Test Input Generation in A Unit Testing Process

--Part of Automated Unit Testing for Agent Systems

Zhiyong Zhang (z.zhang@student.rmit.edu.au)
John Thangarajah (john.thangarajah@rmit.edu.au)
Lin Padgham (lin.padgham@rmit.edu.au)
Test Input Generation in A Unit Testing Process

• Abstract

– We are building and refining a testing framework that can automatically test the units of an agent system based upon its Prometheus design model. When a unit is tested, the input is automatically generated according to the definition of input variables in design. Users are also allowed to add their own test input to form user-defined test cases. This talking discusses these aspects in detail.
Test Input Generation in A Unit Testing Process

• Overview
  – Background
  – Testing Units within an agent
    • What do we test
      – Identification of testable unit types
    • How do we test: Unit Testing Process
  – Test Input Generation
    • What is the test input from
    • How to generate test input
    • Value assignment for input values
    • User-defined test input
Background

• Model based testing
  – Test systems using their design models
    • e.g. OOP systems ← UML design model
    • e.g. Agent based systems ← design model of development methodology

• Prometheus and JACK
  – Our approach tests the agent systems designed using Prometheus and implemented using JACK
Units to be tested

- **event**
  
  *(percept/action/message): the component to trigger the choice and execution of a plan*

- **plan**: the component that achieve one goal

- **beliefset**: agent’s knowledge about the environment

Hierarch of agent model
Unit to be tested

-- What to test

- Testing a unit is the process of monitoring unit execution in the runtime environment to verify that it works as expected.

- Testable aspects are derived by analysing characteristics of unit types
Unit to be tested
-- testable aspects of a plan

• Trigger event
  – Does the plan get triggered by the event that it is supposed to handle?

• Context Condition
  – Is the context condition valid?

• Outgoing messages
  – Does the plan post the events/messages that it should?

• Plan competition
  – Does the plan succeed on completion?
Unit to be tested
--- testable aspects of an event

• Overlap
  – Is there more than one plan applicable for the event?

• Lack of coverage
  – Is the event handled by some plan?
Unit to be tested

-- testable aspects of an cyclic structure

• Cyclic execution path
  – Does the cyclic execution path exist at runtime?
  – Does the cyclic execution path terminate?
Unit to be tested

- Implementation of call-back methods
- Action rules
Unit Test Process

Define the fault model

- For each testable aspect, one or several fault type(s) are defined
  - e.g.

  **Testable aspect:** Does the plan post the events/messages that it should?

  **Fault type:** Some outgoing messages are never posted or sent out at runtime.
Unit Test Process

Define the fault model

- Fault levels
  - Level 3: A fault that may not be an error
    - e.g. Fault type: The context condition is always invalid. This may be caused by incomplete coverage of test input.
  - Level 2: A fault of error
    - Ex. An event triggers multiple plans
  - Level 1: An exception that is thrown
    - Potential errors in source code
Unit Testing Process

- Test driver
  - Test Agent: *sends the testing request, gather the results, generate the reports*
  - Test-driver plan: *activates the execution of the unit to be tested*
- Subsystem under test:
  - Portion of the system
  - Unit to be tested and the relevant components

*Structure for plan test:*

Diagram showing the flow of messages and requests between the Test-Agent and the Subsystem under test.
Unit Testing Process

**System Design**
Design-specific information
........
........

**Design Entity of the Unit:**
Definition of input variables

1. extract the info. about input variables

**Test Driver**
2. perform initialisation
3. generate test input
4. run test cases
   4.1 assign variable values
   4.2 run a test case
5. generate test reports

**System Implementation**

**Unit under Test**
Input Variables

• What are input variables
  – The input variables of the unit to be tested are the environmental variables that involve the unit execution
  • e.g. *Carried information* in the trigger event
Input Variables

• What are input variables
  – an input variable can be

  • Agent variable: A attribute of the agent to be tested

  • Event variable: A attribute of the trigger event of the plan to be tested, or a property of the event to be tested

  • Belief variable: A particular beliefset field
Input Variables

• How are input variables used for providing test input?

1. Specification of Value domains in design, such as the design entities of context-condition, and carried-information.  
   *E.g.*

   *event-var, stock, int, >10.0;*
   *event-var, minPrice, float, >0.0;*
   *event-var, maxPrice, float, >0.0;*
   *compare, minPrice < maxPrice*

2. Generation of the equivalence classes (ECs) of value domains, and sample values

<table>
<thead>
<tr>
<th>variable</th>
<th>index</th>
<th>domain</th>
<th>valid</th>
<th>Sample</th>
</tr>
</thead>
<tbody>
<tr>
<td>stock</td>
<td>EC-1</td>
<td>(-∞, 10)</td>
<td>No</td>
<td>-7</td>
</tr>
<tr>
<td></td>
<td>EC-2</td>
<td>10</td>
<td>No</td>
<td>10</td>
</tr>
<tr>
<td></td>
<td>EC-3</td>
<td>(10, +∞)</td>
<td>Yes</td>
<td>25</td>
</tr>
<tr>
<td>minPrice</td>
<td>EC-1</td>
<td>(-∞, 0.0)</td>
<td>No</td>
<td>-10.0</td>
</tr>
<tr>
<td></td>
<td>EC-2</td>
<td>0.0</td>
<td>No</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>EC-3</td>
<td>(0.0, +∞)</td>
<td>Yes</td>
<td>7.0</td>
</tr>
<tr>
<td>maxPrice</td>
<td>EC-1</td>
<td>(-∞, 0.0)</td>
<td>No</td>
<td>-12.0</td>
</tr>
<tr>
<td></td>
<td>EC-2</td>
<td>0.0</td>
<td>No</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td>EC-3</td>
<td>(0.0, +∞)</td>
<td>yes</td>
<td>19.0</td>
</tr>
</tbody>
</table>
Input Variables

• How are input variables used for providing test input?
  3. Generation of value combinations

<table>
<thead>
<tr>
<th>Index</th>
<th>stock</th>
<th>minPrice</th>
<th>maxPrice</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-7</td>
<td>-10</td>
<td>-12</td>
</tr>
<tr>
<td>2</td>
<td>-7</td>
<td>-10</td>
<td>0.0</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>27</td>
<td>25</td>
<td>7.0</td>
<td>19.0</td>
</tr>
</tbody>
</table>
Value Combinations

• Refine the set of value combinations
  – 1. Reduction of the set size
    • Reason: potential huge size
      – 10 variables, each of which has 3 value domains:
        » $3^{10} = 59049$ combinations
    • Solution: n-wise technique
      – This approach generates a new set of value combinations that cover all n-wise ($n \leq 2$) interactions among the input variables and their values in order to reduce the size of the input data set. (We use $n = 2$)
      – The CTS (Combinational Testing Service) library provides the implementation API. (http://www.alphaworks.ibm.com/tech/cts)
Value Combinations

• Refine the set of value combinations
  – 2. Cover all comparison relationships
    • E.g. compare, \( \text{minPrice} < \text{maxPrice} \)
    • All value combinations are reviewed to verify if all comparison relationships are covered. If not, new combinations will be added.
      – \( \text{minPrice} < \text{maxPrice} \) (valid relationship)
      – \( \text{minPrice} > \text{maxPrice} \) (invalid relationship)
      – \( \text{minPrice} = \text{maxPrice} \) (invalid relationship)
Value assignment

• One value combination provide the input of one test case
• The values of are assigned to related implementation variables
• How is an input variable value assigned depends on how it is implemented.
Value assignment

-- single variables

• An input variable is a *single variable* if it is implemented as:
  – a public variable to which its values can be assigned, or
  – a private variable which can be assigned values by invoking a public implemented function
  – Developers specify these value assignment relationships in the *unit test descriptor* in design.
Value assignment

- Example
  -- single variables

```
Code of the trigger event:
public event TriggerQuery extends Event {
    public String bookID;
    private String bookName:
    public String getBookName() {
        return this.bookName;
    }
    public void setBookName(String val) {
        this.bookName = val;
    }
}
```

```
Code of the plan to be tested:
public plan QueryBookbyName extends Plan {
    # handles event TriggerQuery inKV:
    context() {
        inKV.getBookName().check(!null
        if (inKV.bookID>0) {
            ...
        }
    }
}
```
Value assignment

--- complex variables

- An input variable is a *complex variable* if it is implemented as:
  - a public *nested* variable to which its values can be assigned, or
  - a private variable which can be assigned values using a public *nested* function
Value assignment

• Example

-- complex variables

System Design Document

Plan: Query Book by Name

Definition of Input Variables:

- event-att: Attendee, string, !-null,
- event-att: AttendeeName, string, !-null,

Unit Test Descriptor:

1. Value assignment for input variables
   (Input variables are mapped to implemented variables or functions for value assignment)

<table>
<thead>
<tr>
<th>Inputvariable</th>
<th>type</th>
<th>Value assignment in implementation</th>
</tr>
</thead>
<tbody>
<tr>
<td>AttendeeEmail</td>
<td>complex</td>
<td>att_email</td>
</tr>
<tr>
<td>AttendeeName</td>
<td>complex</td>
<td>and setName(String)</td>
</tr>
</tbody>
</table>

2. Operations for initialization

System Implementation

Code of the trigger event:

```java
public event TriggerQuery extends Event {
    public Attendee att;
}
```

```java
public class Attendee{
    public String email;
    private String name;
    public String getName(){
        return name;
    }
    public void setName(String val){
        name = value;
    }
}
```

Code of the plan to be tested:

```java
public plan QueryBookByName extends Plan {
    # handles event TriggerQuery inEv:
    context();
    inEv, getBookNameName() /=null
    id inEv.BOOK_ID>0 ;
}
```
Value assignment

• An input variable is a *belief variable* if it denotes a beliefset field, which is considered to be *sensible* to the internal logic of the unit to be tested
  
  – E.g.

  belief-var, Beliefset1, FieldValue11, int, >0;
  belief-var, Beliefset1, FieldValue12, int, >0;
  belief-var, Beliefset2, FieldValue21, int, >0;
Value assignment

- belief variables

In a value combination, the values of the belief variables that are related to the same beliefset are considered together for value assignment.

<table>
<thead>
<tr>
<th>Value combinations:</th>
</tr>
</thead>
<tbody>
<tr>
<td>Index</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>.......</td>
</tr>
<tr>
<td>12</td>
</tr>
</tbody>
</table>

Beliefset1:

<table>
<thead>
<tr>
<th>FieldValue11</th>
<th>Fieldvalue12</th>
<th>Fieldvalue13</th>
<th>Fieldvalue14</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>20</td>
<td>random</td>
<td>random</td>
</tr>
</tbody>
</table>

Beliefset2:

<table>
<thead>
<tr>
<th>FieldValue21</th>
<th>Fieldvalue22</th>
<th>Fieldvalue23</th>
<th>Fieldvalue24</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>random</td>
<td>random</td>
<td>random</td>
</tr>
</tbody>
</table>
User defined test input values

- We provide GUI to allow user-defined test input values

---

-- GUI for value editing
User defined test input values

• Users are allowed to initialise beliefsets for testing purpose (on progress)
  – specifies particular records in particular beliefsets for a test case
  – writes initialisation functions in implementation, and specify them in the unit test descriptor, which will then be executed by the test driver
After the test input generation

**System Design**
Design-specific information

**Design Entity of the Unit:**
Definition of input variables

1. extract the info. about input variables

**Test Driver**

2. perform initialisation
3. generate test input
4. run test cases
4.1 assign variable values
**4.2. run a test case**
5. generate test reports

**System Implementation**

**Unit under Test**
Conclusion