EXAM PLAN

PLAN

1. Requirements => ???

2. Modelling

- Explaining the meaning of functions .. ✅
- Writing functions/ code: 😐

3. Testing

- Provinding statements that achieve x % coverage ✅

4. Abstract Interpretation

- interval analysis 🚫
- designing abstract transformers 😐

5. Pointer Analysis

- Flow sensitive/ Flow insensitive: Given state, write the corresponding program 😐
- Given object structure, find points-to sets ✅

6. Symbolic Execution

- Find what is the required input for ex. loop exactly three times ✅
- Proof using symbolic execution, by writing all possible traces fulfilling a certain constraint 🚫
-

TO REMEMBER

Coverage

- Loop Coverage: loops needs to be executed 0 times AND 1 time AND more

Coupling

Data Coupling => shared data structure

- **facade pattern** : restricting access to datastructure. Provides unified interface to a set of interfaces in a subsystem.
  => Advantages: simple interface for complex system, decouples clients from subsystem, can be used to layer subsystems
- **flyweight pattern**: make shared data immutable
  => Advantages: reduce the number of objects using shared objects, reduce state replication
- Pipe-and-filter pattern: avoiding shared data

Procedural Coupling => a module calls a mehtod of another module

- **Observer Pattern**: components may generate events and register for events in other components.
  => Define one-to-many dependency between objects so that when one object changes state all dependents are notified.

  => Bad: one-to-many, tricky, not reusable

- **Flyweight Pattern:** Use sharing to support large numbers of fine-grained objects efficiently.

  => Advantages

  - Reduce storage for large number of objects
    By reducing the number of objects using shared objects By reducing the replication of intrinsic state
    By computing (rather than storing) extrinsic state
- **Factory Method pattern**: define an interface for creating an object but let subclasses decide which class to instantiate.

- Abstract Factory pattern: Provide an interface for creating families of related or dependent objects without specifying their concrete classe:

  - Factory method
    Create one object
    Works at the routine level

Helps a class perform an operation, which requires creating an object

- Abstract factory

  Creates families of objects

  Works at the class level

  Uses factory methods (e.g. new_button is a factory method)

- **Strategy Pattern** : define a family of algos, encapsulate each one and make them interchangeable.
  => Consequences: provides alternative implementation without condition instruction, client must be aware of different strategies, communication overhead between strategy and context, increased number of objects

- **Visitor Pattern**: represent an operation to be performed on the elements of an object structure. Let's you define a new operation without changing the classes of the elements on which it operates.
  Consequences: adding new opertation is easy, better type checking, adding new concrete element is hard

Testing

- Control Flow Graph:
  - For branch coverage: each "if" represents two branches, take the if or don't
  - For finding the minimal set of inputs for havimng the maximal branch coverage, go though control flow graph and try to fulfill every condition and its oopposite.

Abstract Interpretation

- Apply standard Interval Analysis to compute sound and trivial invariants at fix-point:

  => per line: write the values each variable can take.

  - at the beginning all are beween minus and plus infinity
  - when given a value : between (value, value)
  - when inside a loop: between (value, infty) => if counter variable and augmenting
  - **Joins:** before a loop take as extrememum the extremum of the variable after the loop.
  - **Widening:**

Pointer Analysis

Flow Sensitive Pointer Analysis

- abstract state S, give a program that results in the given state : if there is an assignement to a field, do it first, then the assignement to the objects themselves.
- the result of the execution of line 1 is shown in the cells of line 2 etc..
- Can be used to prove that 2 pointers/variables are not aliasing => simply prove that the two points-to sets are disjoint, then the two variables don't alias.

Flow Insensitive Pointer Analysis

- if two labels point to the same object and another one, we cannot distinguish between what they are pointing to, hence cannot assign a field of one to another .
- object gets initialised => gets new allocation space "A1"
- obect gets assigned to another => keeps his allocation space + new one.
- field gets assigned another field => "allocation space of the object the field is attached to"."field" gets ALL the allocation space of the object it is newly assigned to.
  => Ex: "A2.f -> {A3,A2}"

- Abstract Semantics: Compute the least fixed point of the program's abstract seman-tics in the parity domain. Recall the parity domain consists of abstract elements{>,Even,Odd,⊥} : say whether the. mentionned variables are odd/even or else.

Symbolic Execution

- Combines testing and static analysis
- Completely automatic, aims to explore as many program executions as possible
- **But**: has false negatives: may miss program executions/ errors
- At any point during execution, SE keeps two formulas :
  - **Symbolic Store:** $\sigma_s \in SymStore = Var \rightarrow Sym$, with
    - Var: set of variables as before
    - Sym: set of symbolic values
    - σ_s: symbolic store

    => Example:$z = x + y$ will produce $\sigma_s : x \rightarrow x_0, y \rightarrow y_0, z \rightarrow x_0 + y_0$.
  - **Path Constraint**: records the history of all branches taken so far.
    - At the start of the analysis the constraint is set to true
    - Evaluation of the conditionals affects the path constraint but not the symbolic store.

> Example :
>
> Let: $\sigma_s : x \rightarrow x_0, y \rightarrow y_0$, and $pct = x_0 > 10$

> Let's evaluate: $if(x > y + 1)$:..
>
> Then: at label 5 we will get the same symbolic store σ_s.
>
> But we will get an updated path constraint: $pct = x_0 > 10 \wedge x_0 > y_0 + 1$

- Explain what would you change and why, in order to make the symbolic execution of bounded_gcd to be an underapproximation of gcd : use new semantic expression that sets the path constraint to false whenever we do not want it to contiue.

Alloy

Set Interpretation

```
sig S extends E {
      F: one T
  }

  fact {
      all s:S | s.F in T
  }
```

- s is a set
- s is a subset of E
- F is a relation which maps each s to exactly one T
- s is an element of S
- s.F composes the unary relation s with the binary relation F, returning the a unary realtion of type T

`sig FSObject {parent: lone Dir}` : set FSObject, relation parent which relates FSObjects to DIrs, at most one parent dir for each FSObejct.

Sig Declaration

```
sig A,B extends C // A and B are disjoint
sig A, B in C // not necessarily disjoint
one sig A // enforces that has to always be exactly one instance of that
element
```

=> For example if you want an exact number n of objects then use "one" keyword in combination with "extends" and not "in", since extends symbolizes disjoint sets. => uniqueness.

Fact Declaration

```
fact {File + Dir = FSObject} // all system objects are either files or
directories
fact {
    FSObject in Root.*contents // the set of all fiule sysdtem objects is
a subset of everything reachable from the Root by following the contents
relation zero or more times
  }
```

Assert Declaration

```
 assert acyclic {
    no d: Dir | d in d.^contents
  }
  //the contents path is acyclic
```

=> an assert claims that something must be true due to the behaviour of the model

=> checked using `check` keyword, where Alloy either finds a counter example to the assertion or doesn't

Ternary Relations

```
 // A File System
  sig FileSystem {
    root: Dir,
    live: set FSObject,
    contents: Dir lone-> FSObject,
    parent: FSObject ->lone Dir
  }
  {
  // root has no parent
  no root.parent
  // live objects are reachable from the root
  live in root.*contents
  // parent is the inverse of contents
  parent = ~contents
}

contents in live -> live
// contents is in the set of the relation from live objects to live
objects
```

- Each file system is realted to exactly one directory, the root.
- the live relation rlates each file system to the set of file system objects it contains
- the set keyword allows the contents relation to relate FileSystem to any number of file system objects.
  => without means mapping only one-to-one !!
- the contents relation maps each file system to a binary relation from directories to file system objects.
- parents relation relates each file system to file system objects to directories

Run

```
  run example for exactly 1 FileSystem, 4 FSObject
  //alloy will try to find a solution to the model in which there is exactly 1 Filesystem and 4 FileSystemObjects
```

Advanced

```
  root: Dir & live, //intersection
  parent: (live - root) ->one (Dir & live),
```

=> the root of a FileSystem is in the set of live objects of its FileSystem and not just any directory.

=> the parent relation maps every live object except the root to exactly one live Dir.

Alloy Model

```
1. pred move [fs, fs': FileSystem, x: FSObject, d: Dir] {
2.     (x + d) in fs.live
3.     fs'.parent = fs.parent - x->(x.(fs.parent)) + x->d
4. }
```

=> the predicate move is true if file system fs' is the result of moving file system object x to directory d in file system fs.

- Line 2: the object to be moved and the destination directory of the move must both exist in the prestate file system.
- Line 3: the parent relation in the post state is the same as the prestate except the mapping from x to x's old parent is replaced by the mapping from x to x's nwew parent d.

```
pred removeAll [fs, fs': FileSystem, x: FSObject] {
  x in (fs.live - fs.root)
  let subtree = x.*(fs.contents) | fs'.parent = fs.parent - subtree->(subtree.(fs.parent))
}
```

- 2: the file system object to be deleted, x , must b in the prestate file system, fs, but that it cannot be the root of fs.
- 3: a let statement acts as a macro replacing the right side of the assignement whenever the left side of the assignement appears.
-

Advanced Checking

```
moveOkay: check {
   all fs, fs': FileSystem, x: FSObject, d:Dir |
     move[fs, fs', x, d] => fs'.live = fs.live
 } for 5
```

=> claims that the move operation does not alter the set of objects in the file system