Dependable Software Systems

Testing the Protocols of Classes in Object-Oriented Programs

Material drawn from [Kirani, Tsai, 1994]
Method Sequence Specification

• An object accessing the services of other objects must know the proper interface protocol of the objects.

• The *Method Sequence Specification* technique documents the correct order in which methods can be invoked.

• *Regular Expressions* are used to represent method sequence specifications.
Correct Object Behavior

- The result of any method execution depends on previously executed methods.
- Thus, correct object behavior is possible only if methods are invoked in a well-defined sequence.
- The *method sequence specification* for classes is important as it specifies the correct sequences in which objects must receive messages.
Stack Example

• The objects of a Stack class should receive a `pop()` message only after receiving a `push()` message.

• Some ordering rules are due to implementation issues:
  – E.g., objects must receive a `constructor` message before receiving any other message.
Method Sequence

- **Methods(C)** is the set of all public methods defined in class **C**.
  - For a **Stack** class the set of methods can be: \( \text{Methods(Stack)} = \{\text{push, pop, isempty, isfull, top}\} \)

- **A method sequence** \( S \) of **C** is a finite sequence of methods **M** of **C**, \( (M_0. M_1. \ldots . M_n) \), where \( M_i \ (0 \leq i \leq N) \) are in set **M**.
  - E.g., for the **Stack** class, a method sequence is: \( \text{(push.top.pop isempty)} \)

Dependable Software Systems (OO Protocol Testing) © SERG
Regular Definition of Method Sequence Specifications

• Regular expressions are used to represent Method Sequence Specifications for a class \( C \) over the alphabet \( (\Sigma) \) consisting of methods from \( Methods(C) \).
Example of a Regular Definition

\[ l_1 \rightarrow r_1 \]
\[ l_2 \rightarrow r_2 \]
...
\[ l_n \rightarrow r_n \]

where each \( l_i \) is a distinct label, and each \( r_i \) is a regular expression over \( \text{Methods}(C) \cup \{l_i, l_i, \ldots, l_n\} \).

The regular expression can use "*/"+ for specifying zero/one or more occurrences of a symbol.
Specification of a Simple Bank Account Class

• Interface methods of the Account class:
  – \( \Sigma = \{\text{Create,Deposit,Open,Withdraw,Close,Delete}\} \)
  – The regular expression of the method sequence specification for class Account is:
    
    \[ \text{Create.Open.}(\text{Deposit.}(\text{Deposit} \mid \text{Withdraw})^*)\text{Close.Delete} \]

• The regular expression is equivalent to the set:
    \( (\text{Create.Open.Deposit.Deposit.Withdraw.Close.Delete}), \ldots \} \)
Applications of Method Sequence Specification

- Ensuring the conformance of the implementation of a class with the corresponding method sequence specification.
- Constructing a run-time verification system for a class to ensure the correct method invocation sequence for each object of the class.
- Generating test cases for testing the implementation of a class.
Verification of Implementation

- The method sequence specification of a class specifies the correct sequence usage for all objects of the class.
- But the implementation of a class may contain method invocation sequences that are inconsistent with the method sequence specification.
- It is important to identify and correct all such inconsistencies.
Verification of Implementation (Cont’d)

- We can use manual inspection or static analysis of the implementation to identify inconsistencies.
- Note that pointers, polymorphism, and dynamic binding in OO programs complicates the control-flow analysis.
  - For dynamic binding of methods to messages, the static analysis must consider all possible methods that can be bound to a message.
Verification of Implementation
(Example)

- (Open.Deposit.Close.Withdraw)
- This is an invalid sequence!
- Each such invalid method sequence may be a potential fault.
Run-time Verification System

• What if all the invalid method sequences are not identified using inspection and static analysis?
• For safety-critical systems it may be necessary to identify all invalid method sequences during execution.
• The run-time verification system helps identify and recover from invalid method invocations.
Run-time Verification System (Cont’d)

• Each object maintains an access pointer to the method sequence specification of its corresponding class.

• For each method invocation, a check is made for sequence consistency (if the sequence is in the regular language) with respect to the stored sequence specification.

• If the method invoked is not in correct sequence, an exception is raised and handled.
## Implementing a Run-time Verification System

### Account Class

#### Method Dictionary

<table>
<thead>
<tr>
<th>Name</th>
<th>Executable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deposit</td>
<td>-</td>
</tr>
<tr>
<td>Withdraw</td>
<td>-</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

#### Sequence Specification

Open.Deposit(...)
Efficient Algorithm for Consistency Checking

- We must be able to verify whether one regular expression is the same as another.
- Each regular expression defines a finite automata (FA).
- An FA accepts a set of strings and all of the acceptable strings form a language L.
- If L1 and L2 are the languages represented by two regular expressions, then the two regular expressions are equal if L1=L2.
Algorithm for Consistency Checking (Cont’d)

- One can verify that $L_1 = L_2$ by constructing a finite automaton, $D_1$, that satisfies $L_1 \oplus L_2$.
  - $\oplus$ is the exclusive-or operator.
- If $D_1$ only accepts the empty string, then $L_1 = L_2$. 
Test Case Generation from Sequence Specification

• In an OO program, testing a class corresponds to testing the methods supported by the class.
  – **Step 1**: Individual methods must be tested using the white-box techniques we discussed.
  – **Step 2**: Method interactions (protocols) must be tested for correctness. We must define coverage criteria for this ...
Test Case Generation from Sequence Specification (Cont’d)

- **All-node coverage criterion:**

- **All-edge coverage criterion:**
Test Case Generation from Sequence Specification (Cont’d)

• For each method in the sequence, data input of each parameter and the expected output must be determined.

• If a method invokes methods of other objects, then during testing test stubs can be used to supply return values.
From STDs to REs

- STDs are often constructed for each class during the design phase, these STDs can be used for generating the method sequence RE.

- The method sequence specification for a class can be derived automatically from the STD of the class.
  - Just use the events and ignore the states.
  - There exist several efficient algorithms for converting STDs to REs [Hopcroft, Ullman 79].
From STDs to REs (Example)

Start

Open() → Empty Account → Operational Account

Deposit() → Operational Account

Withdraw() → Operational Account

Deposit() → Operational Account

Close() → Null Account

(Open.Deposit.(Deposit|Withdraw)*.Close)
Inheritance and Method Sequence Specification

• Child classes inherit the method sequence specification from their parent class.
• We next present rules to verify consistency between the method sequence specifications of the parent and child classes for three kinds of inheritance:
  – Specialization inheritance
  – Refinement inheritance
  – Implementation inheritance
Specialization Inheritance

- The child class can enhance itself with new methods, in addition to all the inherited methods from the parent class.
- The inherited methods are not changed (overridden).
Specialization Inheritance (Cont’d)

- Let $RE(C1)$ and $RE(C2)$ be two regular expressions defining the method sequence specifications for classes $C1$ and $C2$, where $C2$ inherits from $C1$.
- Let $S$ be the new methods of $C2$.
- Let $NRE = e$-instantiates($RE(C2), S$).
  - Replace symbols of $S$ in $RE(C2)$ with epsilon.
  - E.g., if $RE=M0.M1.(M2|M3)^*$ and $S=\{M1,M3\}$, then $e$-instantiates($RE, S$) = $M0.(M2)^*$
- $C2$ is consistent with $C1$ iff $\text{Lang } (RE(C1)) = \text{Lang}(NRE)$
Specialization Inheritance (Example)

• Let CheckingAcc be a class that specializes class Account.

• CheckingAcc has 2 new methods:
  – $S = \{\text{Balance, Report}\}$

• The method sequence specifications of Account and CheckingAcc are:
  
  Account $\rightarrow (Deposit.(Deposit | Withdraw)^*)$

  CheckingAcc $\rightarrow (Deposit.(Deposit | Withdraw | Balance | Report)^*)$
CheckingAcc and Account are Consistent

• NRE
  = e-instantiates(RE(CheckingAcc),S)
  = (Deposit.(Deposit|Withdraw) *)
  = Lang(RE(Account))
Refinement Inheritance

• The child class modifies the semantics of some of the inherited methods from the parent class.
• The modification in the inherited methods can be either in the:
  – method signature
  – method behavior (overriding)
Refinement Inheritance (Cont’d)

• Let $RE(C1)$ and $RE(C2)$ be two regular expressions defining the method sequence specifications for classes $C1$ and $C2$, where $C2$ inherits from $C1$.

• $S = Ref \cup N$
  – $Ref$ = refined methods in $C2$
  – $N$ = new methods in $C2$

• $NRE = e\text{-instantiates}(RE(C2), S)$

• $C1w/oRef = e\text{-instantiates}(RE(C1), Ref)$. 

• $C2$ is consistent with $C1$ iff $\text{Lang}(RE(C1w/oRef)) = \text{Lang}(NRE)$
Refinement Inheritance (Example)

• Let $\text{CheckingAcc}$ be a class that specializes class $\text{Account}$.

• $\text{CheckingAcc}$ has 2 new and 1 refined methods:
  – $N = \{\text{Balance, Report}\}$
  – $\text{Ref} = \{\text{Withdraw}\}$
  – $S = \{\text{Balance, Report, Withdraw}\}$

• The method sequence specifications of $\text{Account}$ and $\text{CheckingAcc}$ are:
  $\text{Account} \rightarrow (\text{Deposit}.(\text{Deposit} \mid \text{Withdraw})^*)$
  $\text{CheckingAcc} \rightarrow (\text{Deposit}.(\text{Deposit} \mid \text{Withdraw} \mid \text{Balance} \mid \text{report})^*)$
CheckingAcc and Account are Consistent

- \( NRE \)
  \( = e\text{-instantiates}(RE(\text{CheckingAcc}), S) \)
  \( = (\text{Deposit.(Deposit)}\,*) \)
  \( = \text{Lang}(RE(\text{Accountw/oRef})) \)
Implementation Inheritance

• Using inheritance for reuse by excluding some of the methods of a previously defined class.

• Some methods of the parent class are excluded in the child class public interface.
Implementation Inheritance (Cont’d)

• The rule that checks the consistency between parent and child class is similar to the one described in specialization inheritance.

• The only difference is some of the parent methods are not inherited and, therefore, these methods must not be considered in the parent class regular expressions when applying the consistency rule.
References