

- found mobile ipv6 bug
- found bug in proof of faa system safety
- found bugs in HLA
- used to check british rail system spec

allocator specification (1)

```
/* A trivial specification of a memory allocator
  - e.g. malloc */
/* Specify the given types */
[Addr, Data]
/* Define the heap itself with a schema */
Heap =
[
  /* Define Variables */
  usage : Addr -> Data
  used : set Addr=
|
  /* all currently mapped addresses are used*/
  used = dom usage
]
```


allocator specification (2)

```
/* Alloc: an operation to allocate memory */
Alloc(newAddr : Addr) =
      /* newAddr is the newly alloc'd addr */
[
  Heap      /* Include the Heap definition */
|
  /* The existing part of usage' is unchanged*/
  used <: usage' = usage
  /* newAddr is now mapped(to unspecified data) */
  used' = used U {newAddr}
]
```


allocator specification (3)

```
/* Make a claim about Alloc */
uniqueAddrAlloc::
[
  Heap
  newAddr : Addr
|
  /* A newly allocated addr should not
  have been in use */
  Alloc(newAddr) => newAddr not in use
]
```