L3 Mention Informatique
Parcours Informatique et MIAGE

Génie Logiciel Avancé

Part V : Black-Box Test

Burkhart Wolff
wolff@lri.fr

Towards Static Specification-based Unit Test

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

Difficulties with Static Unit Tests so far

- 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 : An Example

The specification in UML/MOAL:

```
Triangles
---------------------------------------------
a, b, c: Integer
---------------------------------------------
- mk(Integer,Integer,Integer):Triangle
- is_Triangle(): {equ (*equilateral*),
                  iso (*isosceles*),
                  arb (*arbitrary*)}
```

Functional Unit Test : An Example

Recall:

```
inv   0<a ∧ 0<b ∧ 0<c
inv   c≤a+b ∧ a≤b+c ∧ b≤c+a
```

Triangles

```
a, b, c: Integer
```

```
- mk(Integer,Integer,Integer):Triangle
- is_Triangle(): {equ (*equilateral*),
                  iso (*isosceles*),
                  arb (*arbitrary*)}
```

```
operation t.is_Triangle():
  pre   t ≠ null
  post  t.a=t.b ∧ t.b=t.c ⟶ result=equ
  post  (t.a≠t.b ∨ t.b≠t.c ∨ t.a≠t.c) ∧
        (t.a=t.b ∨ t.b=t.c ∨ t.a=t.c))⟶ result=iso
  post  (t.a≠t.b ∧ t.b≠t.c ∧ t.a≠t.c))⟶ result=arb
  post  modifiesOnly({})
```

Generating Test-Data by Example

- Consider the test specification (the "Test Goal"):

  mk(x,y,z).isTriangle() $\equiv$ 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 invalid, 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 isoscele 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 ?

Revision: Boolean Logic + Some Basic Rules

- $\neg(a \wedge b) = \neg a \vee \neg b$ (* deMorgan1 *)
- $\neg(a \vee b) = \neg a \wedge \neg b$ (* deMorgan2 *)
- $a \wedge (b \vee c) = (a \wedge b) \vee (a \wedge c)$
- $\neg(\neg a) = a$, $a \vee \neg a = T$, , $a \wedge \neg a = F$,
- $a \wedge b = b \wedge a$; $a \vee b = b \vee a$
- $a \wedge (b \wedge c) = (a \wedge b) \wedge c$
- $a \vee (b \vee c) = (a \vee b) \vee c$
- $a \longrightarrow b = (\neg a) \vee b$
- $(a=b \wedge P(a)) = P(b)$ (* one point rule *)

- let x = E in C(x) = C(E) (* let elimination *)
- if c then C else D $= (c \wedge C) \vee (\neg c \wedge D)$
  $= (c \longrightarrow C) \wedge (\neg c \longrightarrow D)$

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() = r

Test-Data Generation

- Recall the test specification:

  $mk(x,y,z).isTriangle() = r$

  $\equiv$ $inv_{Triangle}(\sigma) \wedge pre_{isTriangle}(mk(x,y,z))(\sigma) \wedge$

  $inv_{Triangle}(\sigma') \wedge post_{isTriangle}(mk(x,y,z),r)(\sigma,\sigma')$

  (* see semantics in MOAL II, page 22. *)

  Some Facts:

  - From modifiesOnly({}) follows $\sigma = \sigma'$ hence

    $inv_{Triangle}(\sigma) = inv_{Triangle}(\sigma')$

  - From $mk(x,y,z) \neq null$ (see $pre_{isTriangle}$) and from $inv_{Triangle}(\sigma)$ and $mk(x,y,z) \in Triangle\ (\sigma)$ follows that:

    $0<x \wedge 0<y \wedge 0<z \ \wedge\ x \leq y+z \ \wedge\ y \leq x+z \ \wedge\ z \leq x+y \qquad (\equiv inv)$

Test-Data Generation

- Recall the test specification:

  mk(x,y,z).isTriangle() = r

  $\equiv$ $inv_{Triangle}(\sigma) \wedge pre_{isTriangle}(mk(x,y,z))(\sigma) \wedge$

  $inv_{Triangle}(\sigma') \wedge post_{isTriangle}(mk(x,y,z),r)(\sigma,\sigma')$

  (* see semantics in MOAL II, page 22. *)

  Some Facts:

  - arb≠equ≠iso
  - $post_{isTriangle}(mk(x,y,z),r)(\sigma,\sigma)$ can be simplified to:

$$(x=y \wedge y=z \longrightarrow r=equ) \wedge$$
$$((x\neq y \vee y\neq z) \wedge (x=y \vee y=z \vee x=z) \longrightarrow r=iso) \wedge$$
$$((x\neq y \wedge y\neq z \wedge x\neq z) \longrightarrow r=arb)$$

Test-Data Generation

- Summing up:

  mk(x,y,z).isTriangle() = r

  $\equiv$ $inv_{Triangle}(\sigma) \wedge pre_{isTriangle}(mk(x,y,z))(\sigma) \wedge$

  $inv_{Triangle}(\sigma') \wedge post_{isTriangle}(mk(x,y,z),r)(\sigma,\sigma')$

  (* see semantics of MOAL II, page 22. *)

  $\Longrightarrow$ (* the discussed facts *)

  $inv \wedge$
  $(x=y \wedge y=z \longrightarrow r=equ) \wedge$
  $((x\neq y \vee y\neq z) \wedge (x=y \vee y=z \vee x=z) \longrightarrow r=iso) \wedge$
  $(x\neq y \wedge y\neq z \wedge x\neq z \longrightarrow r=arb)$

Test-Data Generation

- Recall the test specification:

$$\text{inv} \wedge (x{=}y \wedge y{=}z \longrightarrow r{=}equ) \wedge$$
$$((x{\neq}y \vee y{\neq}z) \wedge (x{=}y \vee y{=}z \vee x{=}z) \longrightarrow r{=}iso) \wedge$$
$$(x{\neq}y \wedge y{\neq}z \wedge x{\neq}z \longrightarrow r{=}arb)$$

$\equiv$ (* elimination $\longrightarrow$, deMorgan*)

$$\text{inv} \wedge$$
$$(x{\neq}y \vee y{\neq}z \vee r{=}equ) \wedge$$
$$((x{=}y \wedge y{=}z) \vee (x{\neq}y \wedge y{\neq}z \wedge x{\neq}z) \vee r{=}iso) \wedge$$
$$(x{=}y \vee y{=}z \vee x{=}z \vee r{=}arb)$$

Test-Data Generation

- This first part of the calculation could be called

  PURIFICATION

  We eliminate UML, object-orientation, MOAL etcpp and reduce it to the pure logical core ...

  Now, under which precise conditions do we have

  - r = iso
  - r = arb
  - r = equ ???

Test-Data Generation

- This first part of the calculation could be called

  PURIFICATION

  We eliminate UML, object-orientation, MOAL etcpp and reduce it to the pure logical core …

Can we transform the spec into the form

- $A_1 \wedge \ldots \wedge A_i \wedge r = iso$
- $B_1 \wedge \ldots \wedge B_k \wedge r = arb$
- $C_1 \wedge \ldots \wedge C_l \wedge r = equ$???

Test-Data Generation

- This first part of the calculation could be called

  PURIFICATION

  We eliminate UML, object-orientation, MOAL etcpp and reduce it to the pure logical core ...

  Can we transform the spec into a

  Disjunctive Normal Form (DNF) ?

Excursion

- Generalized Distribution Laws:

$$(A_1 \vee A_2) \wedge (B_1 \vee B_2) = (A_1 \wedge (B_1 \vee B_2)) \vee (A_2 \wedge (B_1 \vee B_2))$$
$$= (A_1 \wedge B_1) \vee (A_2 \wedge B_1) \vee (A_1 \wedge B_2) \vee (A_2 \wedge B_2)$$

$$(A_1 \vee A_2 \vee A_3) \wedge (B_1 \vee B_2 \vee B_3) \wedge (C_1 \vee C_2 \vee C_3)$$
$$= \ldots$$
$$= (A_1 \wedge B_1 \wedge C_1) \vee (A_1 \wedge B_1 \wedge C_2) \vee (A_1 \wedge B_1 \wedge C_3) \vee$$
$$(A_2 \wedge B_1 \wedge C_1) \vee (A_2 \wedge B_1 \wedge C_2) \vee (A_2 \wedge B_1 \wedge C_3) \vee$$
$$\ldots$$
$$(A_1 \wedge B_3 \wedge C_3) \vee (A_2 \wedge B_3 \wedge C_3) \vee (A_3 \wedge B_3 \wedge C_3)$$

Test-Data Generation

- Recall the test specification:

$$\begin{aligned}
&\ldots\\
\equiv\ & inv \wedge\\
& (x\neq y \vee y\neq z \vee r=equ) \wedge\\
& (x=y \vee y=z \vee x=z \vee r=arb) \wedge\\
& ((x=y \wedge y=z) \vee (x\neq y \wedge y\neq z \wedge x\neq z) \vee r=iso)
\end{aligned}$$

$\equiv$ (* generalized distribution 2nd/3rd line *)

$$\begin{aligned}
& inv \wedge\\
& ((x\neq y \wedge x=y) \vee (x\neq y \wedge y=z) \vee (x\neq y \wedge x=z) \vee (x\neq y \wedge r=arb)) \vee\\
& ((y\neq z \wedge x=y) \vee (y\neq z \wedge y=z) \vee (y\neq z \wedge x=z) \vee (y\neq z \wedge r=arb)) \vee\\
& ((r=equ \wedge x=y) \vee (r=equ \wedge y=z) \vee (r=equ \wedge x=z) \vee (r=equ \wedge r=arb)) \vee\\
& ((x=y \wedge y=z) \vee (x\neq y \wedge y\neq z \wedge x\neq z) \vee r=iso)
\end{aligned}$$

Test-Data Generation

- Recall the test specification:

  ...

  $\equiv$ inv $\land$
  $(x \neq y \lor y \neq z \lor r = equ) \land$
  $(x = y \lor y = z \lor x = z \lor r = arb) \land$
  $((x = y \land y = z) \lor (x \neq y \land y \neq z \land x \neq z) \lor r = iso)$

  $\equiv$ (* elimination contradictions *)
  inv $\land$
  (~~$(x \neq y \land x = y)$~~ $\lor (x \neq y \land y = z) \lor (x \neq y \land x = z) \lor (x \neq y \land r = arb) \lor$
  $(y \neq z \land x = y) \lor$ ~~$(y \neq z \land y = z)$~~ $\lor (y \neq z \land x = z) \lor (y \neq z \land r = arb) \lor$
  $(r = equ \land x = y) \lor (r = equ \land y = z) \lor (r = equ \land x = z) \lor$ ~~$(r = equ \land r = arb)$~~) $\lor$
  $((x = y \land y = z) \lor (x \neq y \land y \neq z \land x \neq z) \lor r = iso)$

Test-Data Generation

- Recall the test specification:

  ...

  $\equiv$ (* elimination contradictions *)

  inv ∧
  ((x≠y ∧ y=z)∨(x≠y ∧ x=z)∨(x≠y ∧ r=arb) ∨
  (y≠z ∧ x=y)∨(y≠z ∧ x=z)∨(y≠z ∧ r=arb) ∨
  (r=equ∧x=y)∨(r=equ∧y=z)∨(r=equ∧x=z)) ∧
  ((x=y ∧ y=z) ∨ (x≠y ∧ y≠z ∧ x≠z) ∨ r=iso)

Test-Data Generation

- $\equiv$ (* generalized distribution 2nd/3rd ((9 * 3 = 27 cases !)*)

$$
\begin{aligned}
&\mathit{inv} \wedge \\
&\big((x\neq y\wedge y=z\wedge x=y\wedge y=z)\vee(x\neq y\wedge x=z\wedge x=y\wedge y=z)\vee(x\neq y\wedge r=\mathit{arb}\wedge x=y\wedge y=z)\vee \\
&\;(y\neq z\wedge x=y\wedge x=y\wedge y=z)\vee(y\neq z\wedge x=z\wedge x=y\wedge y=z)\vee(y\neq z\wedge r=\mathit{arb}\wedge x=y\wedge y=z)\vee \\
&\;(r=\mathit{equ}\wedge x=y\wedge x=y\wedge y=z)\vee(r=\mathit{equ}\wedge y=z\wedge x=y\wedge y=z)\vee(r=\mathit{equ}\wedge x=z\wedge x=y\wedge y=z)\big)\vee \\
&\big((x\neq y\wedge y=z\wedge x\neq y\wedge y\neq z\wedge x\neq z)\vee(x\neq y\wedge x=z\wedge x\neq y\wedge y\neq z\wedge x\neq z)\vee(x\neq y\wedge r=\mathit{arb}\wedge \\
&x\neq y\wedge y\neq z\wedge x\neq z)\vee(y\neq z\wedge x=y\wedge x\neq y\wedge y\neq z\wedge x\neq z)\vee(y\neq z\wedge x=z\wedge x\neq y\wedge y\neq z\wedge \\
&x\neq z)\vee(y\neq z\wedge r=\mathit{arb}\wedge x\neq y\wedge y\neq z\wedge x\neq z)\vee(r=\mathit{equ}\wedge x=y\wedge x\neq y\wedge y\neq z\wedge x\neq z)\vee(r \\
&=\mathit{equ}\wedge y=z\wedge x\neq y\wedge y\neq z\wedge x\neq z)\vee(r=\mathit{equ}\wedge x=z\wedge x\neq y\wedge y\neq z\wedge x\neq z)\big)\vee \\
&\big((x\neq y\wedge y=z\wedge r=\mathit{iso})\vee(x\neq y\wedge x=z\wedge r=\mathit{iso})\vee(x\neq y\wedge r=\mathit{arb}\wedge r=\mathit{iso}) \\
&\;\vee(y\neq z\wedge x=y\wedge r=\mathit{iso})\vee(y\neq z\wedge x=z\wedge r=\mathit{iso})\vee(y\neq z\wedge r=\mathit{arb}\wedge r=\mathit{iso})\vee \\
&\;(r=\mathit{equ}\wedge x=y\wedge r=\mathit{iso})\vee(r=\mathit{equ}\wedge y=z\wedge r=\mathit{iso})\vee(r=\mathit{equ}\wedge x=z\wedge r=\mathit{iso})\big)
\end{aligned}
$$

Test-Data Generation

- ≡ (* elimination of the contradictions and redundancies *)

inv ∧

(~~(x≠y∧y=z∧x=y∧y=z)~~ ∨ ~~(x≠y∧x=z∧ x=y∧y=z)~~ ∨ ~~(x≠y∧r=arb∧x=y∧y=z)~~ ∨
~~(y≠z∧x=y∧x=y∧y=z)~~ ∨ ~~(y≠z∧x=z∧ x=y∧y=z)~~ ∨ ~~(y≠z∧r=arb∧x=y∧y=z)~~ ∨
(r=equ∧x=y∧x=y∧y=z) ∨ (r=equ∧ y=z∧x=y∧y=z) ∨ (r=equ∧x=z∧x=y∧y=z)) ∨

(~~(x≠y∧y=z∧x≠y∧y≠z∧x≠z)~~ ∨ ~~(x≠y∧x=z∧x≠y∧y≠z∧x≠z)~~ ∨ (x≠y∧r=arb∧ x≠y∧y≠z∧x≠z) ∨ ~~(y≠z∧x=y∧x≠y∧y≠z∧x≠z)~~ ∨ ~~(y≠z∧x=z∧x≠y∧y≠z∧ x≠z)~~ ∨ (y≠z∧r=arb∧x≠y∧y≠z∧x≠z) ∨ ~~(r=equ∧x=y∧x≠y∧y≠z∧x≠z)~~ ∨ ~~(r =equ∧y=z∧x≠y∧y≠z∧x≠z)~~ ∨ ~~(r=equ∧x=z∧x≠y∧y≠z∧ x≠z)~~) ∨

((x≠y ∧ y=z∧r=iso) ∨ (x≠y ∧ x=z∧r=iso) ∨ ~~(x≠y∧r=arb∧r=iso)~~
∨ (y≠z∧x=y∧r=iso) ∨ (y≠z∧x=z∧r=iso) ∨ ~~(y≠z∧r=arb∧r=iso)~~ ∨
~~(r=equ∧x=y∧r=iso)~~ ∨ ~~(r=equ∧y=z∧r=iso)~~ ∨ ~~(r=equ∧x=z∧r=iso)~~)

Test-Data Generation

- $\equiv$ (* cleanup, distribution *)

```
(inv ∧ x=y ∧ x=y ∧ y=z ∧ r=equ) ∨            (1)
(inv ∧ x≠y ∧ y≠z ∧ x≠z ∧ r=arb ) ∨           (2)
(inv ∧ x≠y ∧ y=z ∧ r=iso) ∨                  (3)
(inv ∧ x≠y ∧ x=z ∧ r=iso) ∨                  (4)
(inv ∧ y≠z ∧ x=y ∧ r=iso) ∨                  (5)
(inv ∧ y≠z ∧ x=z ∧ r=iso)                    (6)
```

- Test-Case-Construction by DNF Method

  yields six abstract test cases

  relatibng input x y z to output r

- Note: In general, output r is not necessarily uniquely defined as in our example ...

  The spec can be non-deterministic admitting several results.

Test-Data Generation

- Test-Data-Selection:
  For each abstract test-case, we construct one concrete test, by choosing values that make the abstract test case true (« that satisfies the abstract test case »)

case	x	y	z	result
(1)	3	3	3	equ
(2)	3	4	6	arb
(3)	4	5	5	iso
(4)	5	4	5	iso
(5)	5	5	4	iso
(6)	4	3	4	iso

Test-Data Generation

- A First Summary on the Test-Generation Method:
  - PHASE I: Stripping the Domain-Language (UML-MOAL) away, "purification"
  - PHASE II: Abstract Test Case Construction by "DNF computation"
  - PHASE III: Constraint Resolution (by solvers like CVC4 or Z3) "Test Data Selection"
  - COVERAGE CRITERION:
    DNF - coverage of the Spec; for each abstract test-case one concrete test-input is constructed.
    (**ISO/IEC/IEEE** 29119 calls this: Equivalence class testing)
- Remark: During Codiung phase, when the Spec does not change, the test-data-selection can be repeated easily creating always different test sets …

Test-Data Generation

- Variants:
  - Alternative to PHASE II (DNF construction):
    Predicate Abstraction and Tableaux-Exploration.

    Reconsider the (purified) specification:

    $$\begin{array}{l} \mathrm{inv} \wedge \\ (x=y \wedge y=z \longrightarrow r=\mathrm{equ}) \wedge \\ ((x\neq y \vee y\neq z) \wedge (x=y \vee y=z \vee x=z) \longrightarrow r=\mathrm{iso}) \wedge \\ (x\neq y \wedge y\neq z \wedge x\neq z \longrightarrow r=\mathrm{arb}) \end{array}$$

    It is possible to abstract this spec to a fairly small number of „base predicates" … They should be logically independent and not contain the output variable…

Test-Data Generation

- Variants:
  - Alternative to PHASE II (DNF construction): Predicate Abstraction and Tableaux-Exploration.

    Reconsider the (purified) specification:

    $$\begin{array}{l} \mathtt{inv} \wedge \\ (A \wedge B \longrightarrow r{=}equ) \wedge \\ ((\neg A \vee \neg B) \wedge (A \vee B \vee C) \longrightarrow r{=}iso) \wedge \\ (\neg A \wedge \neg B \wedge \neg C \longrightarrow r{=}arb) \end{array}$$

    where $A \mapsto x{=}y$, $B \mapsto y{=}z$, $C \mapsto x{=}z$

    (actually: A and B imply C)

Test-Data Generation

- Variants:
  - … Now we can construct a tableau and get by simplification:

case	A	B	C	spec reduces to
(1)	T	T	T	• `r=equ`
(2)	T	T	F	• `r=equ  (!!!)`
(3)	T	F	T	• `r=iso`
(4)	T	F	F	• `r=iso`
(5)	F	T	T	• `r=iso`
(6)	F	T	F	• `r=iso`
(7)	F	F	T	• `r=iso`
(8)	F	F	F	• `r=arb`

Test-Data Generation

- Variants:
  - PHASE III: Borderline analysis.
    Principle: we replace in our DNF inequalities by „the closest values that make the spec true"

    $x \neq y \quad \mapsto \quad x = y + 1 \lor x = y - 1$

    $x \leq y \quad \mapsto \quad x = y \lor x < y$

    $x < y \quad \mapsto \quad x = y - 1$ etc.

  - … and recompute the DNF. In general, this gives a much finer mesh …

Test-Data Generation

- Variants:
  - PHASE I: Test for exceptional behaviour.

    We negate the precondition and to DNF generation on the precondition only.

    Test objectives could be:

      - should raise an exception if public
      - should not diverge

Test-Data Generation

- How to handle Recursion ?

Test-Data Generation

- How to handle Recursion ?

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

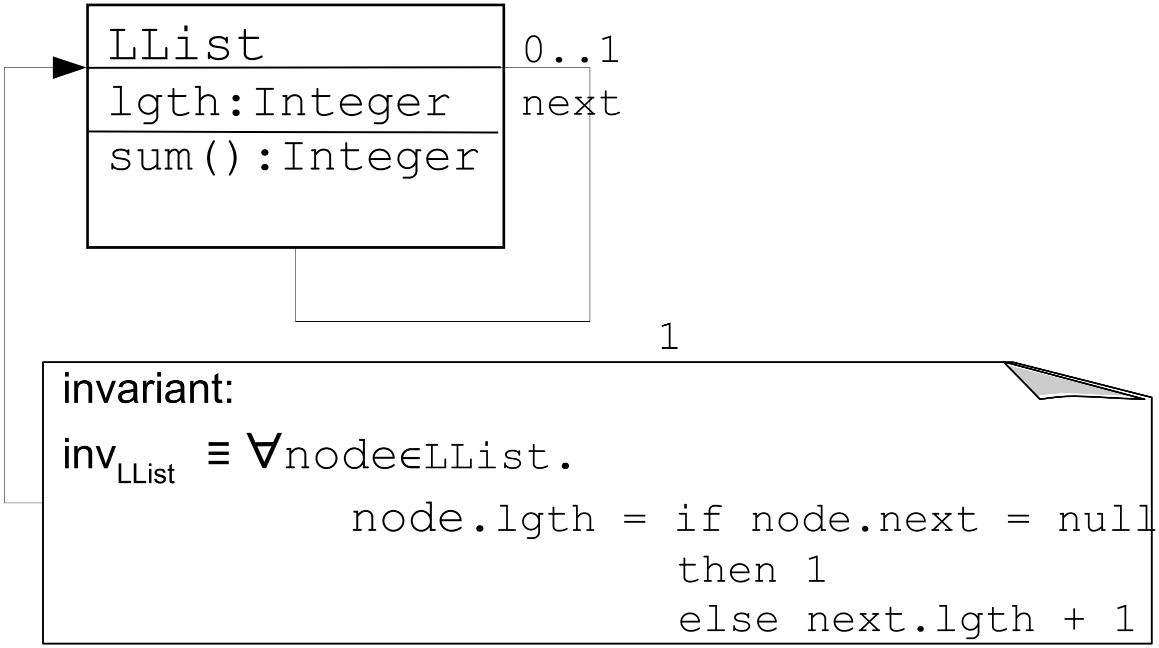
  - at the level of data

Test-Data Generation

- How to handle Recursion ?

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

  - at the level of data

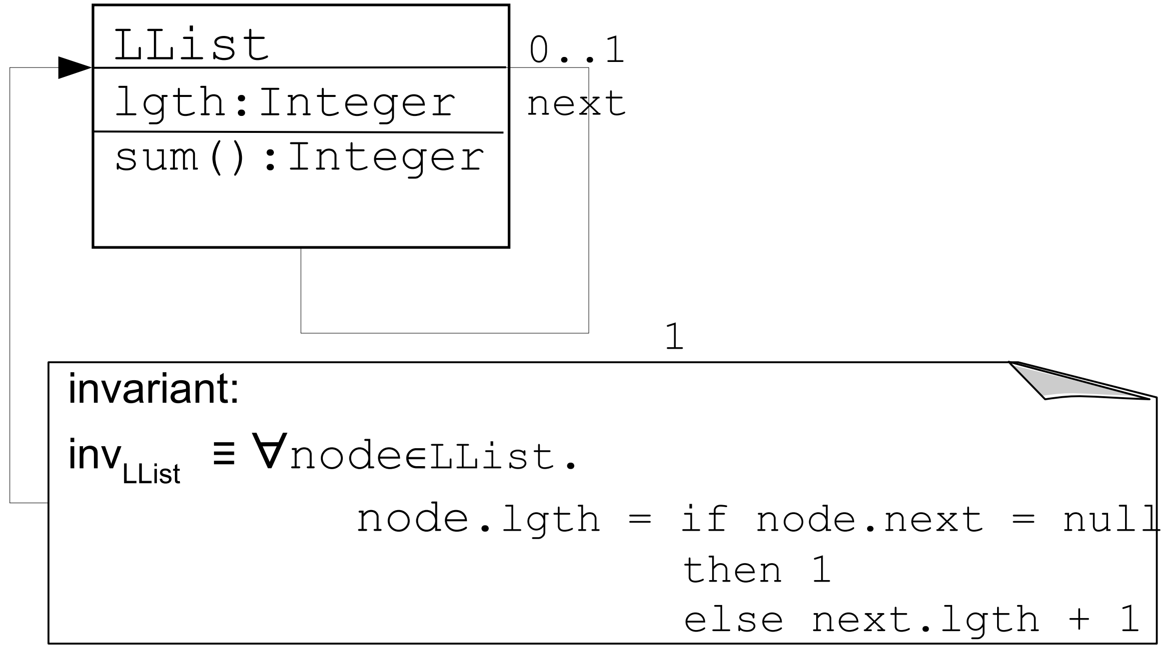


Test-Data Generation

- How to handle Recursion ?

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

  - at the level of data



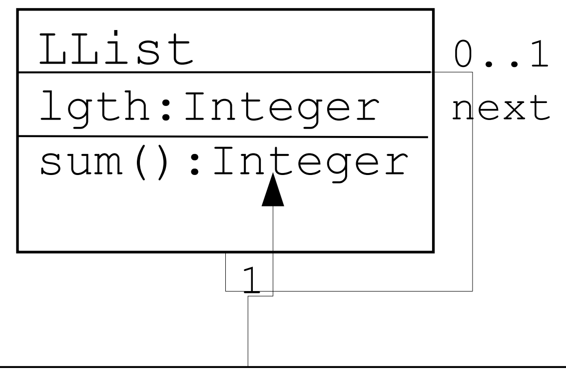
Note that this excludes cyclic lists !!!

Test-Data Generation

- How to handle Recursion ?

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

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



query contract (modifiesOnly({})):

definition $\mathtt{pre}_{\mathtt{sum}}(\mathtt{l}) \equiv \mathtt{True}$

definition $\mathtt{post}_{\mathtt{sum}}(\mathtt{l,res}) \equiv \mathtt{res=if\ l.next=null\ then\ l.lgth}$
$\mathtt{else\ l.lgth + l.next.sum()}$

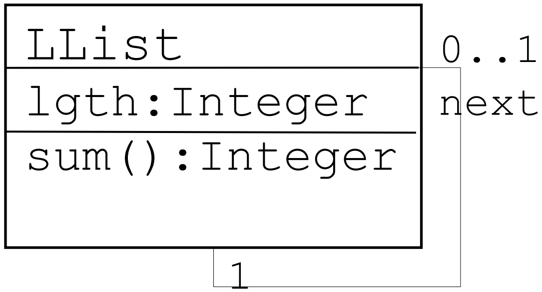
definition $\mathtt{sum(l)} \equiv \mathtt{arb\{r \mid pre}_{\mathtt{sum}}\mathtt{(l)} \wedge \mathtt{post}_{\mathtt{sum}}\mathtt{(l,r)\}}$

Test-Data Generation

- How to handle Recursion ?

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

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



Note that arb(S) gives an arbitrary member of S: arb(S)∈S. Since from x=arb({y}) follows x=y; thus sum(l) is (uniquely) defined. The (σ,σ`) convention applies.

$_{um}$(l) ≡ True

$_{sum}$(l,res)≡ res=if l.next=null then l.lgth
else l.lgth + l.next.sum()

(l)≡ arb{r|pre$_{sum}$(l) ∧ post$_{sum}$(l,r)}

Test-Data Generation

- Prerequisite: We present the invariant as recursive predicate.

definition inv_{LList_Core} `n σ ≡ (n.lgth(σ) = if n.next(σ)=null then 1`
`else n.next.lgth(σ) + 1)`

we have:

inv_{LList} `(σ) = ∀n∈LList(σ).` inv_{LList_Core} `n σ`

and

inv_{LList_Core} `(n)(σ)= (if n.next(σ)=null then n.lgth(σ) = 1`
`else n.lgth(σ) =n.next.lgth(σ) + 1`
`∧ n.next(σ)∈LList(σ)`
`∧` inv_{LList_Core} `(n.next)(σ))`

(under the assumption that `n∈LList(σ)`)

Furthermore we have:

```
sum(l)(σ',σ) = if l.next(σ)=null then l.lgth(σ)
                        else l.lgth(σ) + sum(l.next)(σ',σ)
```

We have σ'=σ (why?). We will again apply (σ',σ) - convention.

Test-Data Generation

- Consider the test specification:

  X.sum() ≡ Y (for some X∈LList, i.e. X≠null)

  $\equiv \mathtt{inv}_{\mathtt{LList}}(\mathtt{X}) \wedge \mathtt{pre}_{\mathtt{sum}}(\mathtt{X}) \wedge \mathtt{post}_{\mathtt{sum}}(\mathtt{X,Y})$

  where:

  $\mathtt{pre}_{\mathtt{sum}}(\mathtt{X}) \equiv \mathtt{true}$

  $\mathtt{post}_{\mathtt{sum}}(\mathtt{X,Y}) \equiv (\mathtt{if\ X.next = null\ then\ Y = X.lgth}$
  $\mathtt{else\ Y = X.lgth + sum(X.next))}$

  $\equiv (\mathtt{X.next{=}null} \wedge \mathtt{Y = X.lgth})$
  $\vee (\mathtt{X.next{\neq}null} \wedge \mathtt{Y = X.lgth{+}sum(X.next)})$

Test-Data Generation

- DNF computation yields already the test cases:

X.sum() ≡ Y (for some X∈LList, i.e. X≠null)

$\Longrightarrow$ inv_{LList_Core}(X) ∧ $post_{sum}$(X,Y))

≡ (if X.next=null then X.lgth = 1
  else X.lgth =X.next.lgth+1 ∧ X.next∈LList ∧ inv_{LList_Core}(X.next)) ∧
  (if X.next = null then Y = X.lgth
                    else Y = X.lgth + sum(X.next))

≡ (DNF)

  (X.next=null ∧ X.lgth=1 ∧ Y = X.lgth)
  ∨ (X.next≠null ∧ X.lgth =X.next.lgth+1
     ∧ X.next∈LList ∧ inv_{LList_Core}(X.next)
     ∧ Y = X.lgth+sum(X.next))

Test-Data Generation

- DNF computation yields already the test cases:

X.sum() ≡ Y (for some X∈LList, i.e. X≠null)

⟹ inv_{LList_Core}(X) ∧ $post_{sum}$(X,Y))

≡ (if X.next=null then X.lgth = 1
else X.lgth =X.next.lgth+1 ∧ X.next∈LList ∧ inv_{LList_Cor}
(if X.next = null then Y = X.lgth
else Y = X.lgth + sum(X.next))

≡ (if c then C else D elim, DNF)

(X.next=null ∧ X.lgth=1 ∧ Y = X.lgth)
∨ (X.next≠null ∧ X.lgth =X.next.lgth+1
∧ X.next∈LList ∧ inv_{LList_Core}(X.next)
∧ Y = X.lgth+sum(X.next))

New Test-Case!!

Test-Data Generation

- Intermediate Summary: test-cases known so far ?

X	Y
	1
...	...
...	...

Test-Data Generation

- Prerequisite: We present the invariant as recursive predicate.

inv_{LList_Core} `(n)=  (if n.next=null then n.lgth = 1`
`else n.lgth =n.next.lgth + 1`
`∧ n.next∈LList ∧` inv_{LList_Core} `(n.next))`

-
```
sum(l) =  if l.next=null then l.lgth
          else l.lgth + sum(l.next)
```

```
sum(l) =  if X.next.next=null then X.next.lgth
          else X.next.lgth + sum(X.next.next)
```

Test-Data Generation

- DNF computation yields already the test cases:

  X.sum() ≡ Y (for some X∈LList, i.e. X≠null)

  ⟹ … ≡ …

  ≡ (unfolding sum and inv_{LList_Core})

```
(X.next=null ∧ X.lgth=1 ∧ Y = X.lgth)
∨ (X.next≠null ∧ X.lgth=X.next.lgth+1 ∧ X.next∈LList
    ∧ (if X.next.next=null then X.next.lgth = 1
            else X.next.lgth =X.next.next.lgth + 1
                  ∧ X.next.next∈LList ∧ inv_LList_Core(X.next.next))
    ∧ (Y = X.lgth+(if X.next.next=null then X.next.lgth
                        else X.next.lgth + sum(X.next.next))))
```

Test-Data Generation

- DNF computation yields already the test cases:

  X.sum() ≣ Y (for some X∈LList, i.e. X≠null)

  ⟹ … ≣ ...

  ≣ (DNF partial)

```
(X.next=null ∧ X.lgth=1 ∧ Y = X.lgth)
∨ (X.next≠null ∧ X.lgth=X.next.lgth+1 ∧ X.next∈LList
    ∧ ((X.next.next=null ∧ X.next.lgth = 1 ∧
         Y = X.lgth+X.next.lgth)
      ∨ (X.next.next≠null ∧ X.next.lgth=X.next.next.lgth+1
          ∧ X.next.next∈LList ∧ inv_LList_Core(X.next.next)
          ∧ Y = X.lgth+ X.next.lgth + sum(X.next.next))
      )
```

Test-Data Generation

- DNF computation yields already the test cases:

X.sum() ≣ Y (for some X∈LList, i.e. X≠null)

⟹ … ≣ ...

≣ (DNF partial)

```
(X.next=null ∧ X.lgth=1 ∧ Y = X.lgth)
∨ (X.next≠null ∧ X.lgth=X.next.lgth+1 ∧ X.next∈LList
   ∧ X.next.next=null ∧ X.next.lgth=1 ∧
     Y = X.lgth+X.next.lgth))
∨ (X.next≠null ∧ X.lgth=X.next.lgth+1 ∧ X.next∈LList
   ∧ X.next.next≠null ∧ X.next.lgth=X.next.next.lgth+1
        ∧ X.next.next∈LList ∧ inv_LList_Core(X.next.next)
        ∧ Y = X.lgth+ X.next.lgth + sum(X.next.next)))
```

Test-Data Generation

- DNF computation yields already the test cases:

  X.sum() ≣ Y (for some X∈LList, i.e. X≠null)

  ⟹ … ≣ …

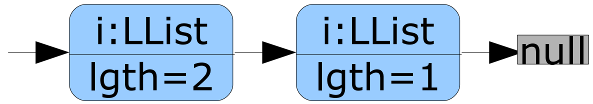
  ≣ (DNF partial)

```
(X.next=null ∧ X.lgth=1 ∧ Y = X.lgth)
∨ (X.next≠null ∧ X.lgth=X.next.lgth+1 ∧ X.next∈LList
    ∧ X.next.next=null ∧ X.next.lgth=1 ∧
      Y = X.lgth+X.next.lgth))
∨ (X.next≠null ∧ X.lgth=X.next.lgth+1 ∧ X.next∈LList
    ∧ X.next.next≠null ∧ X.next.lgth=X.next.next.lgth+1
          ∧ X.next.next∈LList ∧ inv_LList_Core(X.next.next)
          ∧ Y = X.lgth+ X.next.lgth + sum(X.next.next)))
```

New Test-Case!!

Test-Data Generation

Intermediate Summary: test-cases known so far ?

X	Y
	1
	3
...	...

Summary: Symbolic Test-Case Generation

- ... and we could continue forever
  - compile to semantics
    (-> convert in mathematical, logical notation)
  - use recursive predicates, recursive contracts
  - enter loop:
    - unfold predicates one step
    - compute DNF
    - simplify DNF
    - extract test-cases

    until we are satisfied, i.e. have „enough" test cases ...
  - Select test-data: constraint resolution of test cases.

Test-Data Generation

- Observation: "all other cases" ...
  were represented by the clauses still containing recursive predicates.
- Logically: we used a regularity hypothesis, i.e ...

$$(\forall X.\ |X|<k \Rightarrow X.sum() \equiv Y) \Rightarrow (\forall X.\ X.sum() \equiv Y)$$

where we choose as "complexity mesure" |X| just X.lgth and k (the number of unfoldings) was 2 ...

Test-Data Generation

- Coverage Criterion for a Test:

DNF_k

For all data up to complexity k, we constructed abstract test-cases and generated a test.

In our example, the “complexity measure” is just the length of the LLists.

Test-Data Generation

- What are the alternatives to symbolic test-case generation ?

  Must this really be so complicated ???

  Well, think about the probability to "guess" input with a complex invariant and precondition, if you use "blind" random-generation procedure ...

Test-Data Generation

- Summary
  - We have (sketched) a symbolic Test-Case Generation Procedure for UML/MOAL Specifications
  - It takes into account:
    - object orientation
    - data invariants (recursive predicates)
    - recursive functions (via unfolding)
  - The process can be tool-supported (HOL-TestGen)
  - The process is intended for automation.

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 conjoint in the DNF_k)
    one test, i.e. one substitution for the free variables satisfying the test-case !