

## 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 detection)
- simplify testing + can combine w/ current test suite  $\Rightarrow$  try to learn from a single execution

## ALLOY - MODELLING

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

Then user SAT solver to find satisfying assignment 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.

## 1) 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<sup>(1)</sup>:

→ 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<sup>(2)</sup>:

→ unique name, flow of events, conditions, exceptions...

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

## A MODELLING

## CODE DOCUMENTATION

(not important for exam)

# INFORMAL MODELS

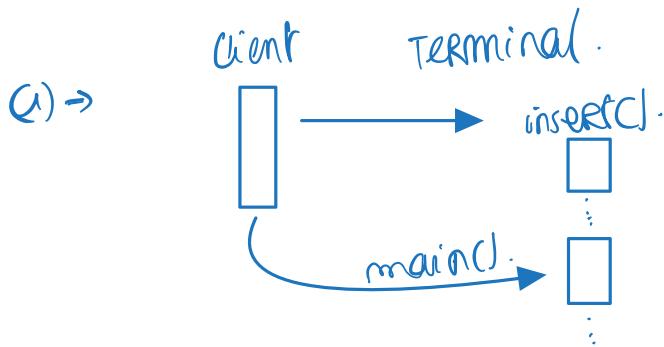
\* Unified Modelling Language : text + graphical notation.

## 1. Static Models :=

- class = state + behaviour (only class name is mandatory).
- ⇒ instance: name is underlined, attributes are represented w/ their values.
- ⇒ Associations b/w classes denote relationships (can be directed)

## 2. Dynamic Models:

- sequence diagrams<sup>(1)</sup> represent collaboration b/w objects, state diagrams<sup>(2)</sup> the lifetime of a single object.



(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 /  $\Rightarrow$  cross product.
- Most important op: **dot join** := given atom tuples  $s_1 \rightarrow \dots \rightarrow s_n$ ,  $t_1 \rightarrow \dots \rightarrow t_m$ , check if  $t_i$ ,  $s_i$  match, if yes return concat  $(s_i, j; - \{s_1, t_1\})$  else return empty
- []: same as dot join but binds less:  $a.b.c.[d] = d.(a.b.c)$ .
- transpose NR: reverse order of atoms.
- p+q: is equivalent to union but tuples of q can replace tuples of p.
- Quantifiers := - all  $\forall$ :  $e \models F \Leftrightarrow F$  holds  $\forall x$  in  $e$ .
  - some  $\exists$
  - no
  - one (at most one).

⇒ Alloy has two types of logic:

- Predicate logic: all  $p$ : Person,  $w$ : Winner |  $p \rightarrow w$  in loves
- Relational Calculus: Person  $\rightarrow$  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 max # of elements in each top-level set of the solution universe.

## \* Dynamic Models

⇒ Alloy has no built-in model of execution, hence ops 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 consistency checking is undecidable

## 7) MODULARITY

\* Benefits of Decomposition :=

- partition the effort
- independent testing of components
- reuse of parts (...).

### Coupling

Coupling := measures interdependence b/w different modules.

The lower the better, but tradeoffs:

- cohesion of modules
- performance
- adaptability
- code duplication.

### \* Data Coupling:

⇒ Caused: operations on shared datastructure ...

⇒ Solutions:

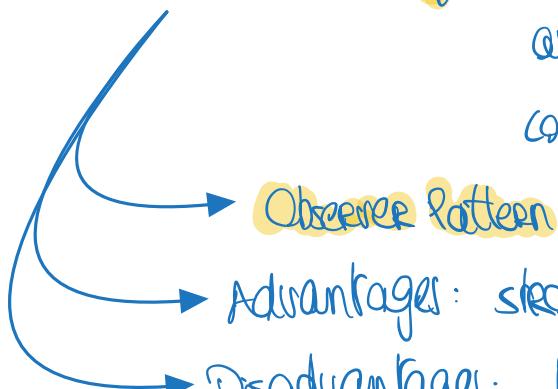
- Restricting access to data  $\Rightarrow$  implemented in the **Facade pattern** (simple interface).

- 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 w/ a callback



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

\* **Class Coupling**: changing superclass may break subclass.

⇒ Solutions:

- Replacing Inheritance w/ **Aggregation**
- Using **Interface**: occurrences of classnames are replaced by supertypes.
- Delegating Allocations

## ADAPTATION

\* **Parametrization**:

⇒ ways to make modules parametric:

- the values they manipulate.
- the datasets they operate on
- the types
- the algs.

→ **Strategy Pattern** = lets you define a family of algs, 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. Create abstract transformers over the abstract domain until fix point is reached.

# ABSTRACT INTERPRETATION

## \* Structures

$\Rightarrow$  Partial Order :=  $\begin{cases} \text{reflexive} \\ \text{transitive} \\ \text{antisymmetric} \end{cases} \Rightarrow$  poset.

$\Rightarrow$  Least / Greatest Element :=  $\forall p \in L : \perp \leq p . / \forall p \in L : p \leq \top .$

$\Rightarrow$  Upper(l)/ Lower Bound(e) :=  $\forall p \in Y : p \leq u . / \forall p \in Y : e \leq p .$

$\Rightarrow$  Join : A  $\sqcup$  B / Meet : A  $\sqcap$  B.

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

## \* Functions

$\Rightarrow$  increasing :=  $\forall a, b \in A : a \leq b \Rightarrow f(a) \leq f(b) .$

$\Rightarrow$  x is a fixed point  $\Leftrightarrow f(x) = x .$

$\Rightarrow$  x is a post-fix point  $\Leftrightarrow f(x) \leq x .$

$\Rightarrow$  If  $p^F$  is a least fixed point  $\Leftrightarrow \begin{cases} p^F \text{ is a fixed point} \\ \forall a \in L : a = f(a) \Rightarrow p^F \leq a . \end{cases}$

$\Rightarrow$  Tarski's Fixed Point Theorem :=  $(L, \leq, \sqcup, \sqcap, \perp, \top)$  a complete lattice, F a monotone function  $\Rightarrow$   $p^F$  exists and  $p^F = \text{Fix}(F) \subseteq \text{Fix}(f)$ .

## \* Approximating Functions:

•  $[EP]$  := the set of reachable states of a program P.

$\hookrightarrow F(S) = I \cup \{c' | c \in S \wedge c \xrightarrow{*} c'\} .$

$F([EP]) = [EP]$  is the FP of F.

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

### • Least Fixed Point Theorem:

Premises  $\left\{ \begin{array}{l} 1. \text{Monotone functions } F: C \rightarrow C, F^\#: A \rightarrow A \\ 2. \alpha: C \rightarrow A \wedge \gamma: A \rightarrow C \text{ forming a Galois connection.} \\ 3. \forall z \in A: \alpha(F(\gamma(z))) \subseteq_A F^\#(z). \end{array} \right.$   
THEN :  $\alpha(\text{lfp}(F)) \subseteq_A \text{lfp}(F^\#)$

### INTERVAL ANALYSES

- **Interval Domain** := abstracts the set of states into a map which captures the range of values that a var can take
  - $L^i = \{[x,y] \mid x, y \in \mathbb{Z}^\infty, x \leq y\} \cup \{\perp_i\}$ ,  $\mathbb{Z}^\infty = \mathbb{Z} \cup \{-\infty, \infty\}$ :
  - $[a,b] \subseteq [c,d]$  if  $c \leq a \wedge b \leq d$
  - $[a,b] \sqcup_i [c,d] = [\min\{a,c\}, \max\{b,d\}]$
  - $[a,b] \sqcap_i [c,d] = \text{meet}(\max\{a,c\}, \min\{b,d\})$ .
  - returns  $[a,b]$  if  $a \leq b$ , else  $\perp_i$ .

### • Abstract Transformers:

- $[\lambda x := a]_i(m) = m[x \mapsto r]$ , 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:  $\nabla_i: L^i \times L^i \rightarrow L^i$ .  $[a,b] \nabla_i [c,d] = [e,f]$   
where:  $\begin{cases} \text{if } c < a \text{ then } e = -\infty, \text{ else } e = a \\ \text{if } d > b \text{ then } f = \infty, \text{ else } f = b. \end{cases}$

## POINTER ANALYSIS

- Two pointers  $p$  and  $q$  are aliases if they point to the same object
- Because a program can create  $\infty$  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): a 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 return 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 assignments, which can be used get concrete inputs for which the program reaches a certain point.

⇒ cannot handle unbounded loops.

## CONCURRENCY EXECUTION

- tries to solve the problem that SMT solvers cannot always find satisfying assignments.

- = 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.

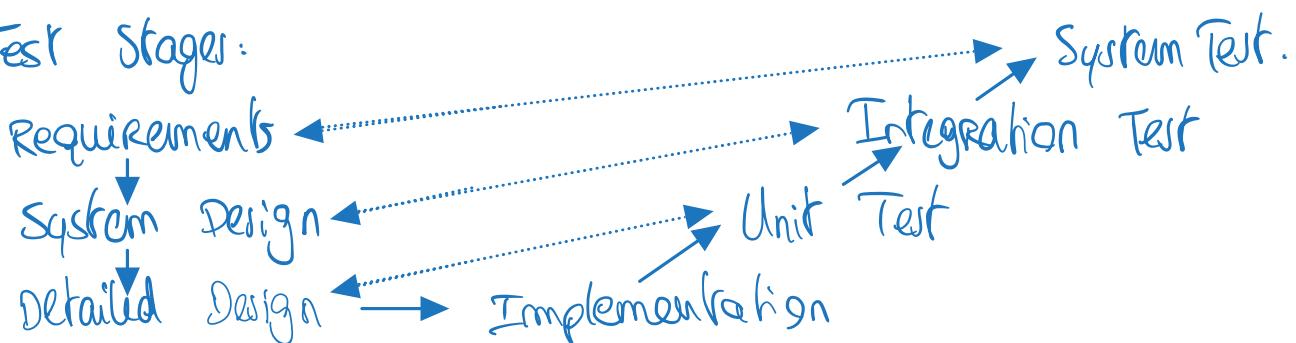
## IV) TESTING

### 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 := ( $\ominus$ ) = 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, algs and data struc is used.
    - goal: cover all the code

## FUNCTIONAL TESTING

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

## SELECTING REPRESENTATIVE VALUES

- ⇒ if the inputs are from a range of valid inputs, we want values below, within and above the range.
- ⇒ if they are from a discrete set of valid values ; we 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.

## IV) 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 bb w/ condition c  $\Leftrightarrow$  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

$\Leftrightarrow \exists \text{ edge } (m, n', c) \text{ with } n \neq n'$   
 $\Rightarrow := \frac{\text{# of executed branches}}{\text{# of branches}}$

complete B. coverage  $\Rightarrow$  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 &  $\exists$  edge  $(n_i, n_{i+1}, c)$  in CFG.  
→ complete p. coverage  $\Rightarrow$  b. coverage & s. coverage.

• Loop Coverage :=  $\frac{\# \text{ of executed loops w/ 0,1, more iterat}^*}{(\# \text{ of loops}) \cdot S}$

→ 100% := execute each loop exactly 0, once OR more than once.

• Soundness: if  $\exists$  transition  $t = (l', a') \rightarrow (l, a)$  in a program trace in  $P$ ; where  $t$  was performed by a, then  $(l', a, l)$  must exist in CFG  $\Rightarrow$  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  $\Rightarrow$  imprecision

Formally:  $\forall z \in L: \gamma(F^\#(z)) \subseteq f(\gamma(z))$ .

## A) 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.

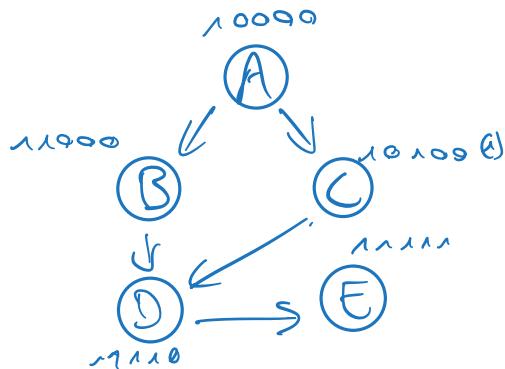
⇒ Challenges:

- 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

(1): { 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
```

$$\begin{aligned}n &= [-\infty, \infty], i = [-\infty, \infty], w = [-\infty, \infty] \\n &= [-\infty, \infty], i = [0, 0], w = [0, 0] \\n &= [0, \infty], i = [0, \infty], w = [0, \infty] \\n &= [0, \infty], i = [1, \infty], w = [1, \infty] \\n &= [\infty, \infty], i = [0, \infty], w = [0, \infty]\end{aligned}$$

- \* recreate program based on flow sensitive pointer analysis:

- 1st handle field assignments
- 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.