Génie Logiciel Avancé

Part V : Black-Box Test

Burkhart Wolff
wolff@lri.fr
Towards **Static** Specification-based Unit Test

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

- How can testing be applied to incomplete programs?
Drivers and Stubs (Lanceurs et Bouchons)

« How to test incomplete modules ? »

- if non-existent, generate code for drivers:
  may be a random-tester against pre-conditions
  or a function running a pre-computed test-data-base.
- if non-existent, generate code for stubs:
  may be a random-tester against post-conditions
  or a “simple” version of the function to be computed.
Difficulties with Static Unit Tests so far

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

- When do we have tested enough? When is our test “adequate”? We have to decide on adequacy criteria in advance. This can be:
  - criteria on the coverage of the spec of the program
  - criteria on statistical models and an error model

- Some empirical observations:
  - No relation between detection order and detection difficulty
  - No relation between detection difficulty and correction
  - The more errors you found, the more you find more...
  - The quality of a test set is independent of its size.
Functional Unit Test : Triangle Revisited

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

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

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

end
```
Functional Unit Test : Triangle Revisited

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

```plaintext
context Triangles:
inv def : a.isInvalid() and b.isInvalid()...
inv pos : 0<a and 0<b and 0<c
inv triangle : a+b>c and b+c>a and c+a>b

context Triangle::isTriangle()
post equi : a=b and b=c implies result=equilateral
post iso :((a<>b or b<>c) and
           (a=b or b=c or a=c))implies result=isosceles
post default: (a<>b or b<>c) and
            (a<>b and b<>c and a<>c)
            implies result=arbitrary
```
Revision: Boolean Logic + some basic Rules

- not(a and b) = not a or not b  (* deMorgan1 *)
- not(a or b) = not a and not b  (* deMorgan2 *)
- a and (b or c) = (a and b) or (a and c)
- not(not a) = a
- a and b = b and a; a or b = b or a
- a and (b and c) = (a and b) and c
- a or (b or c) = (a or b) or c
- a implies b = (not a) or b
- (a=b and P(a)) = P(b)  (* one point rule *)

- let x = E in C(x) = C(E)  (* let elimination *)
- if c then C else D = (c and C) or (not c and D)
  = (c implies C) and (not c and D)
Generating Test-Data by Example

Consider the test specification:

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

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 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?
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:

  \[ \text{mk}(x,y,z).\text{isTriangle}() \equiv X \]
Test-Data Generation

- Recall the test specification:

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

  \[ \equiv (\text{mk}(x,y,z).\text{isTriangle}() \equiv \text{invalid}) \text{ or } \]
  \[ (\text{mk}(x,y,z).\text{isTriangle}() \equiv \text{arbitrary}) \text{ or } \]
  \[ (\text{mk}(x,y,z).\text{isTriangle}() \equiv \text{isosceles}) \text{ or } \]
  \[ (\text{mk}(x,y,z).\text{isTriangle}() \equiv \text{equilateral}) \]

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

- Recall the test specification:

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

\[
\equiv \ (\text{mk}(x,y,z).\text{isTriangle}().\text{isInvalid}()) \text{ or } \ (\text{mk}(x,y,z).\text{isTriangle}() \equiv \text{arbitrary}) \text{ or } \ (\text{mk}(x,y,z).\text{isTriangle}() \equiv \text{isosceles}) \text{ or } \ (\text{mk}(x,y,z).\text{isTriangle}() \equiv \text{equilateral})
\]

(* by definition of oclIsUndefined *)
Test-Data Generation

- Recall the test specification:

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

  \[ \equiv (mk(x,y,z).\text{isInvalid}() \text{ and result} \equiv \text{isInvalid}) \text{ or } \]
  
  let self = mk(x,y,z);
  
  a = self.a; b = self.b; c = self.c;
  
  in  self.isInvalid() \text{ and }
  
  ((post(self,result) \text{ and result} \equiv \text{arbitrary}) \text{ or }
  
  (post(self,result) \text{ and result} \equiv \text{isosceles}) \text{ or }
  
  (post(self,result) \text{ and result} \equiv \text{equilateral}))
  
  end

(*by post-condition of isTriangle*)
Test-Data Generation

- Recall the test specification:

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

\[
\equiv (mk(x, y, z).\text{isInvalid()} \land \text{result} \equiv \text{invalid}) \lor \\
\text{let self} = mk(x, y, z); \\
a = self.a; b = self.b; c = self.c; \\
in self.isInvalid() \\
(a<>b \land b<>c \land a<>c \land \text{result} \equiv \text{arbitrary}) \lor \\
((a=b \land a<>c \land b<>c) \lor \\
(b=c \land b<>a \land c<>a) \lor \\
(a=c \land a<>b \land c<>b)) \land \text{result} \equiv \text{isosceles}) \lor \\
(a=b \land b=c \land \text{result} \equiv \text{equilateral})
\]

(*by\ arbitrary<>\text{isosceles}<>\text{equilateral} \land \text{log. simplif.} *)
Test-Data Generation

- Recall the test specification:

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

\[ \equiv (\text{mk}(x, y, z).\text{isInvalid()} \text{ and result } \equiv \text{invalid}) \text{ or } (\text{mk}(x, y, z).\text{isInvalid()} \text{ and } \text{let } a = x; b = y; c = z; \text{ in } ((a<>b \text{ and } b<>c \text{ and } a<>c \text{ and result } \equiv \text{arbitrary}) \text{ or } ((a=b \text{ and } a<>c \text{ and } b<>c \text{ and result } \equiv \text{isosceles}) \text{ or } (b=c \text{ and } b<>a \text{ and } c<>a \text{ and result } \equiv \text{isosceles}) \text{ or } (a=c \text{ and } a<>b \text{ and } c<>b \text{ and result } \equiv \text{isosceles}) \text{ or } (a=b \text{ and } b=c \text{ and result } \equiv \text{equilateral})) \text{ end } (*\text{by } \text{mk}(x, y, z).\text{isInvalid()} \implies \text{mk}(x, y, z).a=x, \text{ etc. } *)} \]
Test-Data Generation

- Recall the test specification:

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

\[
\equiv (\text{mk}(x,y,z).\text{isInvalid}() \text{ and result}=\text{invalid}) \text{ or } (\text{mk}(x,y,z).\text{isInvalid}() \text{ and } ((x<>y \text{ and } y<>z \text{ and } x<>z \text{ and result}=\text{arbitrary}) \text{ or } ((x=y \text{ and } x<>z \text{ and } y<>z \text{ and result}=\text{isosceles}) \text{ or } (y=z \text{ and } y<>x \text{ and } z<>x \text{ and result}=\text{isosceles}) \text{ or } (x=z \text{ and } x<>y \text{ and } z<>y \text{ and result}=\text{isosceles}) \text{ or } (x=y \text{ and } y=z \text{ and result}=\text{equilateral}))))
\]

(*by \text{mk}(x,y,z).\text{isInvalid}() \rightarrow \text{mk}(x,y,z).a=x, \text{ etc.} *)
Test-Data Generation

- Recall the test specification:

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

\[ \equiv \text{let inv} = 0<x \text{ and } 0<y \text{ and } 0<z \]
\[ \quad \text{and } x+y>z \text{ and } y+z>x \text{ and } z+x>y \]
\[ \text{in (not inv and result=oclUndefined) or} \]
\[ \text{(inv and } x<>y \text{ and } y<>z \text{ and } x<>z \text{ and result} \equiv \text{arbitrary}) \text{ or} \]
\[ \text{(inv and } x=y \text{ and } x<>z \text{ and } y<>z \text{ and result} \equiv \text{isosceles}) \text{ or} \]
\[ \text{(inv and } y=z \text{ and } y<>x \text{ and } z<>x \text{ and result} \equiv \text{isosceles}) \text{ or} \]
\[ \text{(inv and } x=z \text{ and } x<>y \text{ and } z<>y \text{ and result} \equiv \text{isosceles}) \text{ or} \]
\[ \text{(inv and } x=y \text{ and } y=z \text{ and result} \equiv \text{equilateral}) \text{ end} \]

(*by invariant *)
Test-Data Generation

- Recall the test specification:

  ... 

  $\equiv \text{let } \text{inv} = 0<x \text{ and } 0<y \text{ and } 0<z$

  

  

  $\text{and } x+y>z \text{ and } y+z>x \text{ and } z+x>y$

  

  $\text{in } (x<=0 \text{ and result } \equiv \text{invalid or})$

  $y<=0 \text{ and result } \equiv \text{invalid or}$

  $z<=0 \text{ and result } \equiv \text{invalid or}$

  $z=x+y \text{ and result } \equiv \text{invalid or}$

  $x=y+z \text{ and result } \equiv \text{invalid or}$

  $y=z+x \text{ and result } \equiv \text{invalid}) \text{ or}$

  $(\text{inv and } x<>y \text{ and } y<>z \text{ and } x<>z \text{ and result } \equiv \text{arbitrary}) \text{ or}$

  $(\text{inv and } x=y \text{ and } x<>z \text{ and } y<>z \text{ and result } \equiv \text{isosceles}) \text{ or}$

  $(\text{inv and } y=z \text{ and } y<>x \text{ and } z<>x \text{ and result } \equiv \text{isosceles}) \text{ or}$

  $(\text{inv and } x=z \text{ and } x<>y \text{ and } z<>y \text{ and result } \equiv \text{isosceles}) \text{ or}$

  $(\text{inv and } x=y \text{ and } y=z \text{ and result } \equiv \text{equilateral})$}

  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:

```plaintext
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 \rightarrow 3, b \rightarrow 4, c \rightarrow 5, \text{result} \rightarrow \text{arbitrary}\}
Test-Data Generation

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

- Each Conjoint in the DNF is a Test-Case

- Test Data is constructed by picking one instance of variables that makes the conjoint true!

- Test-Hypothesis applied here: **Uniformity hypothesis**
Test-Data Generation

- **Example Uniformity: Testcase**

\[ C(a,b,c,result) = \text{let } \text{inv} = 0<a \text{ and } 0<b \text{ and } 0<c \]
\[ \text{and } a+b>c \text{ and } b+c>a \text{ and } c+a>b \]
\[ \text{in } \text{inv} \text{ and } a<>b \text{ and } b<>c \text{ and } a<>c \]
\[ \text{and result } \equiv \text{arbitrary} \]

**Applied Uniformity Hypothesis:**

\( (\exists(a,b,c). \ C(a,b,c,result) \Rightarrow \text{mk}(a,b,c).\text{isTriangle}()=\text{arbitrary}) \)
\( \Rightarrow (\forall(a,b,c). \ C(a,b,c,result) \Rightarrow \text{mk}(a,b,c).\text{isTriangle}()=\text{arbitrary}) \)
Test-Data Generation

- General Scheme of **Uniformity hypothesis:**

  Formally:

  \[
  (\exists x. \text{class } x \Rightarrow \text{TestSpec(sut} , x)) \\
  \Rightarrow (\forall x. \text{class } x \Rightarrow \text{TestSpec(sut} , x))
  \]

  if there is a data in a test-class for which the system under test \textit{sut} satisfies the test specification, \textit{sut} will always satisfy it for data in this class.
Test-Data Generation

- General Scheme of **Uniformity hypothesis:**

Example: Testcase

```ml
let inv = 0<\textit{a} \text{ and } 0<\textit{b} \text{ and } 0<\textit{c} \\
\text{ and } \textit{a}+\textit{b}>\textit{c} \text{ and } \textit{b}+\textit{c}>\textit{a} \text{ and } \textit{c}+\textit{a}>\textit{b} \\
in inv \text{ and } \textit{a}\neq\textit{b} \text{ and } \textit{b}\neq\textit{c} \text{ and } \textit{a}\neq\textit{c} \\
\text{ and } result \equiv \text{arbitrary}
```

Resulting Test (satisfying this formula !):

\{a \mapsto 3, \textit{b}\mapsto 4, \textit{c}\mapsto 5, \text{result} \mapsto \text{arbitrary}\}
Test-Data Generation

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

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

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

- Surprisingly, Testing requires Theorem-Proving !!!
Test-Data Generation

- How to handle Recursion?
Test-Data Generation

- How to handle Recursion?

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

- at the level of data
Test-Data Generation

- How to handle Recursion?

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

- at the level of data

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

```text
LList
<table>
<thead>
<tr>
<th>lgth: Integer</th>
</tr>
</thead>
<tbody>
<tr>
<td>next</td>
</tr>
<tr>
<td>sum(): Integer</td>
</tr>
</tbody>
</table>
```

02/10/12
Test-Data Generation

- How to handle Recursion?

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

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

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

<table>
<thead>
<tr>
<th>LList</th>
<th>0..1</th>
</tr>
</thead>
<tbody>
<tr>
<td>lgth:</td>
<td></td>
</tr>
<tr>
<td>Integer</td>
<td></td>
</tr>
<tr>
<td>sum():</td>
<td></td>
</tr>
<tr>
<td>Integer</td>
<td></td>
</tr>
</tbody>
</table>

1

LList

lgth: Integer

sum(): Integer

next

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

- Answer:
  apply **regularity hypothesis**.

\[(\forall x. |x| < k \Rightarrow \text{TestSpec}(\text{sut}, x)) \Rightarrow (\forall x. \text{TestSpec}(\text{sut}, x))\]

if for all data up to a mesure \(k\) the system under test \(\text{sut}\) satisfies the test specification, \(\text{sut}\) will always satisfy it

Mesures are: length of lists, depth of trees, ...
Test-Data Generation

- Prerequisite: The constructor cons on lists:

  context Void::cons(r:LLlist):LLlist
  pre True
  post result.oclIsNew() and
         result.lgth = if r=null then 0 else r.next+1 endif and
         result.next = r

  context LLList:
  inv null_or_valid: self.next.isValid()
Test-Data Generation

- Consider the test specification:

\[
\text{X.sum()} \equiv Y
\]

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

(* unfold sum() *)

\[
\equiv (X=\text{null and } 0 \equiv Y \text{ or}
\]

\[
(X<>\text{null and } X.lgth+X.next.sum() \equiv Y
\]

(* logic if_then_else *)

\[
\equiv (X=\text{null and } 0 \equiv Y \text{ or}
\]

\[
(X<>\text{null and } 1+X.next.lgth+X.next.sum() \equiv Y
\]

(* invariant length, simplification *)

\[
\equiv (X=\text{null and } 0 \equiv Y \text{ or}
\]

\[
(X<>\text{null and } 1+X.next.lgth+
\]

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

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

02/10/12 (* unfold sum() *)
Test-Data Generation

- Consider the test specification:

\[ X.\text{sum}() \equiv Y \]

\[ \equiv (X=\text{null} \text{ and } 0 \equiv Y \text{ or } \]
\[ (X<>\text{null} \text{ and } X.\text{next}=\text{null} \text{ and } 1 \equiv Y \text{ or } \]
\[ (X<>\text{null} \text{ and } X.\text{next}<>\text{null} \text{ and } \]
\[ 1+X.\text{next}.\text{lgth}+X.\text{next}.\text{lgth}+X.\text{next.}\text{next.}\text{sum}() \equiv Y \]

(* logic if\_then\_else *)

\[ \equiv (X=\text{null} \text{ and } 0 \equiv Y \text{ or } \]
\[ (X<>\text{null} \text{ and } X.\text{next}=\text{null} \text{ and } 1 \equiv Y \text{ or } \]
\[ (X<>\text{null} \text{ and } X.\text{next}<>\text{null} \text{ and } X.\text{next.}\text{next}=\text{null} \text{ and } 3 \equiv Y \]
\[ (X<>\text{null} \text{ and } X.\text{next}<>\text{null} \text{ and } X.\text{next.}\text{next}<>\text{null} \text{ and } \]
\[ 3+2*X.\text{next.}\text{next}.\text{lgth}+X.\text{next.}\text{next.}\text{next.}\text{sum}() \equiv Y \]

(* same procedure as before *)
Test-Data Generation

- Consider the test specification:

\[ X.\text{sum()} \equiv Y \]

- From these test cases, we construct the test data:
  - \( X=\text{null}, Y=0 \)
  - \( X=\text{cons(null)}, Y=1 \)
  - \( X=\text{cons(cons(null))}, Y=3 \)
  - ... all other cases ...

- All other cases were represented by the regularity hypothesis, i.e ...
Test-Data Generation

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

\[(\forall X. |X|<3 \Rightarrow X.\text{sum()} \equiv Y) \Rightarrow (\forall X. X.\text{sum()} \equiv Y)\]

where we choose as “complexity mesure” just \(X.lgth\) !

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

- main tool to reduce infinite number of test cases to a finite one (which may still contain infinitely many tests!)

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

Summary

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

- Summary
  Key-Ingredients are:
  - Unfolding predicates up to a given depth $k$
  - computing the Disjunctive Normal Form ($\text{DNF}_k$)
  - Adequacy:
    Pick for each test-case (a conjoint in the $\text{DNF}_k$) one test, i.e. one substitution for the free variables satisfying the test-case!
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!
Test-Data Generation

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