- DYNAMIC TESTING
  - ↳ exploring a subset of a program behavior
  - ⊕ : no false positives
  - ⊖ : can have false negatives : miss bugs.

  Concerns:
  - build process - how to get a binary / APK ... to run
  - inputs - figuring out what inputs to give the main method (valid inputs)

STATIC ANALYSIS → need for over-approximation
- ↳ consider all program behaviours
- ⊕ no false negatives
- ⊖ too many positives

Concerns:
- to build analyzers you need abstract transformers: define the effect of the statements on the abstract d.
- Performance VS Precision Tradeoff
- Soundness / Precision

SYMBOLIC EXECUTION / CONCOLIC EXECUTION

⊕ no false positives

⊖ we need to bound loops + reliance on SMT solvers (false n.).

- Dynamic Analysis: many custom analyses (data race defec.)

↳ amplify testing + can combine w/ current test suite ⇒ try to learn from a single execution

ALLOY - MODELLING

↳ based on relations you write models and Alloy bounds the size of these relations.

Then uses SAT solver to find satisfying assignement to SAT formula, then translates it back to a model of the Alloy specification

---

SUMMARY

INTRO

- Main Activities of Software Engineering:
  - find out requirements
  - design (*)
  - implementation
  - validation.

overlap and treated iteratively.

(*) ⇒ can be divided into:

1. System Design: defines components (dbs, layers) and connectors (events...).
2. Detailed Design: chooses among different ways to implement the system design.

I) REQUIREMENTS

:= feature that a system must have to be accepted by the client.

the "what", not the "how".

1. Functional Requirements: what is the software supposed to do + external interfaces.

2. Non-functional Reqs: performance., quality, design.

* Main Reqs: - Clarity - Realism - Verifiability.
  - correctness - Completeness - Consistency ....

ACTIVITIES

* Identifying Actors:
  ⇒ which user groups are supported by the system..
* Identifying Scenarios:(1)
  ⇒ what are the tasks needed from the system.

(1) ⇒ narrative description of what people do as they make use of a program.

* Identifying Use Cases:(2)
  ⇒ unique name, flow of events, conditions, exceptions...

(2) → list of steps describing the interaction btw actor and system to achieve a goal.

~~H~~) MODELING

CODE DOCUMENTATION

(not important for exam.)

Informal Models

* Unified Modelling Language : text + graphical notation.

1. Static Models :=

⇒ class = state + behaviour (only class name is mandatory).

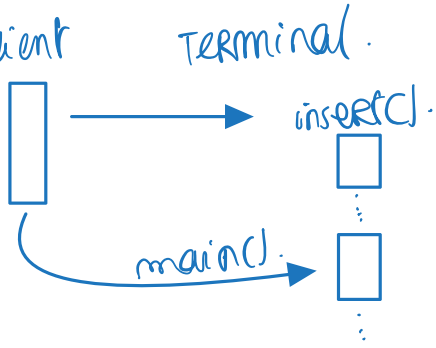
⇒ instance : name is underlined, attributes are represented w/ their values.

⇒ Associations btw classes denote relationships (can be directed)

2. Dynamic Models:

⇒ sequence diagrams[1] represent collaboration btw objects, state diagrams[2] the lifetime of a single object.

(1) →



(2) : abstractions of the attribute values of an object.

* Contracts := diagrams are not detailed enough to express certain properties ⇒ object constraint Language used to specify invariants & conditions.

FORMAL MODELS

* Alloy: formal modelling language based on set theory.
  - specifies a set of constraints that describe a set of structures.
  - everything is a relationship, analysis is done by SAT solver.

- Atoms := primitive entities, indivisible & immutable.
- Sets := unary relations
- Constants := - none - univ - iden
- Operations := + union / & intersection / - difference / in subset
  = equality / -> cross product.

- Most important OP: dot join := given atom tuples $s_1 \to \dots \to s_n$, $t_1 \to \dots \to t_m$, check if t_1 s_1 match, if yes return concat $(s_1, \dots, s_{n-1}, t_2, \dots, t_m)$ else return empty

- [] : same as dot join but binds less : a.b.c.[d] = d.(a.b.c).
- transpose ~R : reverses order of atoms.
- p++q : is equivalent to union but tuples of q can replace tuples of p.
- Quantifiers := - all x : e | F = F holds $\forall x$ in e.
  - some - no - lone (at most one).

⇒ Alloy has two types of logic:

- Predicate logic: all p: Person, w: Winner | p → w in loves
- Relational Calculus: Person → Winner in loves.

* Language / Static Models:

⇒ signature := declares a set of atoms

⇒ fact := records a constraint that is assumed to always hold.

⇒ Functions := define reusable, parametrized expressions
↳ make model easier to read. (= same as pred).

⇒ run <c> for <n> := instructs the analyzer to find a solution to c with n being the the max # of elements in each top-level set of the solution universe.

* Dynamic Models

⇒ Alloy has no built-in model of execution., hence op's are described declaratively by relating the atoms before & after the op.

⇒ instead of expressing invariants as facts, they can be asserted as properties maintained by the operations.

* Analyzing Models:

↳ an alloy model specifies a collection of constraints C that describe a set of structures.

⇒ A formula F is consistent (=satisfiable) : $\exists s\,(C(s) \wedge F(s))$.

⇒ F is valid : $\forall s\,(C(s) \Rightarrow F(s))$.

⇒ validity and consistence checking is undecidable

4) MODULARITY

* Benefits of Decomposition:=
  - partition the effort
  - independent testing of components
  - reuse of parts (...).

COUPLING

Coupling := measures interdependence btw different modules.

the lower the better, but tradeoffs:
- cohesion of modules
- performance
- adaptability
- code duplication.

* Data Coupling:

⇒ Causes: operations on shared datastructures ...

⇒ solutions:

• Restricting access to data → implemented in the facade pattern (simple interface).

- Making shared data immutable: flyweight pattern
  ↳ drawback: copying takes more time + memory.
- Avoiding shared data: pipe-and-filter pattern
  ↳ data-flow is the only communication.

* Procedural Coupling: when a module calls a method of another module.
  ↳ problem: the callers cannot be reused w/o callees.

⇒ Solutions:

- Moving Code
- Event-based Style := components may generate events and register for events in other components as a callback
  ↳ Observer Pattern
  ↳ Advantages: strong support for reuse + adaptability.
  ↳ Disadvantages: loss of control + correctness

- Restricting Calls
  ↳ Advantages: more abstraction + less maintenance + reuse.

* **Class Coupling**: changing superclass may break subclass.
  ⇒ Solutions:
  - Replacing Inheritance wi Aggregation
  - Using Interfaces: occurences of classnames are replaced by supertypes.
  - Delegating Allocations

ADAPTATION

* **Parametrization**:
  ⇒ ways to make modules parametric:
  - the values they manipulate.
  - the datastrucs they operate on
  - the types
  - the algos.

  → **Strategy Patteen** = lets you define a family of algos, put each one into a separate class and make their objects interchangeable.

* **Specialization**:
  → **Dynamic Method Binding**: adding / removing cases (method overrides) does not require changes in the caller.
  → Drawbacks:
  - subclasses share responsibility for maintaining invariants
  - Testing: more possible scenarios.

- Versioning.
- Performance.

→ **State Pattern**: lets an object alter its behavior when its internal state changes

IV) Static Program Analysis

Basics

⇒ we can only build an analyzer that returns "no" when it is unknown if the property holds and when it returns "yes" it holds for certain.

⇒ static analysis := over-approximation of all possible program behaviors.

⇒ General steps:

1. Select / define abstract domain
2. Define abstract semantics for the language & prove soundness
3. Iteratre abstract transformers over the abstract domain until fix point is reached.

Abstract Interpretation

* **Structures**
  - ⇒ Partial Order := { reflexive, transitive, antisymmetric. } ⇒ poset.
  - ⇒ Least / Greatest Element := $\forall p \in L: \bot \sqsubseteq p$. / $\forall p \in L: p \sqsubseteq \top$.
  - ⇒ Upper(u)/ Lower Bound(l) := $\forall p \in Y: p \sqsubseteq u$ / $\forall p \in Y: l \sqsubseteq p$.
  - ⇒ Join : $A \sqcup B$ / Meet : $A \sqcap B$.

  - ⇒ Domains := non-relational domain that doesn't keep the relationship between the variables.

* **Functions**
  - ⇒ increasing := $\forall a, b \in A : a \sqsubseteq b \Rightarrow f(a) \sqsubseteq f(b)$.
  - ⇒ x is a fixed point $\Leftrightarrow f(x) = x$.
  - ⇒ x is a post-fix point $\Leftrightarrow f(x) \sqsubseteq x$.
  - ⇒ $lfp^{\sqsubseteq} f$ is a least fixed point $\Leftrightarrow$ { $lfp^{\sqsubseteq} f$ is a fixed point; $\forall a \in L: a = f(a) \Rightarrow lfp^{\sqsubseteq} f \sqsubseteq a$. }

  - ⇒ Tarsk's Fixed Point Theorem := $(L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ a complete lattice, f a monotone function ⇒ $lfp^{\sqsubseteq} f$ exists and $lfp^{\sqsubseteq} f = \sqcap Red(F) \in Fix(F)$.

* **Approximating Functions:**
  - $[\![P]\!]$:= the set of reachable states of a program P.
    - ↳ $F(S) = I \cup \{c' \mid c \in S \wedge c \rightarrow c'\}$.
      $F([\![P]\!]) = [\![P]\!]$ is the lfp of F.

- we want to define $F^\#$ s.t: $F^\#$ approximates F:
given $F: C \to C$, $F^\#: C \to C$, $F^\#$ appr. $F \iff \forall x \in C: F(x) \sqsubseteq_C F^\#(x)$

- **Lear Fixed Point Theorem**:

Premises:
1. Monotone functions $F: C \to C$, $F^\#: A \to A$
2. $\alpha: C \to A \wedge \gamma: A \to C$ forming a Galois connection.
3. $\forall z \in A: \alpha(F(\gamma(z))) \sqsubseteq_A F^\#(z)$.

THEN: $\alpha(lfp(F)) \sqsubseteq_A lfp(F^\#)$

INTERVAL ANALYSES

- **Interval Domain** := abstracts the set of states into a map which captures the range of values that a var can take

$L^i = \{[x,y] \mid x,y \in \mathbb{Z}^\infty, x \le y\} \cup \{\bot_i\}$, $\mathbb{Z}^\infty = \mathbb{Z} \cup \{-\infty, \infty\}$:

→ $[a,b] \sqsubseteq [c,d]$ if $c \le a \wedge b \le d$

→ $[a,b] \sqcup_i [c,d] = [\min(\{a,c\}), \max(\{b,d\})]$

→ $[a,b] \sqcap_i [c,d] =$ meet $(\max(\{a,c\}), \min(\{b,d\}))$.

↳ returns $[a,b]$ if $a \le b$, else $\bot_i$.

- **Abstract Transformers**:

- $[\![x := a]\!]_i(m) = m[x \mapsto v]$, where $\langle a, m \rangle \Downarrow_i v$ (= given map m, expression a evaluates to $v \in L^i$).
- $[\![b]\!]_i(m)$ for a boolean expression b
- widening operator: $\nabla_i: L^i \times L^i \to L^i$. $[a,b] \nabla_i [c,d] = [e,f]$

where: if $c < a$ then $e = -\infty$, else $e = a$
if $d > b$ then $f = \infty$, else $f = b$.

Pointer Analysis

- Two pointers p and q are aliases if they point to the same object
- Because a program can create ∞ number of objects we need to use abstraction: heap is divided into fixed partition
  ⇒ all objects allocated at the same program point get represented by a single "abstract object".

- Pointer Analysis can be **flow-sensitive** (= respects the program control flow): ∃ separate set of points-to-pair for every program point.
  ⇒ no widening needed

- Flow-insensitive PA: we assume all execution orders are possible.

Proving Determinism

⇒ proving det. of arbitrary program is hard, but **conflict-freedom** is easier and implies det.

Symbolic Execution

→ over- and under- approximation of the program behavior
→ goal: explore as many program executions as possible, but returns false negatives.

- Principle: a symbolic value is associated w/ each variable instead of a concrete value, then the program is run w/ the values, obtaining a constraint formula.
  At any point a constraint solver can be used to find satisfying assignements, which can be used get concrete inputs for which the program reaches a certain point.

⇒ cannot handle unbounded loops.

Concolic Execution

→ tries to solve the problem that SMT solvers cannot always find satisfying assignements.

:= a search-based technique w/ the goal of improving branch coverage in testing

- Principle: the program runs as usual but also maintains the usual symbolic info.

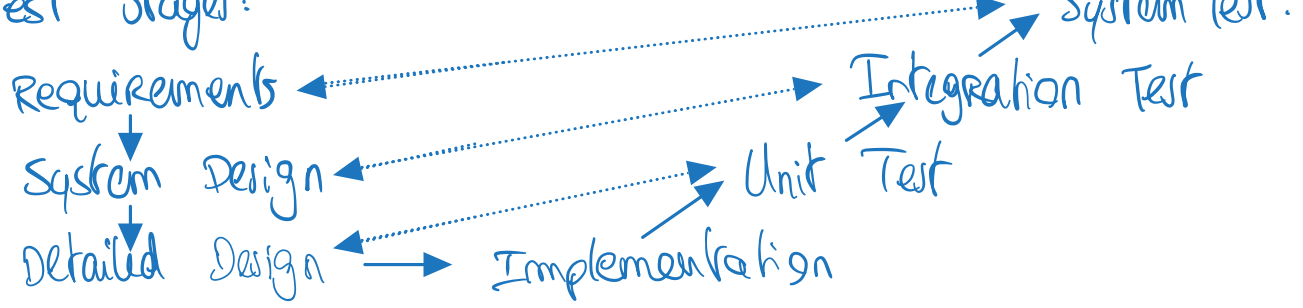
Whenever the concrete execution needs to obtain new inputs, it uses the symbolic info to obtain new inputs.

IV) TESTING

BASICS

* 3 Methods to increase **software reliability**:
  1. **Fault Avoidance** := detect faults statically w/o executing the program
  2. **Fault Detection** := (includes testing) execute program.
  3. **Fault Tolerance** := recover from faults at runtime (including redundancy ...).

* Test Stages:



- **Unit Testing** := individual subsystems are tested
- **Integration Testing** := groups of subsystems are tested
- **System Testing** := whole system.
  - Functional Testing
  - Performance Testing
  - Acceptance (clients understanding)
  - Installation (user env.).

TEST STRATEGIES

1. Exhaustive Testing := check all possible inputs
2. Random Testing := (☹ = treats all inputs as equally valuable).
3. Functional Testing := requirements knowledge is used to determine test cases.
   - goal: cover all requirements
4. Structural Testing := design knowledge about structure, algos and data struct is used.
   - goal: cover all the code

(V) FUNCTIONAL TESTING

- input is divided into equivalence classes, the goal is to have same classes w/ a higher density of failures.

SELECTING REPRESENTATIVE VALUES

⇒ if the inputs are from a range of valid inputs, use values below, within and above the range.

⇒ if they are from a discrete set of valid values; use want valid/invalid discrete vals.

COMBINATORIAL TESTING

* **Semantic Constraints:** use problem domain knowledge to remove unnecessary combinations.
* **Combinatorial Selection:** focus on all possible combinations of each pair of inputs.

* Random Selection.

VII) Structural Testing

⇒ detailed design / coding introduce many behaviors which are not tested by func. testing.

Control Flow Testing

* basic block := sequence of statements such that the code inside has one entry point (= no code within it is the destination of a jump) and one exit point. (= bb).

* Intraprocedural Control Flow Graph: contains:
  - edge from a to b w/ condition c ⇔ execution of bb a is succeeded by bb b if c holds
  - edge (entry, a, true) if a is the 1st bb.
  - edge (b, exit, true) for each bb b that ends w/ a return statement.

* Coverage Metrics:
  - Statement coverage := $\frac{\#\text{ of executed statements}}{\#\text{ of statements}}$

  - Branch Coverage := edge (m, n, c) is a branch ⇔ ∄ edge (m, n', c) with n ≠ n'
    := $\frac{\#\text{ of executed branches}}{\#\text{ of branches}}$.

→ complete B. coverage ⇒ complete statement coverage

- **Path Coverage** := $\frac{\#\text{ of executed paths}}{\#\text{ of paths.}}$
  - path = sequence of nodes $n_1 \dots n_k$ s.t. n_1 = entry, n_k = exit ∧ ∃ edge (n_i, n_{i+1}, c) in CFG.
  - complete p. coverage ⇒ b. coverage ∧ s. coverage.

- **Loop Coverage** := $\frac{\#\text{ of executed loops w/ 0, 1, more iterat°}}{(\#\text{ of loops}) \cdot 3.}$
  - 100% := execute each loop exactly 0, once OR more than once.

- **Soundness**: if ∃ transition $t = (\ell', \sigma') \to (\ell, \sigma)$ in a program trace in p; where t was performed by a, then (ℓ', a, ℓ) must exist in CFG ⇒ sound.
  - Formally: $\forall z \in L: \gamma(F^{\#}(z)) \supseteq f(\gamma(z))$.

- **Precision**: (ℓ', a, ℓ) may exist even if no transition t occurs ⇒ imprecision
  - Formally: $\forall z \in L: \gamma(F^{\#}(z)) \subseteq f(\gamma(z))$.

Data Race Detection

useful for event-driven applications, for which we want fast response-time, but they are asynchronous & have a complex control flow.

* **Data Race** := when we have a reachable program state where:
  - two outgoing transitions by 2 different threads
  - they access the same memory location.
  - at least one access is a write.

* **Modern Dynamic Race Detection** :=
  1. define memory locations on which races can happen
  2. define Happen-before model (op ordering).
  3. find algo to find all races
  4. filter harmless races
  5. implement algo + evaluate.

theorems that an analyzer should ensure:
- no false negatives
- no false positives.

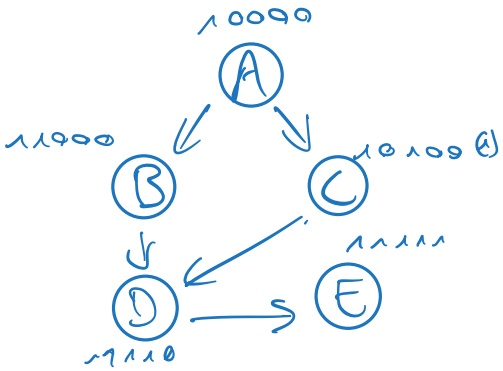
⇒ Challenges:

- synchronization done w\ read/writes → leads to many false races.
- massive # of event handlers → not enough space.

solution: only report races that are guaranteed to exist.

* Vector Clocks

(A): A can reach C: 1
B cannot: 0
C can: 1
(:)



solution: rediscover threads by partitioning the nodes into chains.

EXERCISES

* Use interval analysis to compute sound & non-trivial invariants at fix point.

Given:

```
void foo (int n):
  int i = 0, w = 0
while (i ≤ n):
  i = i+1, w = w+i
```

$n = [-\infty, \infty]$, $i = [-\infty, \infty]$, $w = [-\infty, \infty]$

$n = [-\infty, \infty]$, $i = [0, 0]$ $w = [0, 0]$.

$n = [0, \infty]$, $i = [0, \infty]$, $w = [0, \infty]$

$n = [0, \infty]$, $i = [1, \infty]$, $w = [1, \infty]$

$n = [-\infty, \infty]$, $i = [0, \infty]$, $w = [0, \infty]$

* Recreate program based on flow sensitive pointer analysis:
  - 1st handle field assignements
  - then handle pointer variables.

→ cannot be done using flow-insensitive analysis if two or more variables point to the same allocation site since they are indistinguishable.