Test Statistique Structurel et Fonctionnel

Pascale Thévenod-Fosse, Hélène Waeselynck
{thevenod,waeselyn}@laas.fr

Journée Club SEE
"Systèmes informatiques de confiance"
Thème : Test
Paris, le 1er juin 1999
Probabilistic Generation of Input Patterns

Random testing:
uniform distribution over the input domain
⇒ "Blind" approach

Statistical operational testing:
simulated operational profile
⇒ reliability estimation

Statistical structural / functional testing:
input distribution determined according to a test criterion
⇒ proper coverage of the software model

Focus: fault-finding
Outline

概述

- Statistical structural / functional testing: Why?
- Statistical structural / functional testing: How?
- Some results from industrial case studies
  - Procedural programs (C, Pascal, Assembler): Statistical structural and functional testing versus random testing and deterministic structural testing
  - Synchronous data flow programs (Lustre): Mixed test strategy (statistical + deterministic)
- Conclusion and On-Going Work
criterion $C \rightarrow Sc = \{ \text{elements } k \text{ to be exercised during testing} \}$

**Usual Practice**

**Imperfect Connection**

Criteria $\leftrightarrow$ Faults

**Statistical Testing According to C**

**Why?**

-deterministic test set such that each $k$ is exercised (at least) once

search for more refined criteria

compensate for the weakness of $C$ by exercising each $k$ several times
criterion C + random generation

Balanced input distribution / C → all the elements k have the same probability of being exercised

Balanced coverage of the software model

Proper input distribution / C → maximize the probability of exercising each element k

Proper coverage of the software model

Why?
Design of Statistical Test Sets

Criterion C → Sc = \{ elements k to be exercised \}

1. Search for a probability distribution over the input domain
   - Purpose: maximize the probability \( P_k \) of exercising each \( k \)

2. Assessment of the test size \( N \)
   - Stopping rule:
     - test quality \( q_N \) wrt C
       \[ N = \frac{\log(1-q_N)}{\log(1-P)} \]
     - \( P = \min \{ P_k \} \)

On average, the least likely \( k \) is exercised \( n \) times with \( n \approx -\ln(1-q_N) \):
- \( n \approx 7 \) for \( q_N = 0.999 \)
- \( n \approx 9 \) for \( q_N = 0.9999 \)

How?
Search for an Input Distribution

Analytical techniques

C = \{\text{paths}\} \rightarrow \text{Sc} = \{k_1, \; k_2\}

p_1 = \text{Prob}[0 \leq Y \leq 99]

p_2 = \text{Prob}[99 < Y \leq 9999]

\text{p_1 = p_2 = 0.5}

Structural distribution / C

Empirical techniques

successive refinements of an initial distribution
### Case Studies / French Industry

<table>
<thead>
<tr>
<th>Field</th>
<th># programs / testing level</th>
<th>Testing techniques</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Unit</td>
<td>Component</td>
</tr>
<tr>
<td>Nuclear</td>
<td>24</td>
<td>2</td>
</tr>
<tr>
<td>Avionics</td>
<td>6</td>
<td>–</td>
</tr>
<tr>
<td>Spatial</td>
<td>–</td>
<td>1</td>
</tr>
</tbody>
</table>

**Languages:** C, Pascal, Assembler, Lustre

+ **industrial test sets** (when available)
Mutation Tool: SESAME

- Sources of original functions
- List of mutations
- Mutant generator (C, Pascal, Assembler, Lustre)
- Basis of mutants
- Random data generator
- Input prob. & N
- Deterministic input files
- Execution
- Mutation analysis
- Constant operator symbol
- Output references

Results
Random Testing: a Poor Approach

FCT3: 1416 mutations, $N = 405$

% revealed

Five different test sets

(1) rapid growth
(2) stabilization

Component: 13 genuine faults, $N = 5300$

- 5 faults revealed
- 8 faults not revealed

No new fault revealed after $N = 633$
Deterministic Structural Testing

- All-Paths (Class 1): score < 100%
- Strong impact of the input values chosen to cover the criteria
- Criterion stringency → no guarantee
- Not significantly better than random testing

Results

FCT3: 1416 mutations

FCT4: 587 mutations
### Statistical Structural Testing

<table>
<thead>
<tr>
<th>Test data sets</th>
<th># Mutations not revealed</th>
<th>% Mutations revealed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deterministic structural</td>
<td>292 — 825</td>
<td>70.7% — 89.6%</td>
</tr>
<tr>
<td>Random (uniform)</td>
<td>278 — 687</td>
<td>75.6% — 90.1%</td>
</tr>
<tr>
<td>Statistical structural</td>
<td><strong>5 — 49</strong></td>
<td><strong>98.3% — 99.8%</strong></td>
</tr>
</tbody>
</table>

#### Results

- Statistical structural testing + deterministic testing of extremal values

#### Cost-effective design:
- adopt weak test criteria (e.g. branches)
- require high test quality (e.g. $q_N = 0.9999$)

4 programs FCTi $\rightarrow$ 2816 mutations
Comparison / Industrial Practice

% mutations revealed

Size of the test sets

% mutations revealed

Size of the test sets

Safety critical software (unit testing)

Results

X deterministic (industrial)

statistical structural

Random uniform
Behavior models
- Finite state machines, Decision tables (SA/RT)
- Statecharts
- etc

Hierarchical modeling (top-down)

Complexity
- weak criteria (e.g. state coverage)
- several input distributions, each one focusing on the coverage of a subset of functional hierarchy
**Component: 13 genuine faults**

**Student version:** 12 faults (A, ..., L)

**Industrial version:** 1 fault Z (to be compensated by hardware checks)

→ structural: A, G, J
→ misunderstanding: B, ..., F, I
→ initialization: H, K, L, Z

### Results

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
<th>F</th>
<th>G</th>
<th>H</th>
<th>I</th>
<th>J</th>
<th>K</th>
<th>L</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>●</td>
<td>✖</td>
<td>✖</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>✖</td>
<td>✖</td>
<td>✖</td>
<td>✖</td>
<td>✖</td>
<td>✖</td>
</tr>
</tbody>
</table>

- 1 random test set (N = 5300)
- SA/RT: 5 test sets (N = 441)
- Statecharts: 5 test sets (N = 441)

- ● always revealed
- ✖ not revealed
- i revealed by i sets out of 5
Statistical testing = criterion C + random generation

Proper input distribution, sound probe of the software model

Compensate for the weakness of C, automatic generation of large test sets

Deterministic testing = extremal/special values + selective choice

identified from the software model (structural or functional)

Boundary values

Transient modes (e.g. initialization)
Mixed Strategy for Lustre Programs

Statistical structural testing

Each transition ⇔ one cyclic execution

⇒ Test criterion: All-transitions, except for the initialization operating mode

Deterministic testing

⇒ Initial state $S_0$ and its output transition(s)
⇒ Boundary values / transition conditions

Example: $G X \leq Y$

$X = Y$ is a boundary value

Results
Principle:
- Bottom-up exploration of the program call graph
- Integration levels: simplified automata

Example: SWITCH
21 different nodes → 3 levels

- unit testing
- 1st integration testing
- 2nd integration testing
### Effectiveness of the Mixed Strategy

#### Results

- **High error detection power of statistical testing**
- **Deterministic test inputs are complementary**
- **But, at the top level, deterministic testing does not provide a sound probe of the initialization process**
  - statistical testing based on a specific test profile

<table>
<thead>
<tr>
<th>Nodes</th>
<th>Other nodes tested</th>
<th># Mutations</th>
<th>Statistical</th>
<th>Stat.+Deter.</th>
</tr>
</thead>
<tbody>
<tr>
<td>ctrl</td>
<td>gap, tempo, t_init, t_next</td>
<td>311</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>c_ok</td>
<td>ma</td>
<td>503</td>
<td>0</td>
<td>—</td>
</tr>
<tr>
<td>def_c</td>
<td>md, regul, calc</td>
<td>493</td>
<td>3–8</td>
<td>0–2</td>
</tr>
<tr>
<td>n_f</td>
<td>mc</td>
<td>506</td>
<td>0–1</td>
<td>—</td>
</tr>
<tr>
<td>s_f</td>
<td>mb</td>
<td>504</td>
<td>0–1</td>
<td>—</td>
</tr>
<tr>
<td>plc_pb</td>
<td>ns_f, me</td>
<td>317</td>
<td>0–10</td>
<td>—</td>
</tr>
<tr>
<td>switch</td>
<td>init, plc_ok</td>
<td>173</td>
<td>7–8</td>
<td>0–3</td>
</tr>
</tbody>
</table>
Conclusion

Criterion analysis for the input profile:
complexity ≅ deterministic testing
→ weak criterion + empirical search tractable

Automatic generation of test inputs

High fault revealing power

Separate testing of extremal and special values

Large test sets → oracle problem
how to determine the correctness of the outputs?
→ executable specification or prototyping facilities
Challenging Issues: OO Paradigm

• Basic modular unit: class
  ➔ only objects (class instances) can be executed ... and tested

• Structure
  ➔ distributed among several classes

• Behavior
  ➔ complex dependencies between objects
  ➔ highly dependent on the object states

1. How to define the testing levels (unit, integration, ...)?
2. Which software models & associated test criteria?

On-going work