Lecture aims

1. Refresh knowledge of first-order predicate logic and set theory.
2. Critique the use of formal modelling in software engineering, why it is used, what it is good for, what it is not good for.
3. Read and write basic of Alloy state machines from informal requirements.
4. Use Alloy operators to incrementally derive a specification.

Lecture plan

1. Background:
   (a) Overview of set theory, propositional logic, and predicate logic
   (b) Important concepts:
   (c) Propositional logic: atomic propositions, conjunction, disjunction, implication, equivalence, negation, tautologies and contradictions
   (d) Predicate logic: quantifiers (all, some, none, lone, one), predicates
   (e) Relational calculus: sets as unary relaions, set membership, Cartesian product, union, intersection, difference, dot join, box join, relations, functions (total and partial), function override, domain & range restriction.
2. Motivation for formal, model-based specification:
   (a) Example: sorting a list; binary search
   (b) What is formal specification? What is model-based specification?
   (c) Why use formal specification?
   (d) Why use set theory and predicate logic?
   (e) Cost savings: no ambiguity, solve problems up front, easier to implement, *proof*, post-release costs (more mistakes made *and found* in specification).
   (f) Objections: too hard for "average" programmer
3. Alloy
   (a) Motivation: based on Z which is well-known, highly-used language; intuitive notation, state-of-the-art tool support. Loads and loads of examples.
   (b) Alloy = logic + language + analysis; analysis is NOT proof, but counterexample driven.
   (c) Basic: modules.
   (d) Running example of the PassBook
   (e) Signatures are basically types.
   (f) Facts hold over entire signature space
   (g) Predicates and functions; we use these to specify *operations*.

(h) Assertions state properties about model; can be checked but also serve to help understand the model itself.

(i) Incremental specification: abstract state machines (state, inv, init, operations); preconditions and postconditions, inclusion (building up from components using conjunction and disjunction), assertions (invariant preservation).

4. Formal development process (see figure below).

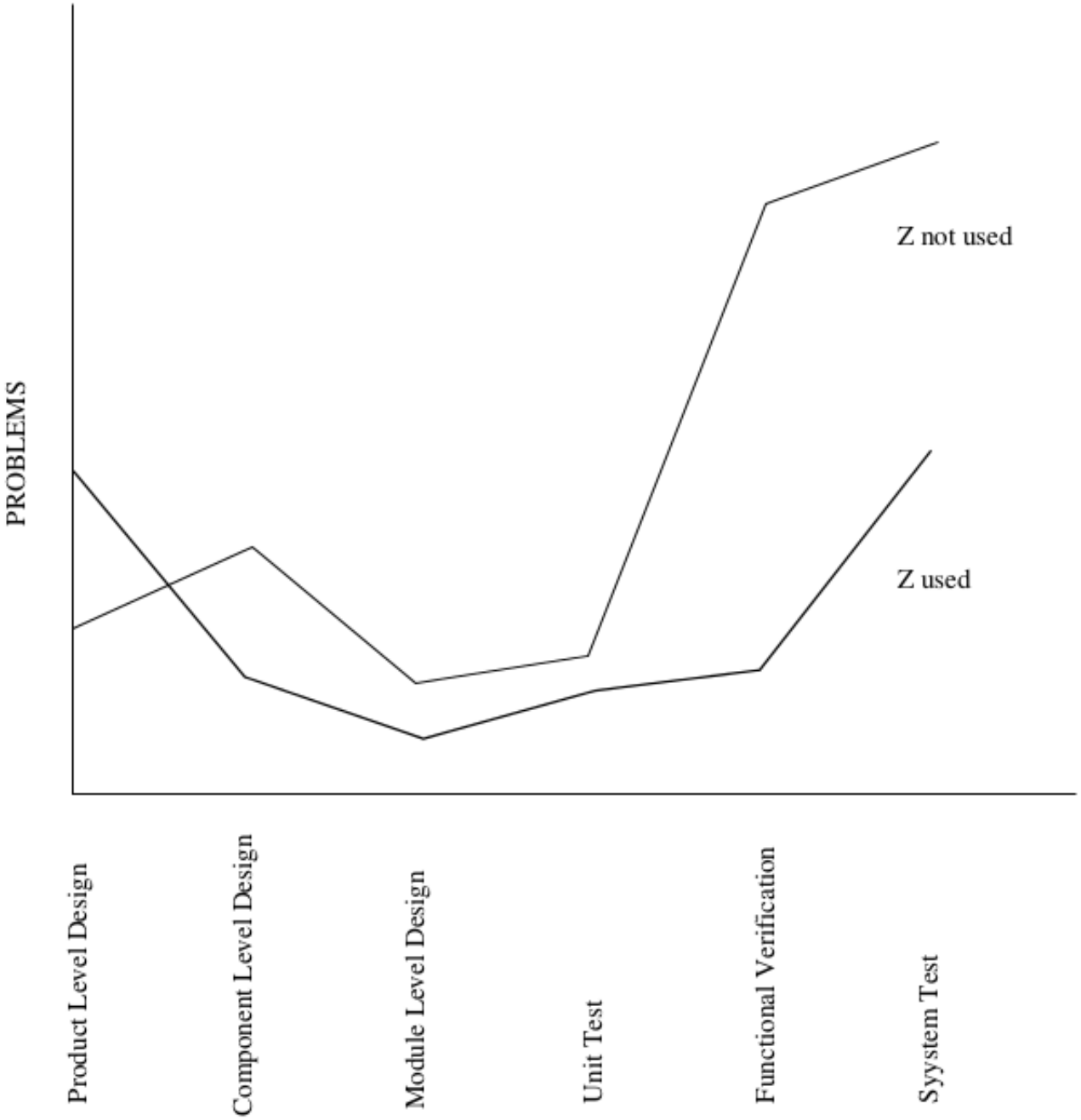
5. V&V: animation, model checking, review!!, and proof.

6. Example: Needham-Schroeder protocol mutual authentication. Considered correct for many years, but man-in-the-middle attach was possible, and found using formal proof.

7. Example: modal checking – NASA's use of checking high-level models to find deadlock.

8. Show Bill Gates quote about proof (without the name!) below.

9. Example proof: storage tank in chemical plant does not overflow.


PROBLEMS
Z not used
Z used
Product Level Design
Component Level Design
Module Level Design
Unit Test
Functional Verification
Syystem Test

Write informal specification
Decompose system
Specify system components
Compose component specifications
Define data sets and types
Define state variables and invariant
Define initial state
Define "normal" behaviour operations
Define exceptional behaviour operations
Compose operation schemas

“Things like even software verification, this has been the Holy Grail of computer science for many decades but now in some very key areas, for example, driver verification we’re building tools that can do actual proof about the software and how it works in order to guarantee the reliability.”

```
module LastPass

/*
 * LastPass password map
 *
 * A simple example to explain basics of Alloy.
 *
 * The 'PassBook' keeps track of a set of users' passwords for a set of URLs.
 * For each User/URL pair, there is one password. Passwords can be added,
 * deleted, looked up, and changed.
 * A user can also request all of their password for all URLs.
 *
 * author: Tim Miller
 */

sig URL {}
sig Username {}
sig Password {}
sig PassBook {known : Username -> URL, password : known -> one Password}

fact NoDuplicates
{
    all pb : PassBook, user : Username, url : URL | lone pb.password[user][url]
}

//Add a password for a new user/url pair
pred add [pb, pb': PassBook, url : URL, user: Username, pwd: Password] {
    no pb.password[user][url]
    pb'.password = pb.password + (user->url->pwd)
}

//Delete an existing password
pred delete [pb, pb': PassBook, url : URL, user: Username] {
    one pb.password[user][url]
    pb'.password = pb.password - (user->url->Password)
}

//Update an existing password
pred update [pb, pb': PassBook, url : URL, user: Username, pwd: Password] {
    one pb.password[user][url]
    pb'.password = pb.password ++ (user->url->pwd)
}

//Return the password for a given user/URL pair
fun lookup [pb: PassBook, url : URL, user : Username] : lone Password {
    pb.password[user][url]
}

//Check if a user's passwords for two urls are the same
pred samePassword [pb : PassBook, url1, url2 : URL, user : Username] {
    lookup [pb, url1, user] = lookup [pb, url2, user]
}

//Retrieve all of the passwords and the url they are for, for a given user
pred retrieveAll [pb: PassBook, user : Username, pwds : URL -> Password] {
    pwds = (pb.password)[user]
}
```

```
//Initialise the PassBook to be empty
pred init [pb: PassBook] {
    no pb.known
}

//If we add a new password, then we get this password when we look it up
assert addWorks {
    all pb, pb': PassBook, url : URL, user : Username, p : Password |
        add [pb, pb', url, user, p] => (lookup [pb', url, user] = p)
}

//If we update an existing password, then we get this password when we look it up
assert updateWorks {
    all pb, pb': PassBook, url : URL, user : Username, p, p' : Password |
        lookup [pb', url, user] = p =>
            (add [pb, pb', url, user, p'] => (lookup [pb', url, user] = p'))
}

//If we add and then delete a password, we are back to 'the start'
assert deleteIsUndo {
    all pb1, pb2, pb3: PassBook, url : URL, user : Username, pwd : Password |
        add [pb1, pb2, url, user, pwd] && delete [pb2, pb3, url, user]
            => pb1.password = pb3.password
}

run add for 3 but 2 PassBook
run add for 5 URL, 5 Username, 10 Password, 2 PassBook
check addWorks for 3 but 2 PassBook expect 0
check updateWorks for 3 but 2 PassBook expect 0
check deleteIsUndo for 3 but 2 PassBook expect 0
```

```
module ExtendedLastPass

open util/ordering[PassBook] as ord

/*
 * LastPass password map
 *
 * A simple example to explain basics of Alloy.
 *
 * The 'PassBook' keeps track of a set of users' passwords for a set of URLs.
 * For each User/URL pair, there is one password. Passwords can be added,
 * deleted, looked up, and changed.
 * A user can also request all of their password for all URLs.
 *
 * author: Tim Miller
 */

sig URL {}
sig Username {}
sig Password {}
sig PassBook {known : Username -> URL, password : known -> one Password}

//Report signatures for informing whether an opertion was successful or not
abstract sig Report {}
sig Failed extends Report {}
sig Success extends Report {}

//Each user/url pair maps to exactly one password
pred inv [pb : PassBook] {
    all user : Username, url : URL | lone pb.password[user][url]
}

//True if and only if the user has no password for this URL
pred preAdd [pb: PassBook, url : URL, user: Username] {
    no pb.password[user][url]
}

//Add a password for a user/url pair
pred postAdd [pb, pb': PassBook, url : URL, user: Username, pwd: Password] {
    pb'.password = pb.password + (user->url->pwd)
}

//Add a password for a *new* user/url pair
pred addOk [pb, pb': PassBook, url : URL, user: Username,
                    pwd: Password, report : one Report] {
    preAdd [pb, url, user]
    postAdd [pb, pb', url, user, pwd]
    report in Success
}

//Fail to add a password that already exists
pred addDuplicate [pb, pb': PassBook, url : URL, user: Username,
                                  report : one Report] {
    not preAdd [pb, url, user]
    pb = pb'
    report in Failed
}
```

```
//Add a password for a *new* user/url, otherwise, add nothing
pred add [pb, pb': PassBook, url : URL, user: Username, pwd: Password,
                 report : one Report] {
    addOk [pb, pb', url, user, pwd, report]
      or
    addDuplicate [pb, pb', url, user, report]
}

//Delete an existing password
pred delete [pb, pb': PassBook, url : URL, user: Username] {
    one pb.password[user][url]
    pb'.password = pb.password - (user->url->Password)
}

//Update an existing password
pred update [pb, pb': PassBook, url : URL, user: Username, pwd: Password] {
    one pb.password[user][url]
    pb'.password = pb.password ++ (user->url->pwd)
}

//Return the password for a given user/URL pair
fun lookup [pb: PassBook, url : URL, user : Username] : lone Password {
    pb.password[user][url]
}

//Check if a user's passwords for two urls are the same
pred samePassword [pb : PassBook, url1, url2 : URL, user : Username] {
    lookup [pb, url1, user] = lookup [pb, url2, user]
}

//Retrieve all of the passwords and the url they are for, for a given user
pred retrieveAll [pb: PassBook, user : Username, pwds : URL -> Password] {
    pwds = (pb.password)[user]
}

//Initialise the PassBook to be empty
pred init [pb: PassBook] {
    no pb.known
}

assert initEstablishes {
    all pb : PassBook | init [pb] => inv [pb]
}

assert addPreservesInv {
    all pb, pb' : PassBook, user : Username, url : URL,
        pwd : Password, report : Report |
        inv [pb] and add [pb, pb', url, user, pwd, report] => inv [pb']
}

assert updatePreservesInv {
    all pb, pb' : PassBook, user : Username, url : URL, pwd : Password |
        inv [pb] and update [pb, pb', url, user, pwd] => inv [pb']
}

assert deletePreservesInv {
    all pb, pb' : PassBook, user : Username, url : URL |
        inv [pb] and delete [pb, pb', url, user] => inv [pb']
}
```

```
//If we add a new password, then we get this password when we look it up
assert addWorks {
    all pb, pb': PassBook, url : URL, user : Username, p : Password, report : Report |
        addOk [pb, pb', url, user, p, report] =>
            (lookup [pb', url, user] = p and report in Success)
}

//If we update an existing password, then we get this password when we look it up
assert updateWorks {
    all pb, pb': PassBook, url : URL, user : Username, p, p' : Password |
        lookup [pb, url, user] = p =>
            (update [pb, pb', url, user, p'] => (lookup [pb', url, user] = p'))
}

//If we add and then delete a password, we are back to 'the start'
assert deleteIsUndo {
    all pb1, pb2, pb3: PassBook, url : URL, user : Username, pwd : Password, report : Report |
        addOk [pb1, pb2, url, user, pwd, report] && delete [pb2, pb3, url, user]
            => pb1.password = pb3.password
}

//If we delete something that is not in the book, nothing happens
assert deleteDuplicateDoesNothing {
   all pb, pb' : PassBook, url : URL, user : Username |
      no lookup [pb, url, user] =>
          (delete [pb, pb', url, user] => pb = pb')
}

run add for 3 but 2 PassBook
run add for 5 URL, 5 Username, 10 Password, 2 Report, 1 Failed, 1 Success, 2 PassBook
check initEstablishes for 3 but 1 PassBook expect 0
check addPreservesInv for 3 but 2 PassBook expect 0
check updatePreservesInv for 3 but 2 PassBook expect 0
check deletePreservesInv for 3 but 2 PassBook expect 0
check addWorks for 3 but 2 PassBook expect 0
check updateWorks for 3 but 2 PassBook expect 0
check deleteIsUndo for 3 but 2 PassBook expect 0
check deleteDuplicateDoesNothing for 3 but 2 PassBook expect 0
```