Test Case Extraction and Test Data Generation from Design Models

Xiaojing ZHANG, Takashi HOSHINO
NTT Cyber Space Laboratories
Tokyo, JAPAN
Agenda

- Background and motivation
- Generation approach
- Tool implementation
- Evaluation results
- Future work and conclusion
Background

- We need better quality assurance!
  - Software defects become a public concern.

- Why testing?
  - The last check on the final product before release.

- What’s testing?
  - A confirmation of whether the product is developed just as one intended.

- How to improve testing?
  - Quantity
    - Test density = Number of test cases / Size of the SUT.
  - Quality
    - Structure coverage = Elements tested / Total number of elements.
    - Input space coverage = Inputs used for testing / Entire input space.
Goal in this paper

- our goal is to improve “test design”, with low cost by automation
  – for both unit test and integrated test

– Test design:
  • Extracting test cases and test data for test execution
  • Needs huge effort to achieve high density and coverage
Motivation

● Related literatures’ limitations
  ● Source code as input. Can we assume it’s 100% right?
  ● Original design notation. Not welcomed by users...
  ● Primitive data type like integer or string

● Our Motivation
  ● software design as input
    • The formalized “intentions” of the customers or designers
  ● familiar notations
    • UML 2.0 activity for behavior, class for data structure
  ● test data with structure
    • More variations have to be considered when the test data has hierarchical or repetition structure
Agenda

- Background and motivation
- Generation approach
- Tool implementation
- Evaluation results
- Future work and conclusion
Generation approach: An Overview

- UML Model
- input design model
- design model repository
- analyze design model
- extract paths
- build test cases
- generate test data
- test model repository
- output test model
- Test Case
- Test Data

2011.10.31 -- 2011.11.4
Test Case Generation

1. **UML Model**
2. **input design model**
3. **design model repository**
4. **analyze design model**
5. **extract paths**
6. **build test cases**
7. **generate test data**
8. **test model repository**
9. **output test model**
10. **testCase**
11. **TestData**
1. Decompose UML model into activities

2. Extract “paths” from UML Activity
   - simple depth first search algorithm
   - If a loop exists,
     case of not passing the loop
     case of passing the loop just once

3. Make one test case for each path
   - a test case is a pair of a particular path and the test data
4. Organize test cases as a test model

A test case is a pair of a particular path and a test data.

<table>
<thead>
<tr>
<th>Test case</th>
</tr>
</thead>
<tbody>
<tr>
<td>- path : Path</td>
</tr>
<tr>
<td>- data : Test data</td>
</tr>
</tbody>
</table>

Path
- node[]
- edge[]
- conditions[]

Test data
- Input value[]
- Expected result

Can be categorized by test level or viewpoints

Sample of Test model

TestCategory

- TestCategory
  - TestCategory
    - TestCategory
      - TestCase
      - TestCase
      - TestCase
Test Data Generation

1. UML Model
2. Input design model
3. Design model repository
4. Analyze design model
5. Extract paths
6. Build test cases
7. Generate test data
8. Test model repository
9. Output test model

Test Case
TestData
In order to achieve high "input space coverage", we thoroughly obtain the variables and the values which constitute the test data.

The generation process has 4 steps.

1. Variables
   - name: String
   - age: Integer

2. Test data specification
   - For each variable, gather constraints to trace a path.
   - UML Model:
     - Activity +
     - Class
     - path

3. Values
   - Choose values to be used, based on the test data specification.
   - [name = xyz]
   - [name = a]
   - [age = 19]
   - [age = 0]

4. Test data (combination)
   - Combine variables and values.
   - [name = xyz / age = 19]
   - [name = a / age = 0]
Generation approach: Test data (2/4)

1. Extraction of the variables
   - activity parameter node

2. Creation of the “test data specification”
   - which hold the constraints for the variables.

UML Activity

Conditions

Path

p: People
flag: Boolean

A

[p.name.length==0] [p.name.length>0]

B [p.age<20] [p.age>=20]

C D

show error message show message A

show message B

Extract

Save to

test data specification

[p.name.length>0]

[p.age>=20]

[p.age>=20]
② Generation of the “test data specification”

- Generated per each variable of each path
- Different “domain” for different data type.
- Domain of Object data type, contains specification of its own attributes: deal with hierarchy structure
③ Generation of the values
• boundary value analysis
• normal values / abnormal values
• as much as one “abnormal value” in a “leaf node” means the entire object is an abnormal value.

④ Combination of the values
• Normal Inputs
  • Minimum combination cover all normal values
• Abnormal Inputs
  • Combination contains only one abnormal value
Generation approach: Restrictions

- **Cannot handle variables** except the input parameters
  - temporary variables, global variables
    - Need more data modeling.

- **Cannot handle conditions** describing:
  - the dependence between two or more variables.
    - Need dynamic domain reduction?
  - variables overwritten or changed.
    - Need symbolic execution or simulation?

- **Inefficient combinations**
  - just use all values generated is not good enough.
    - Need pair-wise methods?
People

```java
String p.name;
Integer p.age;
```

Attribute

- `p.name.length == 0`
- `p.name.length > 0`
- `p.age < 20`
- `p.age >= 20`

Flag: Boolean

UML Activity

- **A**
- **B**
- **C**
- **D**

Branch condition

- `[p.name.length == 0]`
- `[p.name.length > 0]`
- `[p.age < 20]`
- `[p.age >= 20]`

Post condition

- `show message A`
- `show message B`
- `show error message`

Example

UML Class

- **People**
  - name: String
  - age: Integer

Attributes
For each path, extract input parameter as variables, make and initialize a test data specification depends on the variable’s type.

Object typed test data specification

String typed TDS  Integer typed TDS

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
</tr>
</thead>
</table>

Boolean typed test data specification

flag

p: People  flag: Boolean

[p.name.length==0]  [p.name.length>0]

[p.age<20]  [p.age>=20]

show massage A  show message B

show error message

①②
Update the test data specification, based on the branch condition.

Example Object typed test data specification

- **p.name**
  - **String typed TDS**
    - length lower bound = 1

- **p.age**
  - **Integer typed TDS**
    - lower bound = 20

Boolean typed test data specification

- **flag**
Generate normal values and abnormal values from the test data specification.

### Object typed test data specification

<table>
<thead>
<tr>
<th>String typed TDS</th>
<th>Integer typed TDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>age</td>
</tr>
<tr>
<td>length lower bound</td>
<td>lower bound</td>
</tr>
<tr>
<td>1</td>
<td>20</td>
</tr>
</tbody>
</table>

### Boolean typed test data specification

<table>
<thead>
<tr>
<th>flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
</tr>
<tr>
<td>false</td>
</tr>
<tr>
<td>null</td>
</tr>
</tbody>
</table>
input value is a combination of the values set to variables respectively. expected result from user defined postcondition of the path.

<table>
<thead>
<tr>
<th>ID</th>
<th>Input value (Combination)</th>
<th>Expected result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>p= name=&quot;XYZ&quot; age=20 flag=true</td>
<td>show message B</td>
</tr>
<tr>
<td>2</td>
<td>p= name=&quot;XYZ&quot; age=20 flag=false</td>
<td>show message B</td>
</tr>
<tr>
<td>3</td>
<td>p= name=&quot;&quot; age=20 flag=true</td>
<td>indefinite</td>
</tr>
<tr>
<td>4</td>
<td>p= name=&quot;XYZ&quot; age=20 flag=null</td>
<td>indefinite</td>
</tr>
</tbody>
</table>

...
Agenda

- Background and motivation
- Generation approach
- Tool implementation
- Evaluation results
- Future work and conclusion
Tool implementation

UML modelling tool (Enterprise-Architect)

**UML Model (XMI)**

Test Design Support Tool

- UI Controller
- Design Model Input
- Algorithm Manager
- Test Generation
- Test Case Output
- TestData Output

UML Meta Model

**Viewer (MS Excel)**

**Input**

**Output**

**Test cases**

<table>
<thead>
<tr>
<th>Path_Name</th>
<th>TestData_ID</th>
<th>status: OrderStatusNames</th>
<th>orderid:string</th>
<th>Expected_Status</th>
<th>Is_Normal</th>
<th>Test_Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>PathNo0</td>
<td>DataNo0</td>
<td>status:OrderStatusNames  = APPROVED</td>
<td>orderid:string = 任意文字列(999文字)</td>
<td>chagnedOrder が適切に反映されていること。</td>
<td>TRUE</td>
<td>OK / NG</td>
</tr>
<tr>
<td>PathNo0</td>
<td>DataNo1</td>
<td>status:OrderStatusNames  = SHIPPED_PART</td>
<td>orderid:string = 任意文字列(1000文字)</td>
<td>chagnedOrder が適切に反映されていること。</td>
<td>TRUE</td>
<td>OK / NG</td>
</tr>
<tr>
<td>PathNo0</td>
<td>DataNo2</td>
<td>status:OrderStatusNames  = DENIED</td>
<td>orderid:string = 任意文字列(1000文字)</td>
<td>chagnedOrder が適切に反映されていること。</td>
<td>TRUE</td>
<td>OK / NG</td>
</tr>
<tr>
<td>PathNo0</td>
<td>DataNo3</td>
<td>status:OrderStatusNames  = PENDING</td>
<td>orderid:string = 任意文字列(0文字)</td>
<td>INDEFINITE</td>
<td>FALSE</td>
<td>OK / NG</td>
</tr>
<tr>
<td>PathNo0</td>
<td>DataNo4</td>
<td>status:OrderStatusNames  = COMPLETED</td>
<td>orderid:string = 任意文字列(999文字)</td>
<td>INDEFINITE</td>
<td>FALSE</td>
<td>OK / NG</td>
</tr>
</tbody>
</table>

**Test data**

<table>
<thead>
<tr>
<th>Path_Name</th>
<th>TestData_ID</th>
<th>status:OrderStatusNames</th>
<th>orderid:string</th>
<th>Expected_Status</th>
<th>Is_Normal</th>
<th>Test_Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>PathNo0</td>
<td>DataNo0</td>
<td>status:OrderStatusNames  = APPROVED</td>
<td>orderid:string = 任意文字列(999文字)</td>
<td>chagnedOrder が適切に反映されていること。</td>
<td>TRUE</td>
<td>OK / NG</td>
</tr>
<tr>
<td>PathNo0</td>
<td>DataNo1</td>
<td>status:OrderStatusNames  = SHIPPED_PART</td>
<td>orderid:string = 任意文字列(1000文字)</td>
<td>chagnedOrder が適切に反映されていること。</td>
<td>TRUE</td>
<td>OK / NG</td>
</tr>
<tr>
<td>PathNo0</td>
<td>DataNo2</td>
<td>status:OrderStatusNames  = DENIED</td>
<td>orderid:string = 任意文字列(1000文字)</td>
<td>chagnedOrder が適切に反映されていること。</td>
<td>TRUE</td>
<td>OK / NG</td>
</tr>
<tr>
<td>PathNo0</td>
<td>DataNo3</td>
<td>status:OrderStatusNames  = PENDING</td>
<td>orderid:string = 任意文字列(0文字)</td>
<td>INDEFINITE</td>
<td>FALSE</td>
<td>OK / NG</td>
</tr>
<tr>
<td>PathNo0</td>
<td>DataNo4</td>
<td>status:OrderStatusNames  = COMPLETED</td>
<td>orderid:string = 任意文字列(999文字)</td>
<td>INDEFINITE</td>
<td>FALSE</td>
<td>OK / NG</td>
</tr>
</tbody>
</table>
Agenda

- Background and motivation
- Generation approach
- Tool implementation
- Evaluation results
- Future work and conclusion
Evaluation: Overview

- **Case study**
  - One component of a shopping store application
    - small Java web application (about 4.8KLOC)
  - Manually build an design model in UML2.0
  - Compare the artifacts made from the same model,
    - Manually derived by one average skilled developer
    - Automatically generated with our proposed method

- **Viewpoints:**
  - manpower cost
  - SUT coverage, test density

the hypothesis:
**Manpower cost**

- By automation, about 52% manpower can be reduced

<table>
<thead>
<tr>
<th></th>
<th>Manual</th>
<th>Automatic</th>
</tr>
</thead>
<tbody>
<tr>
<td>Man-power for test design (minute)</td>
<td>2330</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1123 *</td>
</tr>
<tr>
<td>Man-power for other overheads (minute)</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Unit man-power per testcase (man-minute)</td>
<td>12.3</td>
<td>3.1</td>
</tr>
<tr>
<td></td>
<td>8.1</td>
<td>3.9</td>
</tr>
<tr>
<td>Unit man-power per KLOC (man-hour)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

* The cost of retouching existing “roughly sketched” UML models to a more detailed level, enough to generate test cases. This assuming to be 50% of the overall cost of UML modeling.
**Test density and SUT coverage**

- By automation, test density increased to 1.9 times and the SUT coverage can be improved.

<table>
<thead>
<tr>
<th>SUT Coverage</th>
<th>Test density (testcase per KLOC)</th>
<th>Structure</th>
<th>Input space</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>covered all paths</td>
<td>covered representative values only</td>
</tr>
<tr>
<td></td>
<td></td>
<td>covered all paths</td>
<td>covered all boundary values</td>
</tr>
<tr>
<td></td>
<td>39.6</td>
<td>75.0</td>
<td></td>
</tr>
</tbody>
</table>
• **Number of test cases**
  
  • **Good:**
    56% of manual test cases can be generated by the proposed approach
  
  • **Bad:**
    the technique could not correspond to 44% of the manual derived test cases.
    the number of generated test data is too large to be used in test execution.
Evaluation: discussion

- **a** Matching test cases
  - Most of automatic test cases with normal inputs
  - Improvement in input space coverage
    - more variation of value
    - more exhaustive combination

- **b** Test cases created only by the expert
  - Difficult to formalize semantics of the software
  - Nested structure of activities

- **c** Test cases created only by tool
  - Most of test cases with abnormal inputs
  - The value of object data type has more variation than human usually consider
Evaluation: discussion

● What Next?

① How to deal with the knowledge not described in the UML model?

• decide a notation and describing them in the UML model.
• Or, extract from information formalized outside of the UML model.

② “many more” does not mean “better”.

• It is necessary to narrow down the number of test cases to a practicable level

— For example, for object data type, Not all combination of “leaf level” abnormal values But only the important cases
Agenda

- Background and motivation
- Generation approach
- Tool implementation
- Evaluation results
- Future work and conclusion
Conclusions

- The proposed method extract test cases, and test data with hierarchical structure, from UML activity and class models.

- Our goal was to...
  improve “test design”, with low cost by automation
  - By introducing automatic generation,
    for A PART OF the test design work, higher SUT coverage and test density can be achieved with fewer man-power.
  - Though further improvement is required.
    - Need to apply other methods to solve the restrictions.
    - Need methods to handle the outside of UML model
Future Works, Potentials, (for discussion...)

- **Vision: Toward totally automated testing**
  (Integrate “test design automation” with “test execution automation”)
  - large number of test cases will not be problem.

- **Gaps!**
  - “Free Style” Modeling is still difficult for normal developer.
  - Convert from “ordinary” design document template
  - Still need human to check the detailed test result.
    - Test oracle

- **Before we reach the utopia...**
  - Test case selection/prioritization. For manual test execution
  - Effectiveness evaluation. Mutation Testing, more large scale software

- **Never forget the development process**
  - Explain the merit of modeling and automated generation.
Xie xie!
Questions or comment?

zhang.xiaojing@lab.ntt.co.jp
● Extra slides
Position Map (target domain)

System

Integrated

Test level

Unit

Communication

Embedded

Enterprise

Domain

TorX

TGV

STG

Autolink

TEMA tool

T-Vec Rave

Simulink V&V

LEIRIOS TG

Reactis

Our Approach

AGEDIS

Our Approach

AGEDIS

2011.10.31 -- 2011.11.4

5WCSQ
③ Generation of the values

- equivalent classes division and the boundary value analysis
- extract not only the values which are inside of the domain as normal values, but also the values which are outside of the domain as abnormal values.
- For object data type, expanding the object hierarchically, if, as much as one “abnormal value” exists in a “leaf node” then the entire object is assumed to be an abnormal value.

Example: from Integer domain

```
 Integer_MIN                      Lower bound l  Upper bound u   Integer_MAX
```

blue: normal values  red: abnormal values
Combination of the values

- **Normal Inputs** consist of normal values
  - minimum combination which can use all values generated.
- **Abnormal Inputs** contains only one abnormal value
  - make the fault localization easier when failure occurs

Example

<table>
<thead>
<tr>
<th>ID</th>
<th>a</th>
<th>b</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a1</td>
<td>b1</td>
<td>c1</td>
</tr>
<tr>
<td>2</td>
<td>a2</td>
<td>b1</td>
<td>c2</td>
</tr>
<tr>
<td>3</td>
<td>a3</td>
<td>b1</td>
<td>c1</td>
</tr>
<tr>
<td>4</td>
<td>a4</td>
<td>b1</td>
<td>c1</td>
</tr>
<tr>
<td>5</td>
<td>a5</td>
<td>b1</td>
<td>c2</td>
</tr>
<tr>
<td>6</td>
<td>a3</td>
<td>b2</td>
<td>c1</td>
</tr>
<tr>
<td>7</td>
<td>a1</td>
<td>b1</td>
<td>c3</td>
</tr>
<tr>
<td>8</td>
<td>a2</td>
<td>b1</td>
<td>c4</td>
</tr>
</tbody>
</table>