POLYTECH PARIS-SUD

2026

Cycle Ingénieur – 2ème année
Département Informatique

Verification and Validation

Part IV : An Introduction to Testing

Burkhart Wolff
Département Informatique
Université Paris-Saclay / LMF

Recall: Validation and Verification

- Validation :
  - Does the system meet the clients requirements ?
  - Will the performance be sufficient ?
  - Will the usability be sufficient ?

  Do we build the right system ?

- Verification:
  - Does the system meet the specification ?
  - Does it correspond to a (mathematical, formal) model ?

  Do we build the system right ?   Is it « correct » ?

How to do Validation ?

- Tests and Experiments over Systems (Integrated artefacts consisting of software and hardware …)

How to do Verification ?

- Test and Proof on the basis of formal specifications (e.g., à la OCL, MOAL, ACSL, ... !) against programs or systems ...

Recall: Verification Costs in an SE Process

- costs ? *35 - 50 % of the global effort ?*
- all "real" (large) software has remaining bugs ...
- The cost of bug ?
  - the cost to reveal and fix it ...
    or:
    the cost of a legal battle it may cause...
    or the potential damage to the image
    (difficult to evaluate, but veeeery real)
    or costs as a result to come later on the market
  - *on the other side – you can't test infinitely, and verification is again 10 times more costly than thoroughly testing !*

Verification Costs

- Conclusion:
  - verification and software quality is vitally important, and also critical in the development
  - to do it cost-effectively, it requires
    - a lot of expertise on products and process
    - a lot of knowledge over methods, tools, and tool chains …

Overview on the part on « Test »

- WHAT IS TESTING ?
- A taxonomy on types of tests
  - Static Test / Dynamic (*Runtime*) Test
  - Structural Test / Functional Test
  - Statistic Tests
- Functional Test; Link to UML/OCL
  - Dynamic Unit Tests, Static Unit Tests,
  - Coverage Criteria
- Structural Tests
  - Control Flow and Data Flow Graphs
  - Tests and executed paths. Undecidability.
  - Coverage Criteria

What is testing ?

- It is an approximation to verification
- Main interest: finding bugs early,
  - either in the model
  - or in the program
  - or in both

- A systematic test is:
  - process programs and specifications and to compute a set of test-cases under controlled conditions.

  - *ideally*: testing is complete if a certain criteria, the adequacy criteria is reached.

Limits of testing ?

- We said, test is an approximation to verification, usually easier (and less expensive)

- Note: Sometimes it is easier to verify than to test. In particular:

  - low-level OS implementations: memory allocation, garbage collection memory virtualization, … crypt-algorithms, ...

  - non-deterministic programs with no control over the non-determinism.

Taxomomy: Static / Dynamic Tests

- **static**: running a program before deployment on data carefully constructed by the analyst (in a testing environment)
  - analyse the result on the basis of all components
  - working on some classes of executions symbolically = representing infinitely many executions
- **dynamic**: running the programme (or component) after deployment, on "real data" as imposed by the application domain
  - experiment with the real behaviour
  - essentially used for post-hoc ananalysis and debugging

Taxonomy: Unit / Sequence / Reactive Tests

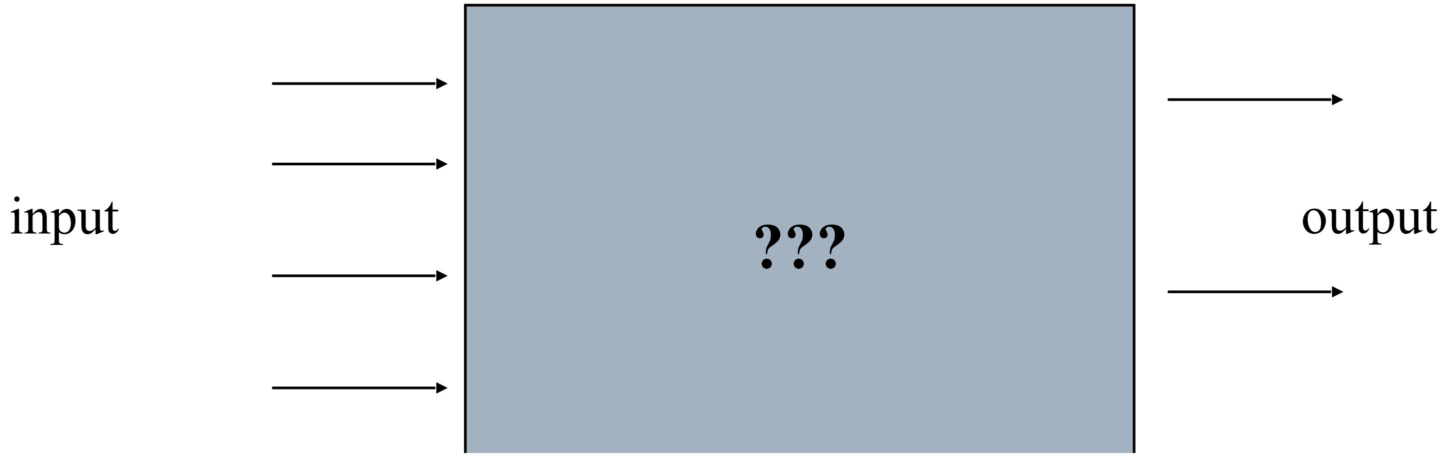
- **unit**: testing of a local component (function, module), typically only one step of the underlying state.
  (In functional programs, thats essentially all what you have to do!)
- **sequence**: testing of a local component (function, module), but typicallY sequences of executions, which typically depend on internal state
- **reactive sequence**: testing components by sequences of steps, but these sequences represent communication where later parts in the seqience depend on what has been earlier cummunicated

Taxonomy: Functional / Structural Test

- **functional**: (also: black-box tests). Tests were generated
  on a specification of the component, the test focusses on input output behaviour.
- **structural**: (also: white-box tests). Tests were generated on the basis of the structure or the program, i.e. using
  control-flow, data-flow paths or by using symbolic executions.
- **both**: (also: grey-box testing).

Functional Dynamic Unit Test

- We got the spec, but not the program, which is considered as a black box:



we focus on what the program *should* do !!!

Functional Dynamic Unit Test : an example

The (informal) specification:

Read a "Triangle Object" (with three sides of integral type), and test if it is isoscele, equilateral, or (default) arbitrary.

Each length should be strictly positive.

Give a specification, and develop a test set ...

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

We add the constraints of the analysis:

```
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():
  post  t.a=t.b ∧ t.b=t.c ⟶ result=equ
  post  (t.a≠t.b ∨ t.b≠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
```

Functional Dynamic Unit Test : an example

Can we use specifications to perform Runtime-Test?

Yes! Compile:

context C::m(a_1:C_1,...,a_n:C_n)

pre : P(self,a_1,...,a_n)

post : Q(self,a_1,...,a_n,result)

to some checking code (with "assert" as in Junit, VCC, Boogie, ...)

check_C(); check_C_1(); ... ; check_C_n();

assert(P(self,a_1,...,a_n));

result=run_m(self,a_1,...,a_n);

assert(Q(self,a_1,...,a_n,result));

Functional Dynamic Unit Test : an example

Dynamic (Unit/Sequence/...) Runtime-Tests are:

- ... easy to implement and enforce
- ... work on real data and are extremely helpful for post-hoc crash-analysis, debugging, and forensics.
- Runtime-tests conflict with efficiency
- But: they are NOT particularly useful during development, where we need systematic test-data EARLY.

Can we do better ?

- We need a method that:
  - generates the tests from the model („model-based testing"): if the model changes, the tests follow. This would all simplify the maintenance problem of large test sets.
  - ... works for partial programs ...
  - ... works in the implementation phase (and gives immediate feedback to programmers) and not at the deployment phase (so: runs very late) ...
  - ... gives clear criteria on the question: „did we test enough" ?

Intuitive Test-Data Generation

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

  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 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?

Intuitive Test-Data Generation

- Ouf, is there a systematic and automatic way to compute all these tests ?
- Can we avoid hand-written test-scripts ? Avoid the task to maintain them ?
- And the question remains:

  When did we test „enough" ?

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 of MOAL in Part III *)

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:

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

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

- 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 d'un appel de methopde, 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:

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

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

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

$\Longrightarrow$ (* 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:

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

≡ (* elimination ⟶ , 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:

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

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

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

$= \ldots$

$= (A_1 \wedge B_1 \wedge C_1) \vee (A_1 \wedge B_1 \wedge C_2) \vee (A_1 \wedge B_1 \wedge C_3) \vee$

$(A_2 \wedge B_1 \wedge C_1) \vee (A_2 \wedge B_1 \wedge C_2) \vee (A_2 \wedge B_1 \wedge C_3) \vee$

…

$(A_1 \wedge B_3 \wedge C_3) \vee (A_2 \wedge B_3 \wedge C_3) \vee (A_3 \wedge B_3 \wedge C_3)$

Test-Data Generation

- Recall the test specification:

distrib

...

$\equiv$ inv $\wedge$

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

$\equiv$

inv $\wedge$

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

Test-Data Generation

- Recall the test specification:

  ...

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

  $\equiv$ (* elimination contradictions *)
  inv ∧
  ((x≠y ∧ x=y)∨(x≠y ∧ y=z)∨(x≠y ∧ x=z)∨(x≠y ∧ r=arb) ∨
  (y≠z ∧ x=y)∨(y≠z ∧ y=z)∨(y≠z ∧ x=z)∨(y≠z ∧ r=arb) ∨
  (r=equ∧x=y)∨(r=equ∧y=z)∨(r=equ∧x=z)∨(r=equ∧r=arb)) ∨
  ((x=y ∧ y=z) ∨ (x≠y ∧ y≠z ∧ x≠z) ∨ r=iso)

Test-Data Generation

- Recall the test specification:

  ...

  ≡ (* elimination contradictions *)

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

Test-Data Generation

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

$$
\begin{array}{l}
\mathrm{inv}\ \land \\
\big((x\neq y\land y=z\land x=y\land y=z)\lor(x\neq y\land x=z\land x=y\land y=z)\lor(x\neq y\land r=arb\land x=y\land y=z)\ \lor \\
(y\neq z\land x=y\land x=y\land y=z)\lor(y\neq z\land x=z\land x=y\land y=z)\lor(y\neq z\land r=arb\land x=y\land y=z)\ \lor \\
(r=equ\land x=y\land x=y\land y=z)\lor(r=equ\land y=z\land x=y\land y=z)\lor(r=equ\land x=z\land x=y\land y=z)\big)\lor \\
\big((x\neq y\land y=z\land x\neq y\land y\neq z\land x\neq z)\lor(x\neq y\land x=z\land x\neq y\land y\neq z\land x\neq z)\lor(x\neq y\land r=arb \\
\land\ x\neq y\land y\neq z\land x\neq z)\lor(y\neq z\land x=y\land x\neq y\land y\neq z\land x\neq z)\lor(y\neq z\land x=z\land x\neq y\land y\neq z\land \\
x\neq z)\lor(y\neq z\land r=arb\land x\neq y\land y\neq z\land x\neq z)\lor(r=equ\land x=y\land x\neq y\land y\neq z\land x\neq z)\lor(\\
r=equ\land y=z\land x\neq y\land y\neq z\land x\neq z)\lor(r=equ\land x=z\land x\neq y\land y\neq z\land\ x\neq z)\big)\lor \\
\big((x\neq y\ \land\ y=z\land r=iso)\lor(x\neq y\ \land\ x=z\land r=iso)\lor(x\neq y\land r=arb\land r=iso) \\
\lor(y\neq z\land x=y\land r=iso)\lor(y\neq z\land x=z\land r=iso)\lor(y\neq z\land r=arb\land r=iso)\ \lor \\
(r=equ\land x=y\land r=iso)\lor(r=equ\land y=z\land r=iso)\lor(r=equ\land x=z\land r=iso)\big)
\end{array}
$$

Test-Data Generation

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

$inv \land$
$\big((x \neq y \land y=z \land x=y \land y=z) \lor (x \neq y \land x=z \land x=y \land y=z) \lor (x \neq y \land r=arb \land x=y \land y=z) \lor$
$(y \neq z \land x=y \land x=y \land y=z) \lor (y \neq z \land x=z \land x=y \land y=z) \lor (y \neq z \land r=arb \land x=y \land y=z) \lor$
$(r=equ \land x=y \land x=y \land y=z) \lor \underline{(r=equ \land y=z \land x=y \land y=z)} \lor \underline{(r=equ \land x=z \land x=y \land y=z)}\big) \lor$
$\big((x \neq y \land y=z \land x \neq y \land y \neq z \land x \neq z) \lor (x \neq y \land x=z \land x \neq y \land y \neq z \land x \neq z) \lor (x \neq y \land r=arb \land \underline{x \neq y} \land y \neq z \land x \neq z) \lor (y \neq z \land x=y \land x \neq y \land y \neq z \land x \neq z) \lor (y \neq z \land x=z \land x \neq y \land y \neq z \land x \neq z) \lor \underline{(y \neq z \land r=arb \land x \neq y \land y \neq z \land x \neq z)} \lor (r=equ \land x=y \land x \neq y \land y \neq z \land x \neq z) \lor (r=equ \land y=z \land x \neq y \land y \neq z \land x \neq z) \lor (r=equ \land x=z \land x \neq y \land y \neq z \land x \neq z)\big) \lor$
$\big((x \neq y \land y=z \land r=iso) \lor (x \neq y \land x=z \land r=iso) \lor (x \neq y \land r=arb \land r=iso)$
$\lor (y \neq z \land x=y \land r=iso) \lor (y \neq z \land x=z \land r=iso) \lor (y \neq z \land r=arb \land r=iso) \lor$
$(r=equ \land x=y \land r=iso) \lor (r=equ \land y=z \land r=iso) \lor (r=equ \land x=z \land r=iso)\big)$

Test-Data Generation

- ≡ (* 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

  relating 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

- Intuitively, what does it mean that we "covered" the DNF by tests
  - Any basic predicate ("literal") has been used at least one time
    - ... provided it is not contradictory ("A=False")
    - ... provided that it is not redundant ("A=True")
    - ... provided it is not implied by another literal, i.e. it is subsumed ("B $\longrightarrow$ A")

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:

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

    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:

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

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

    (actually: $\texttt{A}$ and $\texttt{B}$ imply $\texttt{C}$)

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"

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

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

$$x < y \quad \mapsto \quad x = y - 1 \qquad \text{etc.}$$

  - ... 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

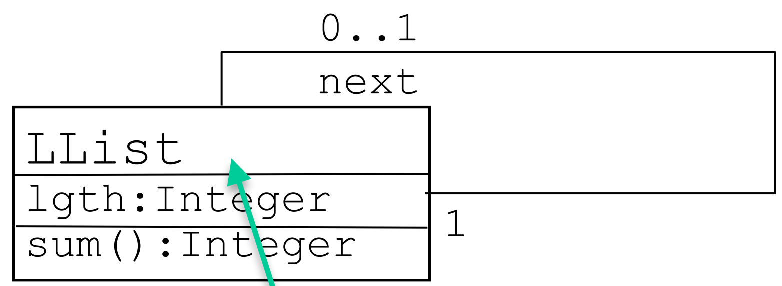
- How to handle Recursion ?

Test-Data Generation

- How to handle Recursion ?

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

  - at the level of data



invariant:

inv_{LList} ≡ ∀node∈LList.

```
node.lgth =if node.next = null
           then 1
           else next.lgth + 1
```

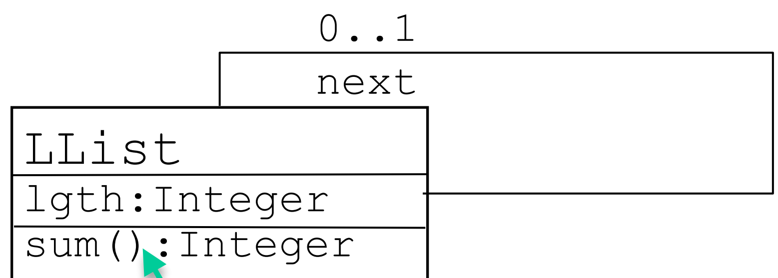
Note that this excludes cyclic lists !!!

Test-Data Generation

- How to handle Recursion ?

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

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



query contract (modifiesOnly({})):

definition $pre_{sum}(l) \equiv True$

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

definition $sum(l) \equiv arb\{r \mid pre_{sum}(l) \wedge post_{sum}(l,r)\}$

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.

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))

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:

$X.sum() \equiv Y$ (for some $X \in LList$, i.e. $X \neq null$)

$\equiv inv_{LList}(X) \wedge pre_{sum}(X) \wedge post_{sum}(X,Y)$

where:

$pre_{sum}(X) \equiv true$

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

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

Test-Data Generation

- DNF computation yields already the test cases:

$X.sum() \equiv Y$ (for some $X \in LList$, i.e. $X \neq null$)

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

$\equiv$ (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))

$\equiv$ (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

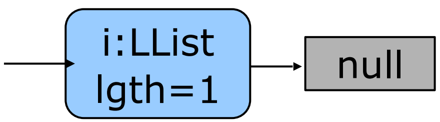
∧ X.next∈LList ∧ inv_{LList_Core}(X.next)

∧ 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:

$X.sum() \equiv Y$ (for some $X \in LList$, i.e. $X \neq null$)

$\Longrightarrow$ … ≡ …

≡ (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:

$X.sum() \equiv Y$ (for some $X \in LList$, i.e. $X \neq null$)

$\Longrightarrow$ ... $\equiv$ ...

$\equiv$ (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))
```

New Test-Case!!

Test-Data Generation

- Intermediate Summary: test-cases known so far ?

X	Y
→ i:LList lgth=1 → null	1
→ i:LList lgth=2 → i:LList lgth=1 → null	2
...	...

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 recursive specification:

  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 or precondition, if you use "blind" random-generation of input...

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 !