



POLYTECH[®]
PARIS-SUD

Cycle Ingénieur – 2^{ème} année

Département Informatique

Verification and Validation

Part IV : System Test II

Burkhart Wolff

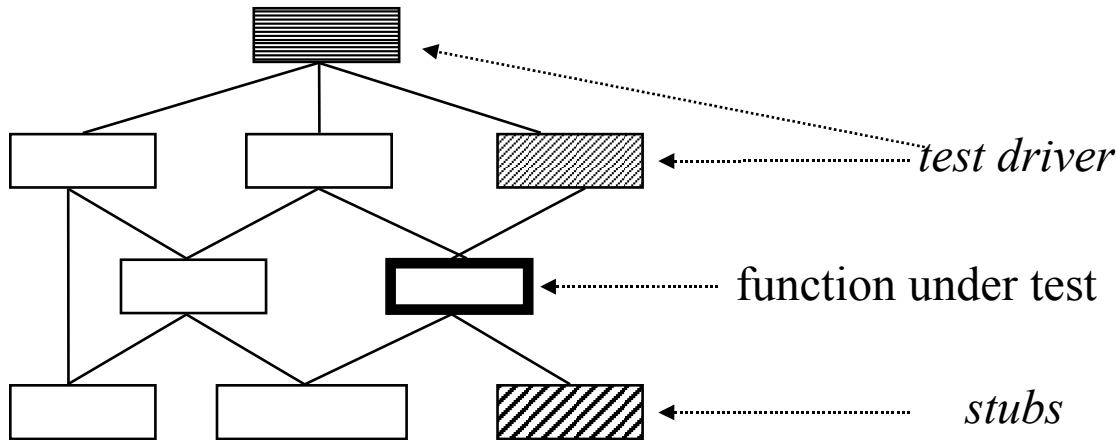
Département Informatique

Université Paris-Sud / Orsay

Towards **Static** Specification-based Unit Test

- ❑ How can this testing scenario be applied a priori (before deployment, at coding time, even at design-time ?)
- ❑ How can testing be applied to incomplete programs ?

Drivers and Stubs (Lanceurs et Bouchons)



« *How to test incomplete modules ?* »

- if non-existent, generate code for drivers:
may be a random-tester against pre-conditions
or a function running a pre-computed test-data-base.
- if non-existent, generate code for stubs: may be a
random-tester against post-conditions
or a "simple" version of the function to be computed.

Difficulties with Static Unit Tests so far

- ❑ The **generation of test-data** is left open; however, this is the CORE of the problem
- ❑ Setup of drivers and stubs necessary (but that can be automated ...)
- ❑ Random-testdata generators are usually **very ineffective** (why ???) ☹
writing „simple“ versions compliant to the post-conditions results in additional programming labour ... ☹

Difficulties with Static Unit Tests so far

- ❑ When do we have tested enough ?
When is our test “adequate” ?

We have to decide on adequacy criteria in advance.

This can be:

- criteria on the coverage of the spec of the program
 - criteria on statistical models and an error model
-
- ❑ Some empirical observations:
 - No relation between detection order and detection difficulty
 - No relation between detection difficulty and correction
 - The more errors you found, the more you find more...
 - The quality of a test set is independent of its size.

Functional Unit Test : Triangle Revisited

The specification in UML/OCL (Classes in USE Notation):

```
class Triangles inherits_from Shapes
  attributes
    a : Integer
    b : Integer
    c : Integer

  operations
    mk(Integer,Integer,Integer):Triangle
    is_Triangle(): triangle
end
```

Functional Unit Test : Triangle Revisited

The specification in UML/OCL (Classes in USE Notation):

context Triangles:

inv def : a.oclIsDefined() and b.oclIsDefined()...

inv pos : $0 < a$ and $0 < b$ and $0 < c$

inv triangle : $a + b > c$ and $b + c > a$ and $c + a > b$

context Triangle::isTriangle()

post equi : $a = b$ and $b = c$ implies result=equilateral

post iso : $((a <> b$ or $b <> c)$ and
 $(a = b$ or $b = c$ or $a = c))$ implies result=isosceles

post default: $(a <> b$ or $b <> c)$ and
 $(a <> b$ and $b <> c$ and $a <> c)$
implies result=arbitrary

Generating Test-Data by Example

- Consider the test specification:

`mk(x,y,z).isTriangle() ≡ X`

i.e. for which input (x,y,z) should an implementation of our contract yield which X ?

Note that we define `mk(0,0,0)` to `oclUndefined`, as well as all other invalid triangles ...

Intuitive Test-Data Generation

- ❑ an arbitrary valid triangle: (3, 4, 5)
- ❑ an equilateral triangle: (5, 5, 5)
- ❑ an isosceles triangle and its permutations :
(6, 6, 7), (7, 6, 6), (6, 7, 6)
- ❑ impossible triangles and their permutations :
(1, 2, 4), (4, 1, 2), (2, 4, 1) -- $x + y > z$
(1, 2, 3), (2, 4, 2), (5, 3, 2) -- $x + y = z$ (necessary?)
- ❑ a zero length : (0, 5, 4), (4, 0, 5),
- ❑ . . .
- ❑ Would we have to consider negative values?

Test-Data Generation

- ❑ Ouf, is there a systematic and automatic way to compute all these cases ?

Test-Data Generation

- ❑ Ouf, is there a systematic and automatic way to compute all these cases ?

Well, lets see and calculate ...

Test-Data Generation

- Recall the test specification:

$mk(x,y,z).isTriangle() \equiv X$

Test-Data Generation

- Recall the test specification:

$\text{mk}(x,y,z).\text{isTriangle}() \equiv X$

$\equiv (\text{mk}(x,y,z).\text{isTriangle}() \equiv \text{oclUndefined})$ or
 $(\text{mk}(x,y,z).\text{isTriangle}() \equiv \text{arbitrary})$ or
 $(\text{mk}(x,y,z).\text{isTriangle}() \equiv \text{isosceles})$ or
 $(\text{mk}(x,y,z).\text{isTriangle}() \equiv \text{equilateral})$

(*by case-split over variable X of type triangle*)

Test-Data Generation

- Recall the test specification:

$mk(x,y,z).isTriangle() \equiv X$

$\equiv (mk(x,y,z).isTriangle().oclIsUndefined())$ or
 $(mk(x,y,z).isTriangle() \equiv \text{arbitrary})$ or
 $(mk(x,y,z).isTriangle() \equiv \text{isosceles})$ or
 $(mk(x,y,z).isTriangle() \equiv \text{equilateral})$

(* by definition of `oclIsUndefined` *)

Test-Data Generation

- Recall the test specification:

`mk(x,y,z).isTriangle() ≡ X`

`≡ (mk(x,y,z).oclIsUndefined() and result=oclUndefined) or
let self = mk(x,y,z);
 a = self.a; b = self.b; c = self.c;
in self.oclIsDefined() and
 ((post(self,result) and result=arbitrary) or
 (post(self,result) and result=isosceles) or
 (post(self,result) and result=equilateral))
end
(*by post-condition of isTriangle*)`

Test-Data Generation

- Recall the test specification:

`mk(x,y,z).isTriangle() ≡ X`

`≡ (mk(x,y,z).oclIsUndefined() and result=oclUndefined) or
let self = mk(x,y,z);
a = self.a; b = self.b; c = self.c;
in self.oclIsDefined() and
((a<>b and b<>c and a<>c and result=arbitrary) or
(((a=b and a<>c and b<>c) or
(b=c and b<>a and c<>a) or
(a=c and a<>b and c<>b)) and result=isosceles) or
(a=b and b=c and result=equilateral))
end`

~~`(*by arbitrary<>isosceles<>equilateral and log. simplif. *)`~~

Test-Data Generation

- Recall the test specification:

`mk(x,y,z).isTriangle() ≡ X`

`≡ (mk(x,y,z).oclIsUndefined() and result=oclUndefined) or
(mk(x,y,z).oclIsDefined() and
let a = x; b = y; c = z;
in ((a<>b and b<>c and a<>c and result=arbitrary) or
((a=b and a<>c and b<>c and result=isosceles) or
(b=c and b<>a and c<>a and result=isosceles) or
(a=c and a<>b and c<>b and result=isosceles) or
(a=b and b=c and result=equilateral))
end
(*by mk(x,y,z).oclIsDefined() => mk(x,y,z).a=x, etc. *)`

Test-Data Generation

- Recall the test specification:

$\text{mk}(x,y,z).\text{isTriangle}() \equiv X$

$\equiv (\text{mk}(x,y,z).\text{oclIsUndefined}() \text{ and result}=\text{oclUndefined}) \text{ or}$
 $(\text{mk}(x,y,z).\text{oclIsDefined}() \text{ and}$
 $((x \langle \rangle y \text{ and } y \langle \rangle z \text{ and } x \langle \rangle z \text{ and result}=\text{arbitrary}) \text{ or}$
 $((x=y \text{ and } x \langle \rangle z \text{ and } y \langle \rangle z \text{ and result}=\text{isosceles}) \text{ or}$
 $(y=z \text{ and } y \langle \rangle x \text{ and } z \langle \rangle x \text{ and result}=\text{isosceles}) \text{ or}$
 $(x=z \text{ and } x \langle \rangle y \text{ and } z \langle \rangle y \text{ and result}=\text{isosceles}) \text{ or}$
 $(x=y \text{ and } y=z \text{ and result}=\text{equilateral})))$

(*by $\text{mk}(x,y,z).\text{oclIsDefined}() \Rightarrow \text{mk}(x,y,z).a=x$, etc. *)

Test-Data Generation

- Recall the test specification:

`mk(x,y,z).isTriangle() ≡ X`

```
≡ let inv = 0 < x and 0 < y and 0 < z
      and x + y > z and y + z > x and z + x > y
  in (not inv and result = oclUndefined) or
      (inv and x <> y and y <> z and x <> z and result = arbitrary) or
      (inv and x = y and x <> z and y <> z and result = isosceles) or
      (inv and y = z and y <> x and z <> x and result = isosceles) or
      (inv and x = z and x <> y and z <> y and result = isosceles) or
      (inv and x = y and y = z and result = equilateral)
  end
```

(*by invariant *)

Test-Data Generation

- Recall the test specification:

...

```
≡ let inv = 0 < x and 0 < y and 0 < z
      and x + y > z and y + z > x and z + x > y
```

```
in (x <= 0 and result = oclUndefined or
    y <= 0 and result = oclUndefined or
    z <= 0 and result = oclUndefined or
    z <= x + y and result = oclUndefined or
    x <= y + z and result = oclUndefined or
    y <= z + x and result = oclUndefined) or
    (inv and x <> y and y <> z and x <> z and result = arbitrary) or
    (inv and x = y and x <> z and y <> z and result = isosceles) or
    (inv and y = z and y <> x and z <> x and result = isosceles) or
    (inv and x = z and x <> y and z <> y and result = isosceles) or
    (inv and x = y and y = z and result = equilateral))
end
```

Test-Data Generation

- Now, we converted the entire specification into a Disjunctive Normal Form (DNF)

Test-Data Generation

- ❑ Now, we converted the entire specification into a Disjunctive Normal Form (DNF)
- ❑ Each Conjoint in the DNF is a **Test-Case**

Test-Data Generation

- Now, we converted the entire specification into a Disjunctive Normal Form (DNF)
- Each Conjoint in the DNF is a **Test-Case**
Example:

```
let inv = 0 < a and 0 < b and 0 < c
        and a + b > c and b + c > a and c + a > b
in inv and a <> b and b <> c and a <> c
    and result = arbitrary
```

Test-Data Generation

- ❑ Now, we converted the entire specification into a Disjunctive Normal Form (DNF)
- ❑ Each Conjoint in the DNF is a **Test-Case**
- ❑ **Test Data** is constructed by picking one instance of variables that makes the conjoint true !

Resulting Test (satisfying this formula !):

$\{a \mapsto 3, b \mapsto 4, c \mapsto 5, \text{result} \mapsto \text{arbitrary}\}$

Test-Data Generation

- ❑ Now, we converted the entire specification into a Disjunctive Normal Form (DNF)
- ❑ Each Conjoint in the DNF is a **Test-Case**
- ❑ **Test Data** is constructed by picking one instance of variables that makes the conjoint true !
- ❑ Test-Hypothesis applied here: **Uniformity hypothesis**

Test-Data Generation

□ Example Uniformity: Testcase

```
C(a,b,c,result) = let inv = 0<a and 0<b and 0<c
                    and a+b>c and b+c>a and c+a>b
                    in inv and a<>b and b<>c and a<>c
                    and result=arbitrary
```

Applied Uniformity Hypothesis:

$$\begin{aligned} &(\exists(a,b,c). C(a,b,c,result) \Rightarrow mk(a,b,c).isTriangle()=arbitrary) \\ &\Rightarrow (\forall(a,b,c). C(a,b,c,result) \Rightarrow mk(a,b,c).isTriangle()=arbitrary) \end{aligned}$$

Test-Data Generation

- General Scheme of **Uniformity hypothesis:**

Formally:

$$\begin{aligned} & (\exists x. \text{class } x \Rightarrow \text{TestSpec}(sut, x)) \\ & \Rightarrow (\forall x. \text{class } x \Rightarrow \text{TestSpec}(sut, x)) \end{aligned}$$

if there is a data in a test-class for which the system under test *sut* satisfies the test specification, *sut* will always satisfy it for data in this class.

Test-Data Generation

- General Scheme of **Uniformity hypothesis:**

Example: Testcase

```
let inv = 0 < a and 0 < b and 0 < c
        and a + b > c and b + c > a and c + a > b
in inv and a <> b and b <> c and a <> c
    and result = arbitrary
```

Resulting Test (satisfying this formula !):

{a ↦ 3, b ↦ 4, c ↦ 5, result ↦ arbitrary}

Test-Data Generation

- ❑ Test-Adequacy Criterion of this Generation Method:
(DNF Partition Adequacy)

All *test cases* result from DNF normalization of the TestSpec ... All Test-Cases must be covered by a test. *Test data* are constructed via application of Uniformity Hypothesis for the test case.

- ❑ Is there an automated procedure to do this?
Yes, DNF's are computable (but NP complete).
But there are also practical algorithms, so-called SMT-solvers, that can indeed be practically used for test-case generation (e.g. Z3, Alt-Ergo, ...).
- ❑ Surprisingly, Testing requires Theorem-Proving !!!

Test-Data Generation

- How to handle Recursion ?

Test-Data Generation

□ How to handle Recursion ?

In UML/OCL, recursion occurs (at least at two points:

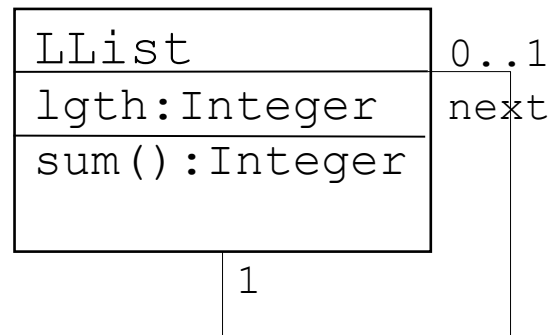
- at the level of data

Test-Data Generation

□ How to handle Recursion ?

In UML/OCL, recursion occurs (at least at two points:

- at the level of data



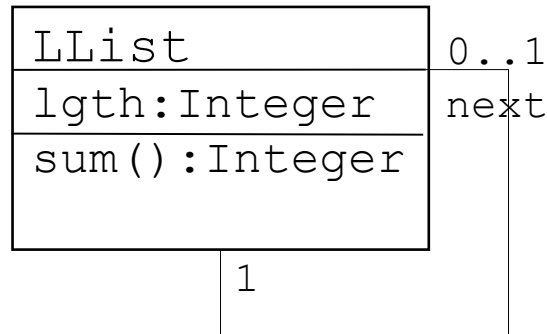
```
context LList:
inv null_or_valid: self.next.oclIsDefined()
inv length: lgth = if next = null
                        then 1
                        else next.lgth + 1
                        endif
```

Test-Data Generation

□ How to handle Recursion ?

In UML/OCL, recursion occurs (at least at two points:

- at the level of operations (post-conds may contain calls ...)



```
context LList:sum()
pre self <> null
post result =
    if self = null then 0
    else lgth + next.sum()
    endif
```

Test-Data Generation

- Answer:
apply **regularity hypothesis**.

$$\begin{aligned} & (\forall x. |x| < k \Rightarrow \text{TestSpec}(sut, x)) \\ & \Rightarrow (\forall x. \text{TestSpec}(sut, x)) \end{aligned}$$

if for all data up to a measure k the system under test *sut* satisfies the test specification, *sut* will always satisfy it

Measures are: length of lists, depth of trees, ...

Test-Data Generation

- Prerequisite: The constructor cons on lists:

```
context Void::cons(r:LList):LList
```

```
pre    True
```

```
post   result.oclIsNew() and
```

```
       result.lgth = if r=null then 0 else r.next+1 endif and
```

```
       result.next = r
```

```
context LList:
```

```
inv null_or_valid: self.next.oclIsDefined()
```

Test-Data Generation

- Consider the test specification:

$X.sum() \equiv Y$

$\equiv (\text{if } X=\text{null} \text{ then } 0 \text{ else } X.lgth+X.next.sum() \text{ endif} \equiv Y)$

(\star unfold sum() \star)

$\equiv (X=\text{null} \text{ and } 0 \equiv Y) \text{ or}$

$(X \neq \text{null} \text{ and } X.lgth+X.next.sum() \equiv Y)$

(\star logic if_then_else \star)

$\equiv (X=\text{null} \text{ and } 0 \equiv Y) \text{ or}$

$(X \neq \text{null} \text{ and } 1+X.next.lgth+X.next.sum() \equiv Y)$

(\star invariant length, simplification \star)

$\equiv (X=\text{null} \text{ and } 0 \equiv Y) \text{ or}$

$(X \neq \text{null} \text{ and } 1+X.next.lgth+$

$\text{if } X.next=\text{null} \text{ then } 0$

$\text{else } X.next.lgth+X.next.next.sum() \text{ endif} \equiv Y)$

Test-Data Generation

- Consider the test specification:

$X.sum() \equiv Y$

$\equiv (X=null \text{ and } 0 \equiv Y)$ or

$(X \langle \rangle null \text{ and } X.next=null \text{ and } 1 \equiv Y)$ or

$(X \langle \rangle null \text{ and } X.next \langle \rangle null \text{ and}$

$1+X.next.lgth+X.next.lgth+X.next.next.sum() \equiv Y)$

(★ logic if_then_else ★)

$\equiv (X=null \text{ and } 0 \equiv Y)$ or

$(X \langle \rangle null \text{ and } X.next=null \text{ and } 1 \equiv Y)$ or

$(X \langle \rangle null \text{ and } X.next \langle \rangle null \text{ and } X.next.next=null \text{ and } 3 \equiv Y)$

$(X \langle \rangle null \text{ and } X.next \langle \rangle null \text{ and } X.next.next \langle \rangle null \text{ and}$

$3+2*X.next.next.lgth+X.next.next.next.sum() \equiv Y)$

(★ same procedure as before ★)

Test-Data Generation

- Consider the test specification:

$$X.\text{sum}() \equiv Y$$

- From these test cases, we construct the test data:
 - $X=\text{null}, Y=0$
 - $X=\text{cons}(\text{null}), Y=1$
 - $X=\text{cons}(\text{cons}(\text{null})), Y=3$
 - ... all other cases ...
- **All other cases** were represented by the regularity hypothesis, i.e ...

Test-Data Generation

- **All other cases** were represented by the regularity hypothesis, i.e ...

$$\begin{aligned} (\forall X. |X| < 3 \Rightarrow X.\text{sum}() \equiv Y) \\ \Rightarrow (\forall X. X.\text{sum}() \equiv Y) \end{aligned}$$

where we choose as “complexity measure” just $X.\text{lgth}$!

So, whenever the program under test (here: $\text{sum}()$) passes the test for lists of length 2, we assume that it will always pass the test ...

Test-Data Generation

- ❑ Remarks on the regularity hypothesis:
 - main tool to reduce infinite number of test cases to a finite one (which may still contain infinitely many tests!)

 - is similar to an induction (corresponds to induction anchor, but leaves out the induction step)

Test-Data Generation

- ❑ Summary
 - We have (sketched) a symbolic Test-Case Generation Procedure for UML/OCL Specifications
 - It takes into account:
 - ❑ data invariants (recursive predicates)
 - ❑ recursive functions (via unfolding)
 - The process can be tool-supported (HOL-TestGen)
 - Doing the process by hand is quite tedious!

Test-Data Generation

□ Summary

Key-Ingredients are:

- Unfolding predicates up to a given depth k
- computing the Disjunctive Normal Form (DNF_k)
- Adequacy:
Pick for each test-case (a conjunct in the DNF_k)
one test, i.e. one substitution for the free
variables satisfying the test-case !

Test-Data Generation

□ Summary

Key-Ingredients are:

- Uniformity Hypothesis necessary to pick one element out of a test case
- Regularity Hypothesis necessary to justify that data space is only explored up to a certain depth
- Both are necessary in practice ...