



POLYTECH[®]
PARIS-SUD

Cycle Ingénieur – 2^{ème} année
Département Informatique

Verification and Validation

Part IV : Test Intro

Burkhart Wolff

Département Informatique
Université Paris-Sud / Orsay

Difference between Validation and Verification

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

Difference between 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 ?

Difference between 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 ?

Difference between 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 ?

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

How to do Validation ?

- Tests and Experiments ...

How to do Verification ?

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

Test in the SE Process

- General questions for verification in a process:
 - How to select test-data ? To which purpose ?
 - How to focus verification activities?
Where to verify formally, and
where to test, and when did we test enough?

Note: The quality of a test does not increase necessarily by the number of test-cases !
 - Automation ? Tools ?

Verification Costs

- ❑ 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 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 emphasis: 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.

Taxonomy: Static / Dynamic Tests

- ❑ **static:** running a program before deployment on data carefully constructed by the analyst (in a testing environ.)
 - 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 analysis and debugging

Taxonomy: Unit / Sequence / Reactive Tests

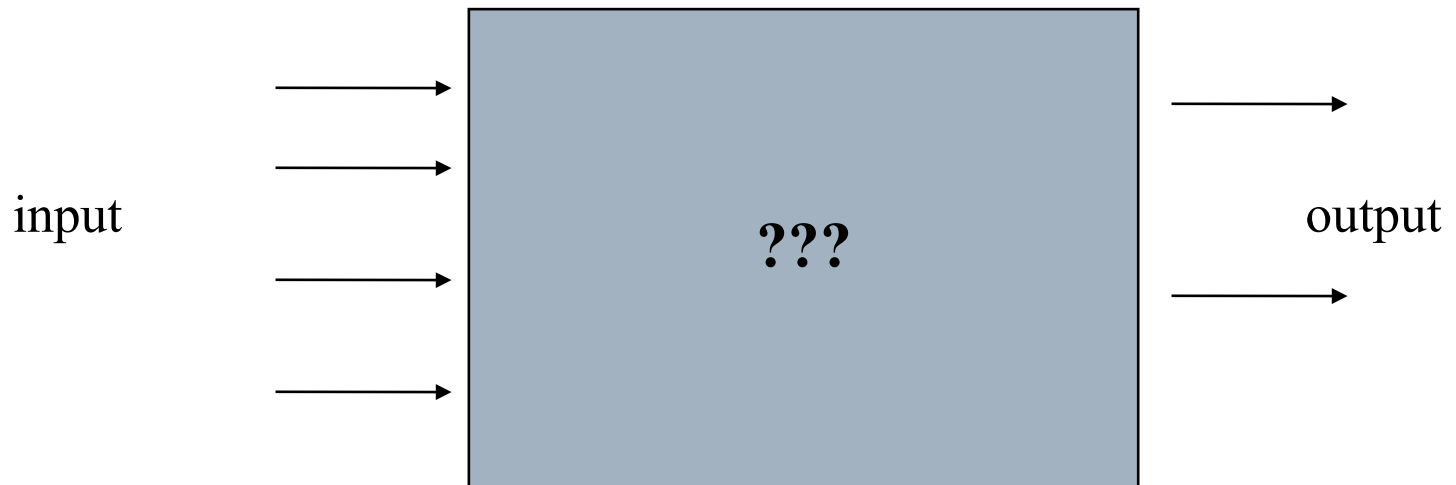
- ❑ **unit**: testing of a local component (function, module), typically only one step of the underlying state. (In functional programs, that's essentially all what you have to do!)
- ❑ **sequence**: testing of a local component (function, module), but typically sequences of executions, which typically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

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 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  $\wedge$  0 < b  $\wedge$  0 < c
inv  c  $\leq$  a + b  $\wedge$  a  $\leq$  b + c  $\wedge$  b  $\leq$  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  $\wedge$  t.b=t.c  $\rightarrow$  result=equ
  post (t.a $\neq$ t.b  $\vee$  t.b $\neq$ t.c)  $\wedge$ 
      (t.a=t.b  $\vee$  t.b=t.c  $\vee$  t.a=t.c)  $\rightarrow$  result=iso
  post (t.a $\neq$ t.b  $\vee$  t.b $\neq$ t.c  $\vee$  t.a $\neq$ t.c)  $\rightarrow$  result=arb
```

Intuitive Test-Data Generation

- Consider the test specification (the “Test Case”):

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

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

Note that we define $\text{mk}(0,0,0)$ to be 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 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?

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“ ?

Functional Dynamic Unit Test : an example

How to perform Runtime-Test?

Moreover, compile:

```
context C::m(a1:C1, ..., an:Cn)  
pre      : P(self, a1, ..., an)  
post     : Q(self, a1, ..., an, result)
```

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

```
check_C(); check_C1(); ... ; check_Cn();  
assert(P(self, a1, ..., an));  
result=run_m(self, a1, ..., an);  
assert(Q(self, a1, ..., an, result));
```


Intuitive Test-Data Generation

- Consider the test specification (the “Test Case”):

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

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

Note that we define $\text{mk}(0,0,0)$ to be invalid, as well as all other invalid triangles ...

Functional Dynamic Unit Test : Problems

- ❑ Thus, any violation of an invariant, a pre-condition or a post-condition is detected.
- ❑ If a violation occurs within an execution of a method, the error is precisely reported.
- ❑ On the other hand – runtime checking is **post-hoc**. Only when a problem occurred, we know where. And we need the **complete** program.
- ❑ Inefficiencies can be partly overcome by optimized compilations.

Functional Unit Test : An Example

We add the constraints of the analysis:

```
inv  0 < a  $\wedge$  0 < b  $\wedge$  0 < c
inv  c  $\leq$  a + b  $\wedge$  a  $\leq$  b + c  $\wedge$  b  $\leq$  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  $\wedge$  t.b=t.c  $\rightarrow$  result=equ
  post (t.a $\neq$ t.b  $\vee$  t.b $\neq$ t.c)  $\wedge$ 
      (t.a=t.b  $\vee$  t.b=t.c  $\vee$  t.a=t.c)  $\rightarrow$  result=iso
  post (t.a $\neq$ t.b  $\vee$  t.b $\neq$ t.c  $\vee$  t.a $\neq$ t.c)  $\rightarrow$  result=arb
  post modifiesOnly({})
```

Functional Dynamic Unit Test : Problems

- ❑ Thus, any violation of an invariant, a pre-condition or a post-condition is detected.
- ❑ If a violation occurs within an execution of a method, the error is precisely reported.
- ❑ On the other hand – runtime checking is **post-hoc**. Only when a problem occurred, we know where. And we need the **complete** program.
- ❑ Inefficiencies can be partly overcome by optimized compilations.

Functional Dynamic Unit Test : an example

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 Dynamic Unit Test : an example

How to perform Runtime-Test?

Well, compile for class invariants:

context X:

inv $l_1 : C_1, \dots,$

inv $l_n : C_n$

to some checking code (with assert as in Java/Junit, assert.h in C, ...)

```
check_X() = assert(C1); ... ; assert(Cn)
```

Functional Dynamic Unit Test : an example

How to perform Runtime-Test?

Moreover, compile for contracts:

```
context C::m(a1:C1, ..., an:Cn)  
pre      : P(self, a1, ..., an)  
post     : Q(self, a1, ..., an, result)
```

to some checking code (with assert as in Java/Junit, assert.h in C, ...)

```
check_C(); check_C1(); ... ; check_Cn();  
assert(P(self, a1, ..., an));  
result=run_m(self, a1, ..., an);  
assert(Q(self, a1, ..., an, result));
```

Functional Dynamic Unit Test : Problems

- ❑ Thus, any violation of an invariant, a pre-condition or a post-condition is detected.
- ❑ If a violation occurs within an execution of a method, the error is precisely reported.
- ❑ On the other hand – runtime checking is **post-hoc**. Only when a problem occurred, we know where. And we need the **complete** program.
- ❑ Inefficiencies can be partly overcome by optimized compilations.

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 alleviate 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“ ?

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:

$$\text{mk}(x,y,z).\text{isTriangle}() = r$$

$$\begin{aligned} \equiv & \text{inv}_{\text{Triangle}}(\sigma) \wedge \text{pre}_{\text{isTriangle}}(\text{mk}(x,y,z))(\sigma) \wedge \\ & \text{inv}_{\text{Triangle}}(\sigma\{r\}) \wedge \text{post}_{\text{isTriangle}}(\text{mk}(x,y,z),r)(\sigma,\sigma\{r\}) \\ & (* \text{ see semantics in MOAL II, page 22. } *) \end{aligned}$$

Some Facts:

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

$$\text{inv}_{\text{Triangle}}(\sigma) = \text{inv}_{\text{Triangle}}(\sigma\{r\})$$

- From $\text{mk}(x,y,z) \neq \text{null}$ (see `preisTriangle`) and from $\text{inv}_{\text{Triangle}}(\sigma)$ and $\text{mk}(x,y,z) \in \text{Triangle}(\sigma)$ follows that:

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

Test-Data Generation

- Recall the test specification:

$$\text{mk}(x,y,z).\text{isTriangle}() = r$$

$$\begin{aligned} \equiv & \text{inv}_{\text{Triangle}}(\sigma) \wedge \text{pre}_{\text{isTriangle}}(\text{mk}(x,y,z))(\sigma) \wedge \\ & \text{inv}_{\text{Triangle}}(\sigma\{r\}) \wedge \text{post}_{\text{isTriangle}}(\text{mk}(x,y,z),r)(\sigma,\sigma\{r\}) \\ & (* \text{ see semantics in MOAL II, page 22. } *) \end{aligned}$$

Some Facts:

- $\text{arb} \neq \text{equ} \neq \text{iso}$
- $\text{post}_{\text{isTriangle}}(\text{mk}(x,y,z),r)(\sigma,\sigma)$ can be simplified to:

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

Test-Data Generation

□ Summing up:

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

$\equiv inv_{Triangle}(\sigma) \wedge pre_{isTriangle}(mk(x,y,z))(\sigma) \wedge$
 $inv_{Triangle}(\sigma\{r\}) \wedge post_{isTriangle}(mk(x,y,z),r)(\sigma,\sigma\{r\})$
(* see semantics of MOAL II, page 22. *)

\Rightarrow (* the discussed facts *)

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

Test-Data Generation

- Recall the test specification:

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

\equiv (* elimination \rightarrow , deMorgan*)

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

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

- $r = \text{iso}$
- $r = \text{arb}$
- $r = \text{equ} \quad ???$

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

- $A_1 \wedge \dots \wedge A_i \wedge r = \text{iso}$
- $B_1 \wedge \dots \wedge B_k \wedge r = \text{arb}$
- $C_1 \wedge \dots \wedge C_l \wedge r = \text{equ} \quad ???$

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:

$$\begin{aligned}(A_1 \vee A_2) \wedge (B_1 \vee B_2) &= (A_1 \wedge (B_1 \vee B_2)) \vee (A_2 \wedge (B_1 \vee B_2)) \\ &= (A_1 \wedge B_1) \vee (A_2 \wedge B_1) \vee (A_1 \wedge B_2) \vee (A_2 \wedge B_2)\end{aligned}$$

$$\begin{aligned}(A_1 \vee A_2 \vee A_3) \wedge (B_1 \vee B_2 \vee B_3) \wedge (C_1 \vee C_2 \vee C_3) \\ &= \dots \\ &= (A_1 \wedge B_1 \wedge C_1) \vee (A_1 \wedge B_1 \wedge C_2) \vee (A_1 \wedge B_1 \wedge C_3) \vee \\ &\quad (A_2 \wedge B_1 \wedge C_1) \vee (A_2 \wedge B_1 \wedge C_2) \vee (A_2 \wedge B_1 \wedge C_3) \vee \\ &\quad \dots \\ &\quad (A_1 \wedge B_3 \wedge C_3) \vee (A_2 \wedge B_3 \wedge C_3) \vee (A_3 \wedge B_3 \wedge C_3)\end{aligned}$$

Test-Data Generation

- Recall the test specification:

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

- ≡ (* generalized distribution 2nd/3rd line *)

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

Test-Data Generation

- Recall the test specification:

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

- ≡ (* elimination contradictions *)

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

Test-Data Generation

- Recall the test specification:

$$\begin{aligned} & \dots \\ \equiv & \text{ (* elimination contradictions *)} \\ & \text{inv} \wedge \\ & \left((x \neq y \wedge y = z) \vee (x \neq y \wedge x = z) \vee (x \neq y \wedge r = \text{arb}) \vee \right. \\ & \quad (y \neq z \wedge x = y) \vee (y \neq z \wedge x = z) \vee (y \neq z \wedge r = \text{arb}) \vee \\ & \quad \left. (r = \text{equ} \wedge x = y) \vee (r = \text{equ} \wedge y = z) \vee (r = \text{equ} \wedge x = z) \right) \wedge \\ & \left((x = y \wedge y = z) \vee (x \neq y \wedge y \neq z \wedge x \neq z) \vee r = \text{iso} \right) \end{aligned}$$

Test-Data Generation

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

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

Test-Data Generation

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

$$\begin{aligned}
 & \text{inv } \wedge \\
 & \left(\cancel{(\cancel{x \neq y} \wedge y = z \wedge x = y \wedge y = z)} \vee (\cancel{x \neq y} \wedge x = z \wedge \right. \\
 & \qquad \qquad \qquad \left. \cancel{x = y} \wedge y = z) \vee (\cancel{x \neq y} \wedge r = \text{arb} \wedge x = y \wedge y = z) \vee \right. \\
 & \left(\cancel{y \neq z} \wedge x = y \wedge x = y \wedge y = z) \vee (\cancel{y \neq z} \wedge x = z \wedge \right. \\
 & \qquad \qquad \qquad \left. \cancel{x = y} \wedge y = z) \vee (\cancel{y \neq z} \wedge r = \text{arb} \wedge x = y \wedge y = z) \vee \right. \\
 & \left(r = \text{equ} \wedge x = y \wedge x = y \wedge y = z) \vee \dots \vee (r = \text{equ} \wedge \right. \\
 & \qquad \qquad \qquad \left. y = z \wedge x = y \wedge y = z) \vee (r = \text{equ} \wedge x = z \wedge x = y \wedge y = z) \right) \vee \\
 & \left(\cancel{(\cancel{x \neq y} \wedge y = z \wedge x \neq y \wedge y \neq z \wedge x \neq z)} \vee (\cancel{x \neq y} \wedge x = z \wedge x \neq y \wedge y \neq z \wedge x \neq z)} \vee (x \neq y \wedge r = \text{arb} \wedge \right. \\
 & \left. \cancel{x \neq y} \wedge y \neq z \wedge x \neq z) \vee (\cancel{y \neq z} \wedge x = y \wedge x \neq y \wedge y \neq z \wedge x \neq z)} \vee (\cancel{y \neq z} \wedge x = z \wedge x \neq y \wedge y \neq z \wedge \right. \\
 & \left. \cancel{x \neq z)} \vee \dots \vee (\cancel{y \neq z} \wedge r = \text{arb} \wedge x \neq y \wedge y \neq z \wedge x \neq z) \vee (r = \text{equ} \wedge x = y \wedge x \neq y \wedge y \neq z \wedge x \neq z) \vee (r \right. \\
 & \left. = \text{equ} \wedge y = z \wedge x \neq y \wedge y \neq z \wedge x \neq z) \vee (r = \text{equ} \wedge x = z \wedge x \neq y \wedge y \neq z \wedge \cancel{x \neq z}) \right) \vee \\
 & \left((x \neq y \wedge y = z \wedge r = \text{iso}) \vee (x \neq y \wedge x = z \wedge r = \text{iso}) \vee (\cancel{x \neq y} \wedge r = \text{arb} \wedge r = \text{iso}) \vee \right. \\
 & \left. \vee (y \neq z \wedge x = y \wedge r = \text{iso}) \vee (y \neq z \wedge x = z \wedge r = \text{iso}) \vee (\cancel{y \neq z} \wedge r = \text{arb} \wedge r = \text{iso}) \vee \right. \\
 & \left. \vee (\cancel{r = \text{equ}} \wedge x = y \wedge r = \text{iso}) \vee (\cancel{r = \text{equ}} \wedge y = z \wedge r = \text{iso}) \vee (\cancel{r = \text{equ}} \wedge x = z \wedge r = \text{iso}) \right)
 \end{aligned}$$

Test-Data Generation

□ \equiv (* cleanup, distribution *)

$$(\text{inv} \wedge x=y \wedge x=y \wedge y=z \wedge r=\text{equ}) \vee \quad (1)$$

$$(\text{inv} \wedge x \neq y \wedge y \neq z \wedge x \neq z \wedge r=\text{arb}) \vee \quad (2)$$

$$(\text{inv} \wedge x \neq y \wedge y=z \wedge r=\text{iso}) \vee \quad (3)$$

$$(\text{inv} \wedge x \neq y \wedge x=z \wedge r=\text{iso}) \vee \quad (4)$$

$$(\text{inv} \wedge y \neq z \wedge x=y \wedge r=\text{iso}) \vee \quad (5)$$

$$(\text{inv} \wedge y \neq z \wedge x=z \wedge r=\text{iso}) \quad (6)$$

□ **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

- ❑ 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 Coding 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:

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

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:

$$\begin{aligned} & \text{inv} \wedge \\ & (A \wedge B \longrightarrow r=\text{equ}) \wedge \\ & ((\neg A \vee \neg B) \wedge (A \vee B \vee C) \longrightarrow r=\text{iso}) \wedge \\ & (\neg A \wedge \neg B \wedge \neg C \longrightarrow r=\text{arb}) \end{aligned}$$

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“

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

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

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

➤ ... 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:

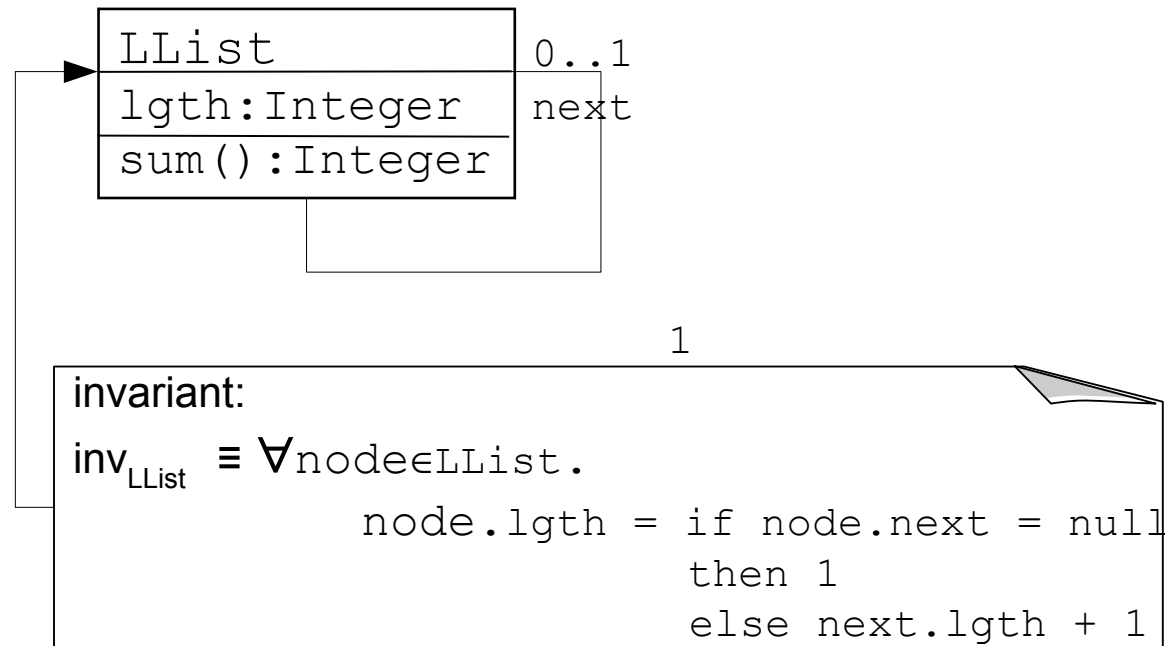
➤ at the level of data

Test-Data Generation

□ How to handle Recursion ?

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

- at the level of data

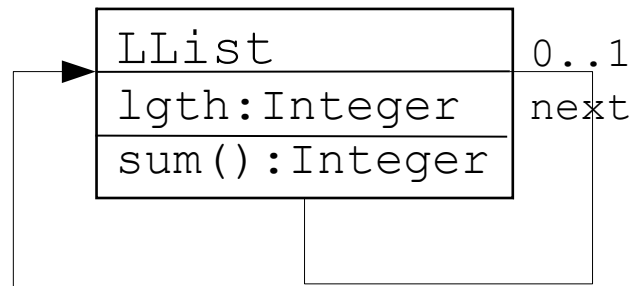


Test-Data Generation

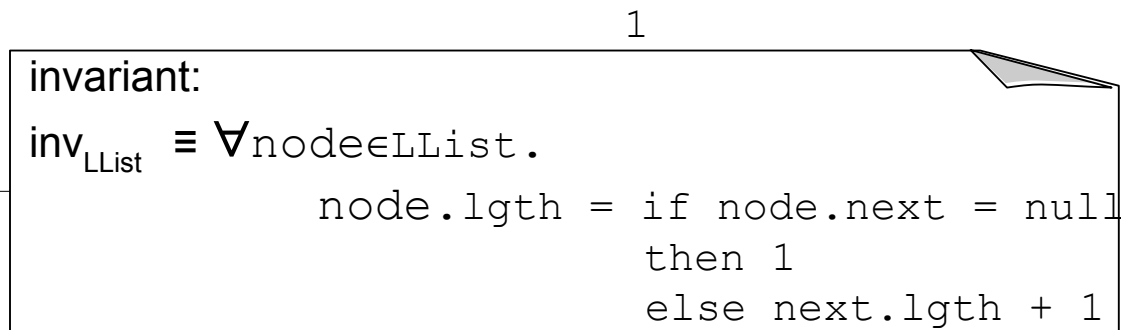
□ How to handle Recursion ?

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

- at the level of data



Note that this excludes cyclic lists !!!

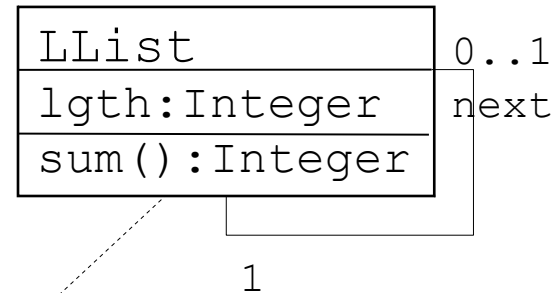


Test-Data Generation

□ How to handle Recursion ?

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

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

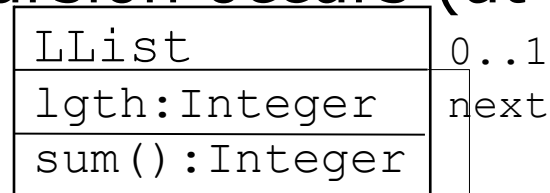


```
query contract (modifiesOnly({})):
definition presum(l) ≡ True
definition postsum(l, res) ≡ res = if l.next = null then l.lgth
                                else l.lgth + l.next.sum()
definition sum(l) ≡ arb{r | presum(l) ∧ postsum(l, r)}
```

Test-Data Generation

□ How to handle Recursion ?

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



Note that $\text{arb}(S)$ gives an arbitrary member of S :

$\text{arb}(S) \in S$.

Since from $x = \text{arb}(\{y\})$ follows

$x = y$;

thus $\text{sum}(l)$ is (uniquely) defined.

The (σ, σ') convention applies.

of oper-
ands
calls ...)

$\text{sum}(l) \equiv \text{True}$

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

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

Test-Data Generation

- Prerequisite: We present the invariant as recursive predicate.

definition $\text{inv}_{\text{LList_Core}}^n \sigma \equiv (\text{n.lgth}(\sigma) = \text{if } \text{n.next}(\sigma) = \text{null} \text{ then } 1$
 $\text{else } \text{n.next.lgth}(\sigma) + 1)$

we have:

$$\text{inv}_{\text{LList}}(\sigma) = \forall n \in \text{LList}(\sigma). \text{inv}_{\text{LList_Core}}^n \sigma$$

and

$$\text{inv}_{\text{LList_Core}}(n)(\sigma) = (\text{if } \text{n.next}(\sigma) = \text{null} \text{ then } \text{n.lgth}(\sigma) = 1$$
$$\text{else } \text{n.lgth}(\sigma) = \text{n.next.lgth}(\sigma) + 1$$
$$\wedge \text{n.next}(\sigma) \in \text{LList}(\sigma)$$
$$\wedge \text{inv}_{\text{LList_Core}}(\text{n.next})(\sigma))$$

(under the assumption that $n \in \text{LList}(\sigma)$)

Furthermore we have:

$$\text{sum}(l)(\sigma', \sigma) = \text{if } l.\text{next}(\sigma) = \text{null} \text{ then } l.\text{lgth}(\sigma)$$
$$\text{else } l.\text{lgth}(\sigma) + \text{sum}(l.\text{next})(\sigma', \sigma)$$

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

Test-Data Generation

- Consider the test specification:

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

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

where:

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

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

$$\equiv (X.\text{next} = \text{null} \wedge Y = X.\text{lgth})$$

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

Test-Data Generation

- DNF computation yields already the test cases:

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

$\Rightarrow \text{inv}_{\text{LList_Core}}(X) \wedge \text{post}_{\text{sum}}(X, Y)$

$\equiv (\text{if } X.\text{next}=\text{null} \text{ then } X.\text{lgth} = 1$
 $\text{else } X.\text{lgth} = X.\text{next}.\text{lgth}+1 \wedge X.\text{next} \in \text{LList} \wedge \text{inv}_{\text{LList_Core}}(X.\text{next})) \wedge$
 $(\text{if } X.\text{next} = \text{null} \text{ then } Y = X.\text{lgth}$
 $\text{else } Y = X.\text{lgth} + \text{sum}(X.\text{next}))$

\equiv (DNF)

$(X.\text{next}=\text{null} \wedge X.\text{lgth}=1 \wedge Y = X.\text{lgth})$
 $\vee (X.\text{next} \neq \text{null} \wedge X.\text{lgth} = X.\text{next}.\text{lgth}+1$
 $\wedge X.\text{next} \in \text{LList} \wedge \text{inv}_{\text{LList_Core}}(X.\text{next})$
 $\wedge Y = X.\text{lgth} + \text{sum}(X.\text{next}))$

Test-Data Generation

- DNF computation yields already the test cases:

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

$\Rightarrow \text{inv}_{\text{LList_Core}}(X) \wedge \text{post}_{\text{sum}}(X, Y)$

\equiv (if $X.\text{next}=\text{null}$ then $X.\text{lgth} = 1$
else $X.\text{lgth} = X.\text{next}.\text{lgth}+1 \wedge X.\text{next} \in \text{LList} \wedge \text{inv}_{\text{LList_Core}}$
(if $X.\text{next} = \text{null}$ then $Y = X.\text{lgth}$
else $Y = X.\text{lgth} + \text{sum}(X.\text{next})$)

\equiv (if c then C else D elim, DNF)

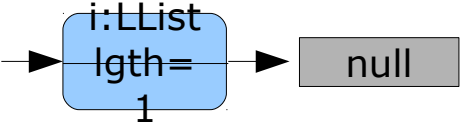
$(X.\text{next}=\text{null} \wedge X.\text{lgth}=1 \wedge Y = X.\text{lgth})$

$\vee (X.\text{next} \neq \text{null} \wedge X.\text{lgth} = X.\text{next}.\text{lgth}+1$
 $\wedge X.\text{next} \in \text{LList} \wedge \text{inv}_{\text{LList_Core}}(X.\text{next})$
 $\wedge Y = X.\text{lgth} + \text{sum}(X.\text{next}))$

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.

$$\text{inv}_{\text{LList_Core}}(n) = (\text{if } n.\text{next}=\text{null} \text{ then } n.\text{lgth} = 1 \\ \text{else } n.\text{lgth} = n.\text{next}.\text{lgth} + 1 \\ \wedge n.\text{next} \in \text{LList} \wedge \text{inv}_{\text{LList_Core}}(n.\text{next}))$$

- $\text{sum}(l) = \text{if } l.\text{next}=\text{null} \text{ then } l.\text{lgth} \\ \text{else } l.\text{lgth} + \text{sum}(l.\text{next})$

$$\text{sum}(l) = \text{if } X.\text{next}.\text{next}=\text{null} \text{ then } X.\text{next}.\text{lgth} \\ \text{else } X.\text{next}.\text{lgth} + \text{sum}(X.\text{next}.\text{next})$$

Test-Data Generation

- DNF computation yields already the test cases:

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

$\Rightarrow \dots \equiv \dots$

\equiv (unfolding sum and $\text{inv}_{\text{LList_Core}}$)

$(X.\text{next}=\text{null} \wedge X.\text{lgth}=1 \wedge Y = X.\text{lgth})$

$\vee (X.\text{next} \neq \text{null} \wedge X.\text{lgth}=X.\text{next}.\text{lgth}+1 \wedge X.\text{next} \in \text{LList}$

$\wedge (\text{if } X.\text{next}.\text{next}=\text{null} \text{ then } X.\text{next}.\text{lgth} = 1$

$\text{else } X.\text{next}.\text{lgth} = X.\text{next}.\text{next}.\text{lgth} + 1$

$\wedge X.\text{next}.\text{next} \in \text{LList} \wedge \text{inv}_{\text{LList_Core}}(X.\text{next}.\text{next}))$

$\wedge (Y = X.\text{lgth}+(\text{if } X.\text{next}.\text{next}=\text{null} \text{ then } X.\text{next}.\text{lgth}$

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

Test-Data Generation

- DNF computation yields already the test cases:

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

$\Rightarrow \dots \equiv \dots$

\equiv (DNF partial)

$(X.next = \text{null} \wedge X.lgth = 1 \wedge Y = X.lgth)$

$\vee (X.next \neq \text{null} \wedge X.lgth = X.next.lgth + 1 \wedge X.next \in \text{LList}$

$\wedge ((X.next.next = \text{null} \wedge X.next.lgth = 1 \wedge$
 $Y = X.lgth + X.next.lgth)$

$\vee (X.next.next \neq \text{null} \wedge X.next.lgth = X.next.next.lgth + 1$
 $\wedge X.next.next \in \text{LList} \wedge \text{inv}_{\text{LList_Core}}(X.next.next)$
 $\wedge Y = X.lgth + X.next.lgth + \text{sum}(X.next.next))$
 $)$

Test-Data Generation

- DNF computation yields already the test cases:

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

$\Rightarrow \dots \equiv \dots$

\equiv (DNF partial)

$(X.next = \text{null} \wedge X.lgth = 1 \wedge Y = X.lgth)$

$\vee (X.next \neq \text{null} \wedge X.lgth = X.next.lgth + 1 \wedge X.next \in \text{LList}$
 $\wedge X.next.next = \text{null} \wedge X.next.lgth = 1 \wedge$
 $Y = X.lgth + X.next.lgth)$

$\vee (X.next \neq \text{null} \wedge X.lgth = X.next.lgth + 1 \wedge X.next \in \text{LList}$
 $\wedge X.next.next \neq \text{null} \wedge X.next.lgth = X.next.next.lgth + 1$
 $\wedge X.next.next \in \text{LList} \wedge \text{inv}_{\text{LList_Core}}(X.next.next)$
 $\wedge Y = X.lgth + X.next.lgth + \text{sum}(X.next.next))$

Test-Data Generation

- DNF computation yields already the test cases:

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

$\Rightarrow \dots \equiv \dots$

\equiv (DNF partial)

$(X.next = \text{null} \wedge X.lgth = 1 \wedge Y = X.lgth)$

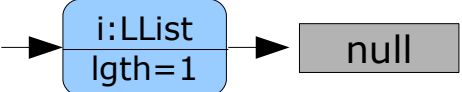
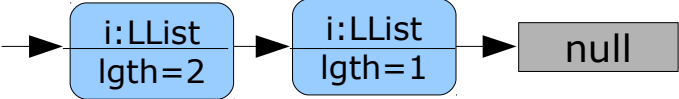
$\vee (X.next \neq \text{null} \wedge X.lgth = X.next.lgth + 1 \wedge X.next \in \text{LList}$
 $\wedge X.next.next = \text{null} \wedge X.next.lgth = 1 \wedge$
 $Y = X.lgth + X.next.lgth)$

$\vee (X.next \neq \text{null} \wedge X.lgth = X.next.lgth + 1 \wedge X.next \in \text{LList}$
 $\wedge X.next.next \neq \text{null} \wedge X.next.lgth = X.next.next.lgth + 1$
 $\wedge X.next.next \in \text{LList} \wedge \text{inv}_{\text{LList_Core}}(X.next.next)$
 $\wedge Y = X.lgth + X.next.lgth + \text{sum}(X.next.next))$

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 ...

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

where we choose as “complexity measure” $|X|$ just $X.\text{lgth}$ and k (the number of unfoldings) was 2 ...

Test-Data Generation

- Coverage Criterion for a Test:

$$\text{DNF}_k$$

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

In our example, the “complexity measure” is just the length of the LLists. It could be the depth in trees or ...

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