An Aspect-Oriented Testability Framework

Nankai Pan and Eunjee Song

Department of Computer Science
Baylor University
October 26, 2012

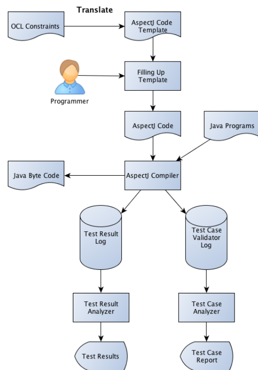
Motivation

- OCL cannot be explicitly used as testing since it is not executable.
- Translating OCL constraints into code automatically is hard.
- Run time constraint checking can help but it has a crosscutting concern.
- Making a software testable is hard.
- Testability of a software can be largely improved by improving observability.
- Improving software observability is also a crosscutting concern.

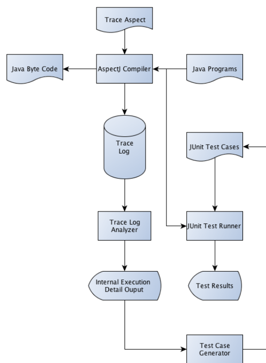
Framework Overview

- Run-time constraint checking subsystem
  - Translating OCL constraints into AspectJ code.
  - Using OCL constraints to filter testing data.
  - Using OCL constraints to determine test results.
- Execution trace logging subsystem
  - A generic trace aspect
- Fault locator logging subsystem
  - Logging aspect

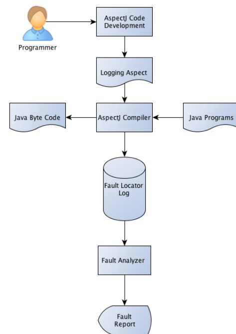
Framework Overview (cont.)



(a) Run-time Constraint Checking



(b) Execution Trace Logging



(c) Fault Locator Logging

AOP-based Run-time Constraint Checking Subsystem

- Input OCL constraints are for testing purpose only. Filter them by keeping these constructs:
  - Preconditions
  - Postconditions
  - Invariant
- Filter test data
  - Detected by OCL precondition violation.
  - A test case is inappropriate for testing a method if it doesn't satisfy the precondition of the method.
- Determine test results
  - Detected by OCL postcondition and invariant violation.

Constraint Checking Aspect

- Two approaches
  - A single big checking aspect.
  - Each class have its own checking aspect.
- Checking aspects have to be *privileged* aspects.
- Three kinds of checking aspects
  - Preconditions checking aspect
  - Postconditions checking aspect
  - Invariant checking aspect

Checking Preconditions

- Template for checking preconditions of a non-static method

```
// Precondition checking for non−static method
before(ClassName self [, MethodParameters]) :
  execution(return_type ClassName.MethodName()[MethodParameterTypes]])
  && target(self) [&& args(ParameterNames)]
  && within(ClassName) {
  // Check the precondition
  ...
}
```

- Template for checking preconditions of a constructor

```
before(ClassName self [, MethodParameters]) :
        execution(ClassName.new([ParameterTypes]))
        && target(self) [&& args(ParameterNames)]
        && within(ClassName) {
        // Check the precondition
        ...
}
```

Checking Postconditions

- Template for checking postconditions of a non-static method

```
MethodRetrunType around(ClassName self [,
    MethodParameters]) :
        execution(MethodRetrunType ClassName.MethodName
            ([MethodParameterTypes]))
        && target(self)
        && args(ParameterNames) {
        // Create any necesary @pre variables
        ...
        // Used when keyword "result" is used in OCL
            postcondition
        [MethodRetrunType result;]
        // Proceed
        [result =] proceed(self [, MethodParameters]);
        ...
        // Check the postcondition
}
```

Checking Invariants

- Advice code template for checking invariants

```
void ClassName.invariant() {
        [super.invariant();]
    ...
    // Check the invariant
}

before(ClassName self) :
        execution(public * ClassName.*(..))
        && target(self)
        && within(ClassName)
        && !within(ConstraintCheckingAspect) {
        self.invariant();
}

after(ClassName self) :
        execution(public * ClassName.*(..))
        && target(self)
        && within(ClassName)
        && !within(ConstraintCheckingAspect) {
        self.invariant();
}
```

Execution Trace Logging Subsystem

- Problems of constraint checking
  - OCL constraints may not be accurate and complete.
  - Some software system even does have OCL constraints defined.
- Components:
  - Trace aspect
  - Trace log
  - Trace log analyzer

Using Trace Aspect to Improve Software Observability

- Goal
  - Help programmers to observe the internal and external execution details of a software artifact.
  - Keep original program intact.
  - Logging module should be plug-and-playable
- Execution details can be displayed in three levels:
  - Unit level (L3)
  - Integration level (L2)
  - System level (L1)

Trace Aspect

```
privileged aspect TraceAspect {
    pointcut traceL2(): (execution(* *.*(..)) || execution(*.new(..)) && !within
         (TraceAspect));

    pointcut traceNL2(): !within(TraceAspect) && !execution(* *.*(..)) && !
         execution(*.new(..));

    pointcut traceGetSetL3(): (get(* *.*) || set(* *.*)) && !within(TraceAspect)
         ;

    pointcut traceCF(): cflow(traceL2()) && !within(TraceAspect);
        :
    // Checking preconditions
    before(): traceL2() {
        printSystemLevel(thisJoinPointStaticPart);
        printIntegrationLevel(thisJoinPointStaticPart);
        printUnitLevel(thisJoinPoint);
        System.out.println(thisEnclosingJoinPointStaticPart.getSourceLocation())
             ;
    }

    // Checking post-conditions
    after(): traceL2() {
        printSystemLevel(thisJoinPointStaticPart);
        printIntegrationLevel(thisJoinPointStaticPart);
        printUnitLevel(thisJoinPoint);
        System.out.println(thisEnclosingJoinPointStaticPart.getSourceLocation())
             ;
    }

    befeore(): traceNL2() {   }
    after(): traceNL2() {   }
        :
}
```

Unit Level

- What we can observe:
  - The input and output of methods
  - State of objects before and after methods execution
  - The arguments passed with the object
  - The behavior of loop join points such as if statement or loop statements
- The method defined in Trace Aspect for unit level logging

```
private void traceUnit(JoinPoint jp) {
  System.out.println(jp);
  StringBuffer argst = new StringBuffer("ARGS:\t");
  Object [] args = jp.getArgs();
  for(int length = args.length, i = 0; i < length; i++)
        {
    argst.append("[" + i + "] =" + args[i]);
  }
  System.out.println(argst);
}
```

Internal Execution Details of Stack Class at Unit Level

Stack.java

```
1  import java.io.IOException;
2
3  public class Stack {
4     private int maxStack;
5     private int emptyStack;
6     private int top;
7     private char[] items;
8     public Stack(int size) {
9        maxStack= size;
10       emptyStack = -1;
11       top = emptyStack;
12       items = new char[maxStack];
13    }
14    public void push(char c) {
15       items[++top] = c;
16    }
17
18    public char pop() {
19       return items[top--];
20    }
21
22    public boolean full()  {
23       return top + 1 == maxStack;
24    }
25
26    public boolean empty()  {
27       return top == emptyStack;
28    }
29    public static void main(String[] args) throws IOException {
30          Stack s = new Stack(10);
            char ch;
```

Javadoc | Declaration | Console | Search

```
<terminated> Stack [Java Application] /System/Library/Java/JavaVirtualMachines/1.6.0.jdk/Contents/Home
Stack.java:18
d     L3/ execution(boolean Stack.empty())
L3/ARGS:
L3/THIS: Stack@2e8f4fb3  L3/TARGET: Stack@2e8f4fb3
Stack.java:26
    L3/ execution(char Stack.pop())
L3/ARGS:
L3/THIS: Stack@2e8f4fb3  L3/TARGET: Stack@2e8f4fb3
Stack.java:18
c     L3/ execution(boolean Stack.empty())
L3/ARGS:
L3/THIS: Stack@2e8f4fb3  L3/TARGET: Stack@2e8f4fb3
Stack.java:26
    L3/ execution(char Stack.pop())
L3/ARGS:
L3/THIS: Stack@2e8f4fb3  L3/TARGET: Stack@2e8f4fb3
Stack.java:18
b     L3/ execution(boolean Stack.empty())
```

Internal Execution Details of Stack Class at Unit Level

Class name	Level No./Method	Line Number	Argument
Stack	L3/Stack.full()	26	L3/ARGS:
Stack	L3/Stack.push(char)	18	L3/ARGS: [0] =a
Stack	L3/Stack.full()	26	L3/ARGS:
Stack	L3/Stack.push(char)	18	L3/ARGS: [0] =b
Stack	L3/Stack.full()	26	L3/ARGS:
Stack	L3/Stack.push(char)	18	L3/ARGS: [0] =c
Stack	L3/Stack.full()	26	L3/ARGS:
Stack	L3/Stack.push(char)	18	L3/ARGS: [0] =d
Stack	L3/Stack.full()	26	L3/ARGS:
Stack	L3/Stack.push(char)	18	L3/ARGS: [0] =e
Stack	L3/Stack.full()	26	L3/ARGS:
Stack	L3/Stack.empty()	30	L3/ARGS:
Stack	L3/Stack.pop()	22	L3/ARGS:
Stack	L3/Stack.empty()	30	L3/ARGS:
Stack	L3/Stack.pop()	22	L3/ARGS:
Stack	L3/Stack.empty()	30	L3/ARGS:

Fault Locator Logging Subsystem

- Logging information can be used to debug code and locate fault.
- An example logging aspect

```
public class LoggingAspect {
        ......;

        Pointcut methCall(INT_Stack s): target(s) && execution(void
             INT_Stack.*(..));
        INT_Stack.*(..)
        ......;

        before(INT_Stack s) : methCall(s) {
                System.out.println("Enter into: " + thisJoinPoint.
                     getSignature());
                System.out.println("Stack's size before execution: " + s.
                     getSize());
        }

        after(INT_Stack s) returning: methCall(s) {
                System.out.println("Leaving: " + thisJoinPoint.
                     getSignature());
                System.out.println("Stack's size after execution: " + s.
                     getSize());
        }
        ......;
}
```

Using Logging Aspect to Locate Fault (Cont.)

- Testing a Stack class.
- Test case: tc = push(2), pop(), push(4), setEmpty()
- Logging info:

```
Enter into: void INT_Stack.push(int)
Stacks size before execution: 0
Leaving: void INT_Stack.push(int)
Stacks size after execution: 1
Enter into: void INT_Stack.pop()
Stacks size before execution: 1
Leaving: void INT_Stack.pop()
Stacks size after execution: 0
Enter into: void INT_Stack.push(int)
Stacks size before execution: 0
Leaving: void INT_Stack.push(int)
Stacks size after execution: 1
Enter into: void INT_Stack.setEmpty()
Stacks size before execution: 1
Leaving: void INT_Stack.setEmpty()
Stacks size after execution: 1
```

Conclusion and Future work

- It is a framework to facilitate software testing without affecting the original programs.
- Three components:
  - Run-time constraint checking subsystem
  - Execution trace logging subsystem
  - Fault locator logging subsystem
- Future work
  - Conduct a case study to apply the framework to some real-world programs.
  - Extend the framework to improve controllability of the software artifact by using the similar AOP-based approach.