Software Testing

Wishnu Prasetya

Email: wishnu@cs.uu.nl
URL: www.cs.uu.nl/~wishnu

Content

- Chapter 2 of LN. Here I will just give you a summary of the chapter:
  - Coverage
  - White box testng
  - Black box testing
- Some addition: symbolic testing

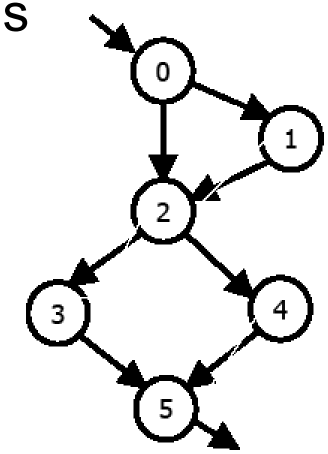
Testing vs Verification

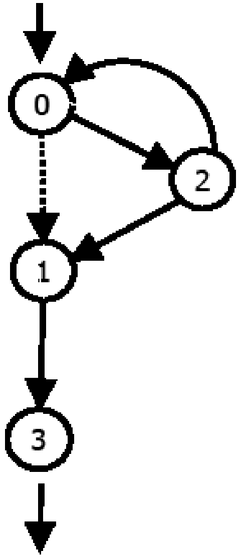
```
tax(income | tx) {
    tx = 0 ;
    if (income > 20000) then
       { tx := 0.2 * (income – 20000) ;  income := 20000 }
    if (income ≤ 10000)  then  tx := 0 ;
    else tx := tx + 0.1 * (income – 10000)
}
```

- *Verification* : show that *all possible executions* satisfy the given specification (very hard).
- *Testing* : show that at least few executions satisfy the specification.
  - not trivial to determine which executions to choose
  - not trivial to figure out how to trigger those executions (undecidable)

Coverage

- Introduce a “reasonable” *equivalence relation* over the executions, then try to “cover” all induced equivalence classes (an EC is covered if there is at least one test case’s whose execution belongs to it).
- Popular: CFG-based coverage:
  - Try to cover all nodes, or all edges
  - Try to cover all paths, or all prime paths, or all linearly independent paths





Automated testing

- Consider a program f(x|y), specified by {P} f(x,y) {Q}
- A test-case for f is an instance of x, satisfying P. The result y is checked if it satisfies Q.
- While inputs (instances of x) can be generated, no algorithm can guess what's in your mind. So, there is no way to "generate" P and Q.
- **Coverage problem**: come up with an instance of x, such that the resulting execution covers some target c, e.g. a certain branch.
  (recall that such a problem is undecidable)

From Unit Tetsing Tool Competition 2015

- Benchmarking of automated unit testing tools for Java; “unit testing” at the class level.
- 63 classes from various open source projects
- Base lines: randoop (random testing tool), human testers.
- Tools → all exploit reflection
  - T3 (random + pair-wise)
  - GRT (guided random + light static analysis)
  - JTexpert (guided random + light static analysis)
  - Mosa (evolutionary algorithm + light static analysis)
  - Evosuite (evolutionary algorithm + light static analysis + memetic)
  - Commercial (secret)

From Unit Tetsing Tool Competition 2015

	Rando op	Human	T3	GRT	JTexpe rt	Mosa	Evosuit e	CT
cov_b	32	68	57	61	55	56	55	26
cov_m	20	50	41	45	31	39	41	12
Tgen	1.77h	23h	0.85h	4.6h	1.55h	1.48h	6.5h	1.01h

	tools	tools+humans
Average cov_i	78.0 %	84.9 %
Average cov_b	64.7 %	70.1 %
Average cov_m	60.3 %	69.4 %
# CUTs with $cov_b = 100\%$	6	7
# CUTs with $cov_b \geq 80\%$	31	34
CUTs with $cov_i \leq 10\%$	{ 43,45,49,61 }	{ 45 }
CUTs with $cov_i \leq 5\%$	{ 45,61 }	{ 45 }
SCORE	266.7	277.8

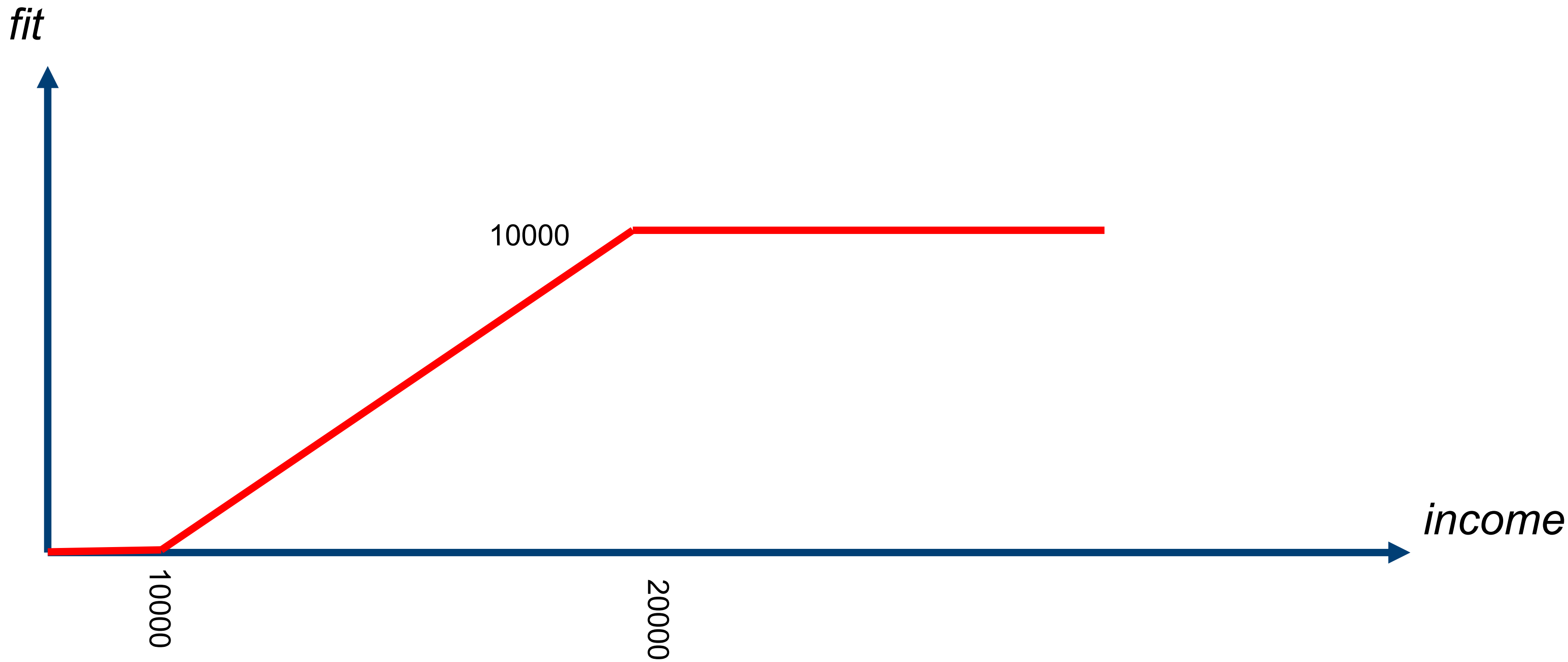
TABLE IV
COMBINED STRENGTH OF THE CONTESTING TOOLS

Cov. problem as a search problem

- Consider again the example program; ■ is the target to cover. The program is extended with "instrumentation" ; for simplicity, fit is a global-var.
- The problem can be re-expressed to searching an input for tax, such that the value of "fit" is minimized.

```
tax(income | tx) {
   tx := 0 ;
   if (income > 20000) then
      { tx := 0.2 * (income – 20000) ;  income := 20000 }
   fit := max(0,income - 10000)
   if (income ≤ 10000)  then ■ tx := 0 ;
   else tx := tx + 0.1 * (income – 10000)
}
```

Hill climbing



- (actually hill descent here)
  - start with some input vector i
  - adjust i, if this improves fit, repeat the process.
  - stop if no improvement is obtained.
- you can get stuck in a local-minimum
- other search-algorithgms, e.g. genetic

Using “symbolic execution”

- Consider $\{P\}\ \ f(x|y)\ \ \{Q\}$
- Consider a “progran path” ρ in $f(x)$
  - full: from start to end
  - you may require it to pass a certain “target branch” b
  - we convert all branching conditions along ρ to the corresponding **assert**, and in the “right orientation”.
  - let’s ignore loop and exception for now.
- Testing-1: any instance of x satisfying $P \wedge$ **wlp** ρ true will trigger an execution that covers b and ρ → solving the coverage problem.
- Testing-2: we can also use testing to check the validity of $P \Rightarrow$ **wlp** ρ Q. But we need to convert conditions to **assume** rather than assert.

Example

```
tax(income | tx) {
    tx := 0 ;
    if (income > 20000) then
       { tx := 0.2 * (income – 20000) ;  income := 20000 }
    if (income ≤ 10000)  then ■ tx := 0 ;
    else tx := tx + 0.1 * (income – 10000)
}
```

- Suppose pre-cond is “income ≥ 0”, and post-cond “tx<income”
- A program path ρ covering ■

```
tx := 0 ;
assert income ≤ 20000 ;
assert income ≤ 10000 ;
tx := 0
```

- **wlp** ρ true = income ≤ 10000 ∧ income ≤ 20000
- Testing-1 mode: find **a solution** of income ≥ 0 ∧ income ≤ 10000 ∧ income ≤ 20000 ; it triggers an execution that will cover the entire ρ, thus also ■

Example

- If we use **assume** to encode ρ we get:
  tx := 0 ;
  assume income ≤ 20000 ;
  assume income ≤ 10000 ;
  tx := 0
- calculating P ⇒ wlp Q we obtain :
  income ≥ 0 ⇒ income ≤ 10000 ⇒ income ≤ 20000 ⇒ 0<income
- The path ρ is **correct** if the above implication is **valid**. In Testing-2 mode we use testing to check this validity.
- The predicate it is not valid. Counter example: income = 0.

Issues with the wlp-based approach

- Such a path-based wlp approach can be expensive (a program may have exponentially many paths, even if we have no loop) → mitigation by merging some paths.
- Some paths may turn out to be infeasible (thus wasting our testing effort)
- What if the program contains loops?
  - you can do $[loop]^k$ or $\langle loop \rangle^k$ unrolling depending on your purpose.
    - For Testing-1 mode use $[loop]^k$
    - For Testing-2 mode use $\langle loop \rangle^k$
- What if the program contains calls to methods whose source code is unknown?

Single-assignment approach

- Given the following program path, we can convert it to a set of single-assignment-based contraints :

assert $x>10$
$tx := 0.2 * (x - 10)$
$x := 10$
assert $xy > 10$
$tx := tx + 0.1 * (xy - 10)$

→

assert $x>10$
$\wedge\ tx_1 = 0.2 * (x - 10)$
$\wedge\ x_1 = 10$
$\wedge$ **assert** $x_1 y > 10$
$\wedge\ tx_2 = tx_1 + 0.1 * (x_1 y - 10)$

Only the yellow part form the condition for fully traversing the path.

- The end product is equivalent to what you get through **wlp**.
- Solving the constraints gives you an input (x,y) such that the resulting execution fully traverses the path.

Combined concrete-symbolic testing

- A program path containing “back-box” method call.

assert $x > 10$	→	**assert** $x > 10$
$tx := 0.2 * (x – 10)$		$\wedge\ tx_1 = 0.2 * (x – 10)$
$x := f()$		$\wedge\ x_1 = ??$
assert $xy > 10$		$\wedge$ **assert** $x_1 y > 10$
$tx := tx + 0.1 * (xy – 10)$		$\wedge\ tx_2 = tx_1 + 0.1 * (x_1 y – 10)$

- It is not possible to solve the constraints because the assignment to x_1 is unknown.
- We can however combine this with concrete executions. If you manage to get an execution with an x that passes the first assert, the resulting x1 can be instrumented, and we can solve the remaining constraints to get the needed value of y.