



*Cycle Ingénieur – 2<sup>ème</sup> année*  
*Département Informatique*

# Verification and Validation

## Part IV : System Test II

Burkhart Wolff

Département Informatique  
Université Paris-Sud / Orsay

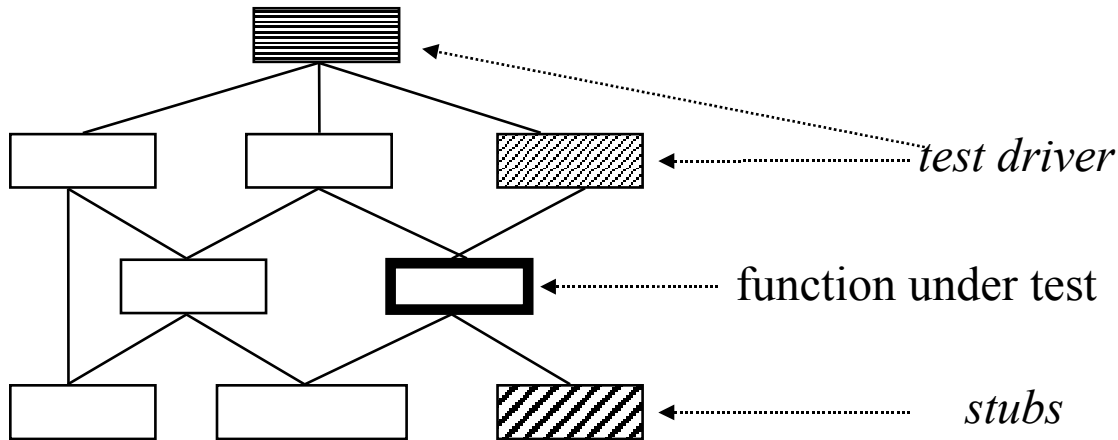
# Towards **Static** Specification-based Unit Test

---

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

# Drivers and Stubs (Lanceurs et Bouchons)

---



« *How to test incomplete modules ?* »

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

# Difficulties with Static Unit Tests so far

---

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

# Difficulties with Static Unit Tests so far

---

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

We have to decide on adequacy criteria in advance.  
This can be:

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

# Generating Test-Data by Example

---

- Consider the test specification:

`mk(x,y,z).isTriangle() ≡ X`

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

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

# Intuitive Test-Data Generation

---

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

# Test-Data Generation

---

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

# Test-Data Generation

---

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

Well, lets see and calculate ...

# Test-Data Generation

---

- Recall the test specification:

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

# Test-Data Generation

---

- Recall the test specification:

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

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

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

# Test-Data Generation

---

- Recall the test specification:

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

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

(\* by definition of oclIsUndefined \*)

# Test-Data Generation

---

- Recall the test specification:

`mk(x,y,z).isTriangle() ≡ X`

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

# Test-Data Generation

---

- Recall the test specification:

`mk(x,y,z).isTriangle() ≡ X`

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

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

# Test-Data Generation

---

- Recall the test specification:

`mk(x,y,z).isTriangle() ≡ X`

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

# Test-Data Generation

---

- Recall the test specification:

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

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

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

# Test-Data Generation

---

- Recall the test specification:

**mk(x,y,z).isTriangle() ≡ X**

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

(\*by invariant \*)

# Test-Data Generation

---

- Recall the test specification:

...

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

```
in (x <= 0 and result = oclUndefined or
    y <= 0 and result = oclUndefined or
    z <= 0 and result = oclUndefined or
    z <= x + y and result = oclUndefined or
    x <= y + z and result = oclUndefined or
    y <= z + x and result = oclUndefined) or
```

```
(inv and x <> y and y <> z and x <> z and result = arbitrary) or
(inv and x = y and x <> z and y <> z and result = isosceles) or
(inv and y = z and y <> x and z <> x and result = isosceles) or
(inv and x = z and x <> y and z <> y and result = isosceles) or
(inv and x = y and y = z and result = equilateral))
```

```
end
```

# Test-Data Generation

---

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

# Test-Data Generation

---

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

# Test-Data Generation

---

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

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

# Test-Data Generation

---

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

Resulting Test (satisfying this formula !):

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

# Test-Data Generation

---

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

# Test-Data Generation

---

## □ Example Uniformity: Testcase

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

## Applied Uniformity Hypothesis:

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

# Test-Data Generation

---

- General Scheme of **Uniformity hypothesis:**

Formally:

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

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

# Test-Data Generation

---

- General Scheme of **Uniformity hypothesis:**

## Example: Testcase

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

Resulting Test (satisfying this formula !):

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

# Test-Data Generation

---

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

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

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

# Test-Data Generation

---

- How to handle Recursion ?

# Test-Data Generation

---

## □ How to handle Recursion ?

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

- at the level of data

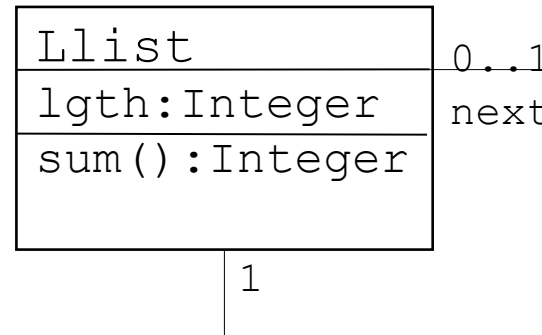
# Test-Data Generation

---

## ❑ How to handle Recursion ?

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

- at the level of data



```
context LList:
inv lgth = if next.oclIsUndefined()
    then 0
    else next.lgth + 1
    endif
```

# Test-Data Generation

---

## □ How to handle Recursion ?

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

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

```
context LList:sum()  
post result =  
    if next.oclIsUndefined()  
    then 0  
    else lgth +  
        next.lgth.sum()  
    endif
```

# Test-Data Generation

---

- Answer:  
apply **regularity hypothesis**.

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

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

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

# Test-Data Generation

---

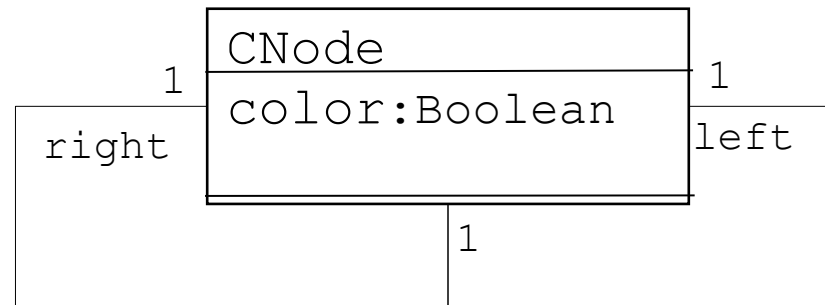
- ❑ Remarks on the regularity hypothesis:
  - main tool to reduce infinite number of test cases to a finite one (which may still contain infinitely many tests!)
  
  - is similar to an induction (corresponds to induction anchor, but leaves out the induction step)

# Test-Data Generation

---

- ❑ Problem:  
Invariants in OCL may describe graphs, not just trees.

Example:



```
context Cnode
```

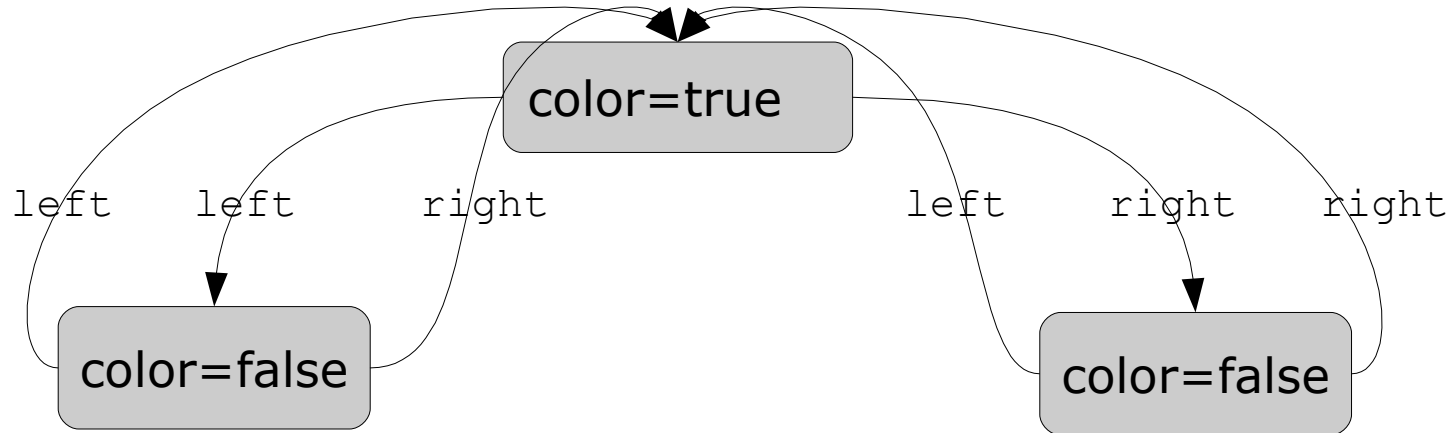
```
inv flip_strong:
```

```
(left.oclIsDefined() and left.color<>color) and  
(right.oclIsDefined() and right.color<>color)
```

# Test-Data Generation

---

- An Object Model for this may look like this:



a natural *measure* on graphs is the number of different nodes.

# Test-Data Generation

---

- Cycles in Graphs correspond to equalities:  
For our example:

```
S = {self.left.left = self,  
      self.left.right = self,  
      self.right.left = self,  
      self.right.right = self}
```

We call this set of path equalities  
a “3-congruence for Cnode” or  $S \in C_{CNode}(3)$

# Test-Data Generation

---

- ❑ Construct a  $C_{CNode}(2)$

```
S = {self.left = self.right,  
     self.left.left = self}
```

Are there valid object models for  $C_{CNode}(i)$  ( $i \leq 3$ )?

# Test-Data Generation

---

- ❑ This gives a procedure for enumerating congruences up to  $k$ :
  - enumerate the possible paths up to “length”  $k+1$  (self, self.left, ....) (overapprox !)
  - create equations for identifying paths (Class-Model conform and type-correct!) upto the point when there is no path of length  $k+1$  (modulo equality) any more
  - count the number  $k$  of different objects in the object model

# Test-Data Generation

---

- How to compute *valid* object models up to k?

Trick: We convert the invariants in a recursive predicate (the *invariant predicate*)

Example:

```
flip_strong(self) =  
    (self.left.oclIsDefined() and  
     self.left.color<>self.color)  
and (self.right.oclIsDefined() and  
     self.right.color<>self.color)  
and flip_strong(self.left)  
and flip_strong(self.left)
```

# Test-Data Generation

---

- How to compute *valid* object models up to k?

Trick 2: we pick an  $S \in C_{\text{CNode}}(n)$  ( $n \leq k$ )  
and built  $S^0$  by conjoining its elements:

Example:

```
 $S^0(\text{self}) = \text{self.left.left} = \text{self}$  and  
                   $\text{self.left.right} = \text{self}$  and  
                   $\text{self.right.left} = \text{self}$  and  
                   $\text{self.right.right} = \text{self}$ 
```

# Test-Data Generation

---

- How to compute *valid* object models up to k?

Trick 3: we conjoin the invariant predicate  
with our  $S^0$  and compute:

Example:

```
flip_strong(self) and  $S^0$ (self)  
≡
```

# Test-Data Generation

---

- How to compute *valid* object models up to k?

Trick 3: we conjoin the invariant predicate  
with our  $S^0$  and compute:

Example:

```
flip_strong(self) and  $S^0$ (self)  
≡ flip_strong(self) and flip_strong(self) and  $S^0$ (self)  
≡
```

# Test-Data Generation

---

- How to compute *valid* object models up to k?

Trick 3: we conjoin the invariant predicate  
with our  $S^0$  and compute:

Example:

```
flip_strong(self) and  $S^0$ (self)
≡ flip_strong(self) and flip_strong(self) and  $S^0$ (self)
≡ flip_strong(self) and
self.left.oclIsDefined() and self.left.color<>self.color and
self.right.oclIsDefined() and self.right.color<>self.color and
and flip_strong(self.left) and flip_strong(self.right)
and self.left.left = self and self.left.right = self and
self.right.left = self and self.right.right = self
```

# Test-Data Generation

---

## ❑ Interupt !

We use a mathematical notation for derivations (built over strong equality  $\equiv$ ):

<code>x and y</code>	$\equiv$	<code>x ^ y</code>
<code>x or y</code>	$\equiv$	<code>x v y</code>
<code>x implies y</code>	$\equiv$	<code>x -&gt; y</code>
<code>not x</code>	$\equiv$	<code>¬x</code>
<code>x.oclIsDefined()</code>	$\equiv$	<code>∂x</code>
<code>x.oclIsUndeined()</code>	$\equiv$	<code>¬∂x</code>

# Test-Data Generation

---

- We compute:

```
flip_strong(s)  $\wedge$  S0(s)
≡ flip_strong(s)  $\wedge$ 
  ∂s.left  $\wedge$  s.left.color<>s.color  $\wedge$ 
  ∂s.right  $\wedge$  s.right.color<>s.color
 $\wedge$  flip_strong(self.left)  $\wedge$  flip_strong(self.right)
 $\wedge$  s.left.left = s  $\wedge$  s.left.right = s  $\wedge$ 
  s.right.left = s  $\wedge$  s.right.right = s
```

# Test-Data Generation

---

- We compute:

```
flip_strong(s)  $\wedge$  S0(s)
 $\equiv$  flip_strong(s)  $\wedge$ 
     $\partial$ s.left  $\wedge$  s.left.color<>s.color  $\wedge$ 
     $\partial$ s.right  $\wedge$  s.right.color<>s.color
 $\wedge$  flip_strong(s.left)  $\wedge$  flip_strong(s.right)
 $\wedge$  s.left.left = s  $\wedge$  s.left.right = s  $\wedge$ 
    s.right.left = s  $\wedge$  s.right.right = s
```

# Test-Data Generation

---

- We compute:

```
flip_strong(s)  $\wedge$  S0(s)
 $\equiv$  flip_strong(s)
 $\wedge$   $\partial$ s.left  $\wedge$  s.left.color<>s.color
 $\wedge$   $\partial$ s.right  $\wedge$  s.right.color<>s.color
 $\wedge$  flip_strong(s.left)
 $\wedge$   $\partial$ s.left.left  $\wedge$  s.left.left.color<>s.left.color
 $\wedge$   $\partial$ s.left.right  $\wedge$  s.left.right.color<>s.left.color
 $\wedge$  flip_strong(s.left.left)  $\wedge$  flip_strong(s.left.right)
 $\wedge$  flip_strong(s.right)
 $\wedge$   $\partial$ s.right.left  $\wedge$  s.right.left.color<>s.color
 $\wedge$   $\partial$ s.right.right  $\wedge$  s.right.right.color<>s.right.color
 $\wedge$  flip_strong(s.right.left)  $\wedge$  flip_strong(s.right.right)
 $\wedge$  s.left.left = s  $\wedge$  s.left.right = s  $\wedge$ 
    s.right.left = s  $\wedge$  s.right.right = s
```

# Test-Data Generation

---

- We compute:

```
flip_strong(s)  $\wedge$  S0(s)
 $\equiv$  flip_strong(s)
 $\wedge$   $\partial$ s.left  $\wedge$  s.left.color<>s.color
 $\wedge$   $\partial$ s.right  $\wedge$  s.right.color<>s.color
 $\wedge$  flip_strong(s.left)
 $\wedge$   $\partial$ s.left.left  $\wedge$  s.left.left.color<>s.left.color
 $\wedge$   $\partial$ s.left.right  $\wedge$  s.left.right.color<>s.left.color
 $\wedge$  flip_strong(s.left.left)  $\wedge$  flip_strong(s.left.right)
 $\wedge$  flip_strong(s.right)
 $\wedge$   $\partial$ s.right.left  $\wedge$  s.right.left.color<>s.color
 $\wedge$   $\partial$ s.right.right  $\wedge$  s.right.right.color<>s.right.color
 $\wedge$  flip_strong(s.right.left)  $\wedge$  flip_strong(s.right.right)
 $\wedge$  s.left.left = s  $\wedge$  s.left.right = s  $\wedge$ 
s.right.left = s  $\wedge$  s.right.right = s
```

# Test-Data Generation

---

- We compute:

```
flip_strong(s)  $\wedge$  S0(s)
 $\equiv$  flip_strong(s)
 $\wedge$   $\partial$ s.left  $\wedge$  s.left.color<>s.color
 $\wedge$   $\partial$ s.right  $\wedge$  s.right.color<>s.color
 $\wedge$  flip_strong(s.left)
 $\wedge$   $\partial$ s.left.left  $\wedge$  s.left.left.color<>s.left.color
 $\wedge$   $\partial$ s.left.right  $\wedge$  s.left.right.color<>s.left.color
 $\wedge$  flip_strong(s)  $\wedge$  flip_strong(s)
 $\wedge$  flip_strong(s.right)
 $\wedge$   $\partial$ s.right.left  $\wedge$  s.right.left.color<>s.color
 $\wedge$   $\partial$ s.right.right  $\wedge$  s.right.right.color<>s.right.color
 $\wedge$  flip_strong(s)  $\wedge$  flip_strong(s)
 $\wedge$  s.left.left = s  $\wedge$  s.left.right = s  $\wedge$ 
  s.right.left = s  $\wedge$  s.right.right = s
```

# Test-Data Generation

---

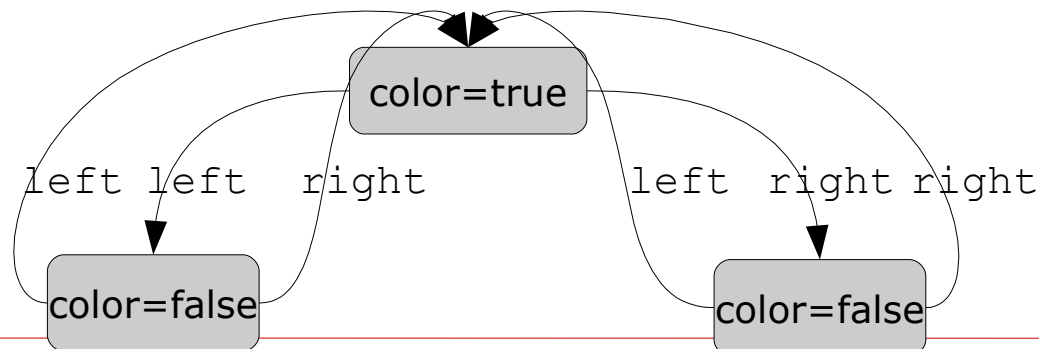
- We compute:

```
flip_strong(s)  $\wedge$  S0(s)
 $\equiv$  flip_strong(s)
 $\wedge$   $\partial$ s.left  $\wedge$  s.left.color<>s.color
 $\wedge$   $\partial$ s.right  $\wedge$  s.right.color<>s.color
 $\wedge$  flip_strong(s.left)
 $\wedge$   $\partial$ s.left.left  $\wedge$  s.left.left.color<>s.left.color
 $\wedge$   $\partial$ s.left.right  $\wedge$  s.left.right.color<>s.left.color
 $\wedge$  flip_strong(s.right)
 $\wedge$   $\partial$ s.right.left  $\wedge$  s.right.left.color<>s.color
 $\wedge$   $\partial$ s.right.right  $\wedge$  s.right.right.color<>s.right.color
 $\wedge$  s.left.left = s  $\wedge$  s.left.right = s  $\wedge$ 
    s.right.left = s  $\wedge$  s.right.right = s
```

# Test-Data Generation

---

- ❑ ... and we are saturated, i.e. any further unfolding of an recursive predicate will not produce further information.
- ❑ By skipping the recursive predicates and applying uniformity hypothesis, we can construct a witness for this formula, e.g.



# Test-Data Generation

---

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