Automatic Testing & Verification

Recap

Juan Pablo Galeotti, Alessandra Gorla,
Software Engineering Chair – Computer Science
Saarland University, Germany

Feb. 7th : Exam

- 30% projects (10% each)
  - At least 50% threshold for exam admittance
  - Groups of 2
- 70% final exam (see course schedule)
  - Closed-book
  - Allowed: one A4 page (both sides!)

Verification, Validation, Synthesis, Inference

- Verification
  - Against a specification
    - It might be an implicit specification
- Validation
  - Does the system do what the user wants?
  - Failures in specifications
- Inference
  - Discover some interesting properties about the program
- Synthesis
  - Create a new program: optimize (compiler), control (scheduler)

We will focus on verification and inference

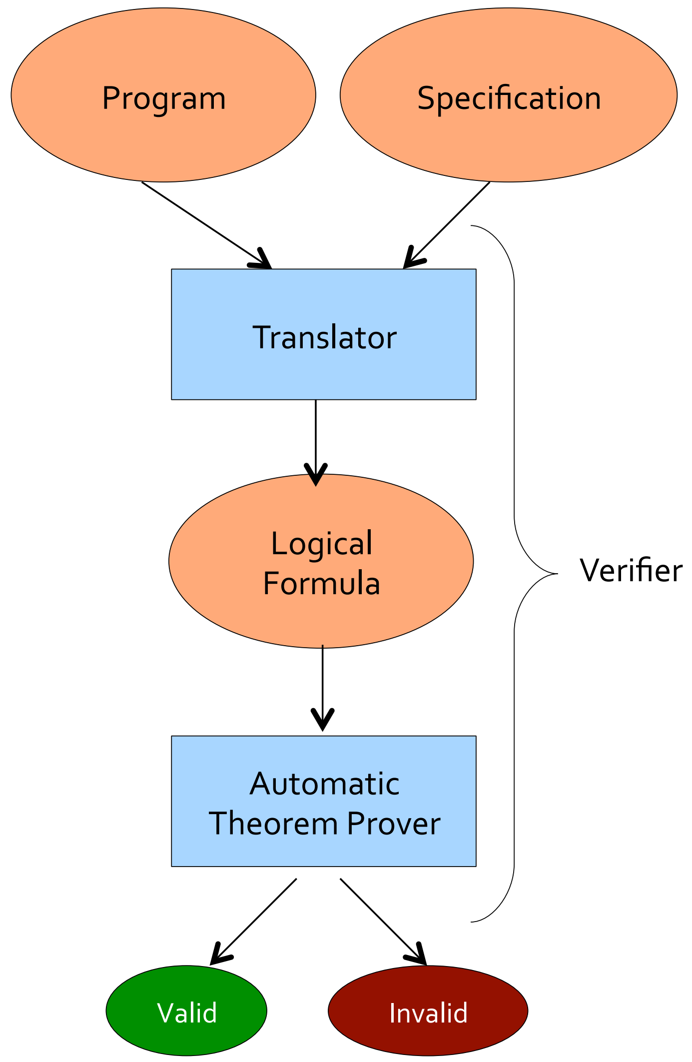
Programming with Contracts

Contract

A (formal) agreement between

Method M (callee)		**Callers of M**	
Rights	Responsabilities	Rights	Responsabilities

Verifying Programs

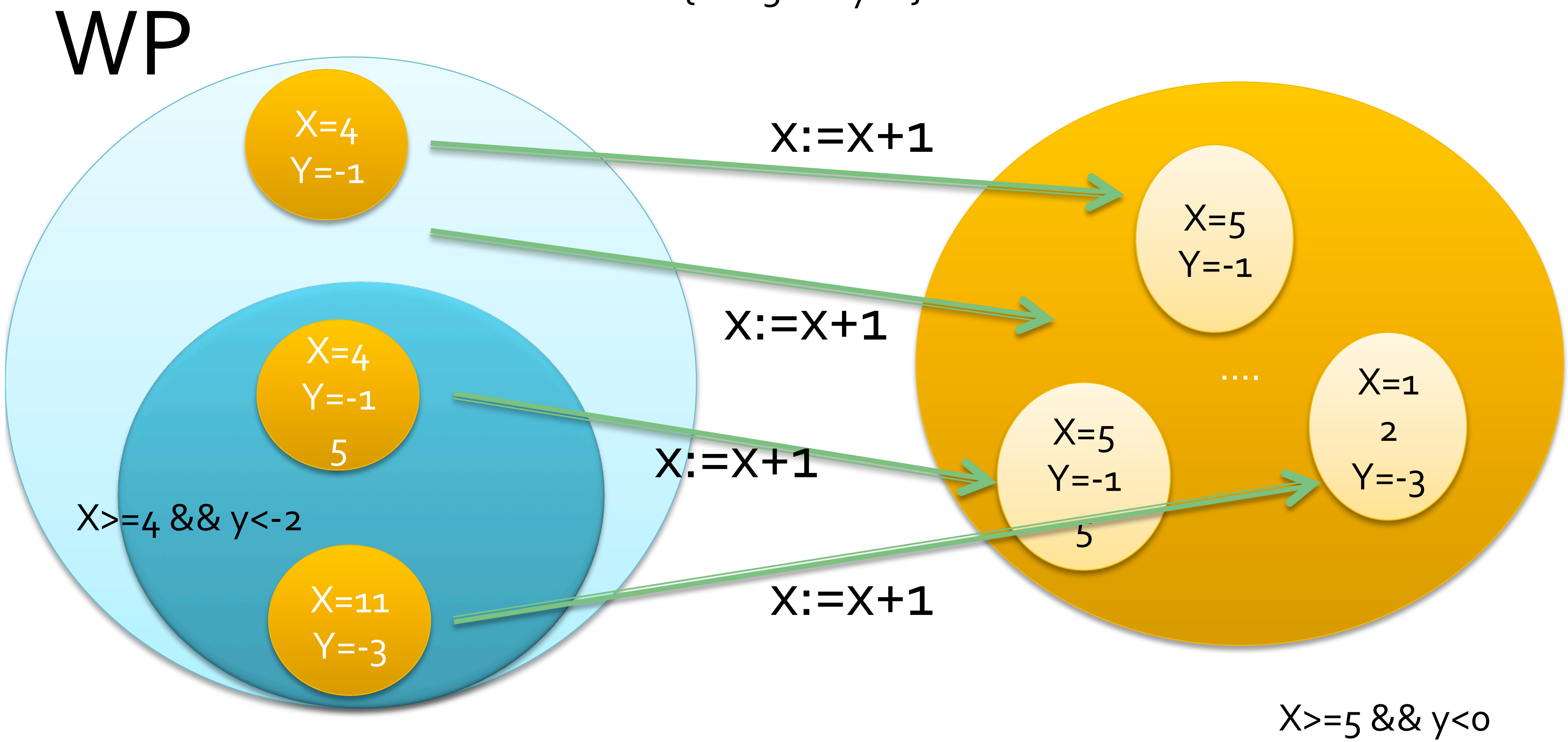


Some JML Annotations

- @requires
- @ensures
- @signals
- @normal_behavior/exceptional_behavior
- @assert/assume
- @assignable/pure
- @loop_invariant/decreases
- @ghost

Program states

{ x>= 4 && y<-2}
x := x +1
{ x>=5 && y<0}



Calculating the Weakest Precondition

- WP(`skip`, B) $=_{def}$ B
- WP(`x:=E`, B) $=_{def}$ B[x→E]
- WP(`s1;s2`, B) $=_{def}$ WP(s1, WP(s2, B))
- WP(`if(E){s1}else{s2}`, B) $=_{def}$

  E=> WP(s1,B) &&

  !E => WP(s2,B)

Exercise!

- Complete the following Hoare Triple with the weakest precondition:

```
{???}
While_(x>=0,x) x>0 do
    X:=x-1
    EndWhile
{x=0}
```

Problems with WP computation?

- **Loop iterations!**
- WP_k(`while(E) {S}`, B)
  - WP_0(…) $=_{def}$!E => B
  - WP_1(…) $=_{def}$!E => B && E => WP (S,B)
    = WP_0(…) && E => WP(S,B)
  - WP_2(…) $=_{def}$ WP_1(…) && E=>WP(S, WP_1(…))
  - ….
  - WP_i+1(…) $=_{def}$ WP_i && E=>WP(S,WP_i(…))

Dealing with loops

- Solutions:
  - **Unroll loops**: Verify a fixed set of execution traces
  - Add loop invariants to programs

Handling Loops

- We extend our WP definition for the new language constructs:
  - WP (havoc x, B) == \forall x. B
  - WP (assume E, B) == E=>B
  - WP (assert E, B) == E && B

Verifying Loops

- We transform loop code following this rule:

```
While_(I,T) E do S endwhile ==
        assert I
        havoc T
        assume I
        if (E) then
                S
                assert I
                assume false
        endif
```

Check Invariant hold at loop entry

Check loop body preservers Invariant

Object Invariant semantics

- An object invariant is a property that holds on every visible state of an object.
- What is a visible state?
  - The pre and post state of an invocation to a method of that object
- How to verify object invariants?

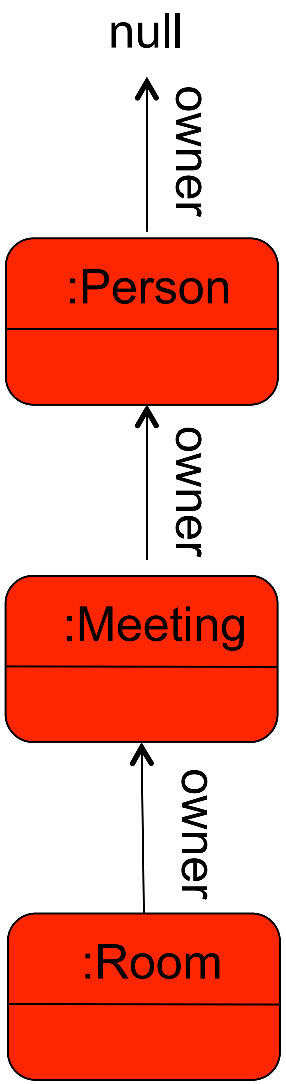
Modularity

- When we verify a method C.M() :
  - **Assume** that **ALL** invariants of all pre existitng objects hold at the method entry.
  - **Prove** that **ALL** invariants of all existing objects at the method exit hold
- When we invoke method C'.M'() from method C.M():
  - **Prove** that ALL invariants of all pre-existing objects hold before executing the method.
  - **Assume** ALL invariants of all existing objects hold a

But this semantics is not modular

Object invariants + ownership

- Object states:
  - Mutable
  - Valid
  - Commited
- Each object might have a single owner
  - Ownership is a acyclic relation
- In order to change a field value the object must be in mutable state
- In order to make the object valid all owned objects have to be in valid state.
- The Committed state acts as a lock



Dataflow Analysis

- Over approximates all program behaviors
- Abstract State of behavior
- Dataflow direction: forward vs. backward
- May analysis vs. Must Analysis

Direction\⊕	**∪ (MAY)**	**∩ (MUST)**
Forward	**reaching defs, zero analysis**	**available expressions**
Backward	**live variable analysis**	**very busy expressions**

(Forward) work-list algorithm

Compute out[*n*] ***for each n*** $\in$ ***N:***

out[*n*] *:=* $\perp$

work.add= {entry}

WHILE *work* is not empty:

n:= *work*.pop();

in'[*n*] *:=* $\oplus$ *{* ***out****[m] | m* $\in$ ***pred****(n) }*

out'[*n*] := ***transfer****[n](****in'****[n])*

IF !(***out'***[*n*] $\subseteq$ ***out****[n])*

for each m $\in$ ***succ****(n) work.add(m);*

out[*n*] *:=* ***out'***[*n*]*;* ***in***[*n*] *:=* ***in'***[*n*]*;*

Interprocedural Dataflow Analysis

- Analyze a program with many methods
- Strategies:
  - **Build an interprocedural CFG**
    - **Inlining/Cloning**
  - Assume/Guarantee
  - Context sensitivity
    - Inlining
    - Call string
    - Compute "summaries"

Dynamic Symbolic Execution

Choose next path

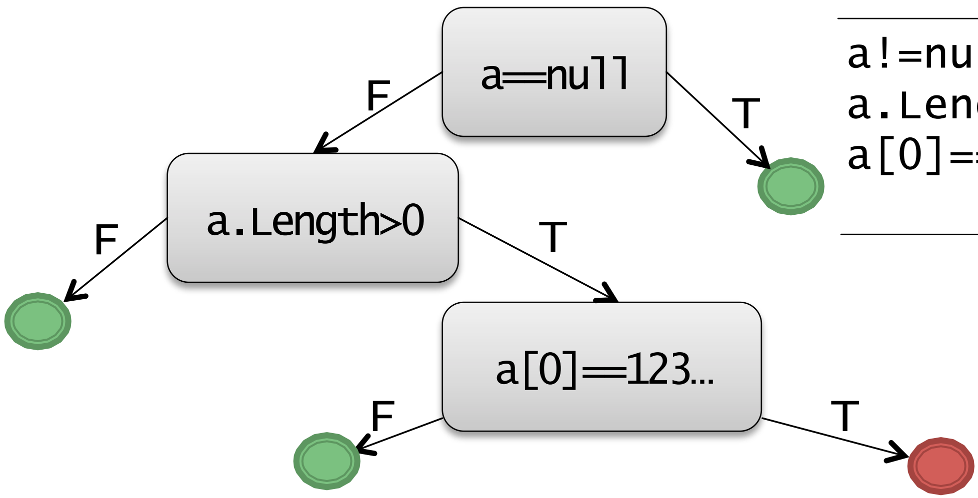
Solve → Execute&Monitor

Code to generate inputs for:

```
void CoverMe(int[] a)
{
  if (a == null) return;
  if (a.Length > 0)
    if (a[0] == 1234567890)
      throw new Exception("bug");
}
```

Constraints to solve	Data	Observed constraints
	null	a==null
a!=null	{}	a!=null && !(a.Length>0)
a!=null & [illegible] a.Length> [illegible]	[illegible]	[illegible] & [illegible] 90
a!=null && a.Length>0 && a[0]==1234567890	{123..}	a!=null && a.Length>0 && a[0]==1234567890

Negated condition



Done: There is no path left.

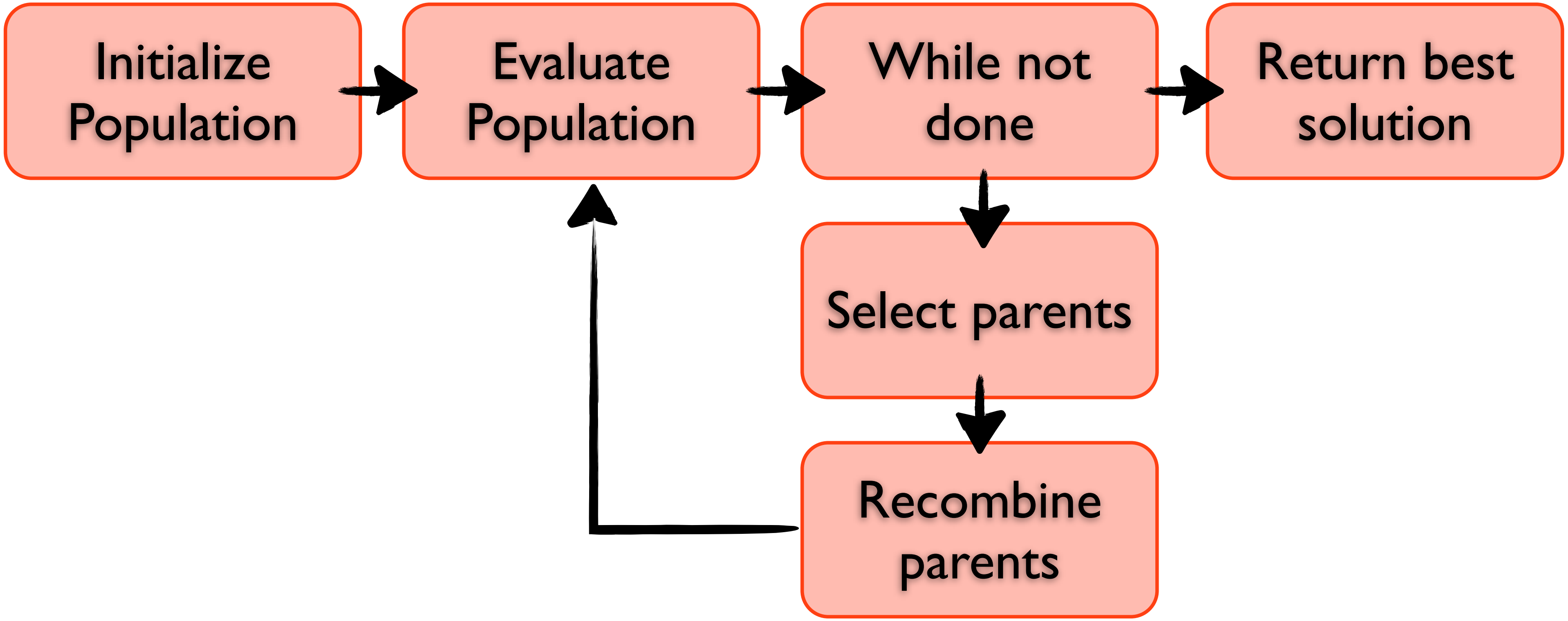
Random Testing

- Create program inputs randomly
- Observe if the program behaves “correctly”
  - Using explicit contracts (pre & posts)
  - Implicitly: runtime undeclared exceptions
- Advantages:
  - Easy to implement
  - Good coverage if the test suite is big enough

Exhaustive Testing - Idea

- Generate all non-isomorphic valid inputs up to a given size.
- Use programmatic contracts to decide if an input is valid.
- Prune search space efficiently.

Genetic Algorithms



Fitness

- Approach level
  - Number of control dependent edges between goal and chosen path
  - Approach = Number of dependent nodes - number of executed nodes
- Branch distance
  - Critical branch = branch where control flow diverged from reaching target
  - Distance to branch = distance to predicate being true / false

Some tools

- ESC/Java2, JMLForge
- Spec#
- Soot
- Javari/Plural
- Pex
- Korat
- EvoSuite