UML-based Test Generation and Execution

Axel Ruder
Email: axel.ruder@siemens.com
+1 (609) 734 3632

Software Engineering Department
Siemens Corporate Research
Princeton, NJ

Outline

- Research at SCR
- Model based testing
  - Category/Partition Method
  - TDE/UML for Unit/Integration Testing
  - TDE/UML for System Testing
- Benefits
- Summary
Motivation

- Siemens is one of the largest software development houses in the world (>30,000 developers worldwide).
- Siemens software ranges from embedded to enterprise level software systems.
- These complex, distributed systems need to be modeled using some standardized notation => UML.
- For embedded systems, for example telecom systems and reactive components, we need to model their behavior based on statecharts and sequence diagrams.
- For enterprise level applications, for example hospital IT systems and web-based applications, we need to model their behavior based on Use Cases and Activity diagrams.
- With this presentation, I'll be providing you an overview of our techniques, tools and their applications.
Testing Research at SCR

- Focus on developing efficient techniques and tools for test case generation
- Coupling them with suitable execution tools
- Specification-based black-box tests
- Testing based on UML diagrams
- Support of all test levels:
  - Unit testing TDE/UML 1999
  - Integration testing TDE/UML 2000
  - System testing TDE/UML 2003

The Category-Partition Method

- The method identifies behavioral equivalence classes within the structure of a SUT
- A category or partition is defined by specifying all possible data choices that it can represent.
  - Choices: data values, references to other categories or partitions, or a combination of both.
- A test design is created using TSL - Test Specification Language

Example:
Test Bank_Account {
  frame value expr
  "Soperation Saccount_number Samount
  Soperation Saccount_number Samount
  Soperation Saccount_number Samount"
  partition amount: {
    * zero value 0;  
    * small value select (1..100); 
    * high value select (101..10000); 
  }
  partition account_number: {
    * individual value "01-123456";
    * joint value "02-222567";
    * corporate value "11-987654";
  }
  partition operation: {
    * create_account
    * deposit [pre (operation = create_account);]
    * withdraw [pre (operation = create_account);]
  }
} ..

Test Generation and Execution

Target:
- System level functional testing
  - Diagram used: Use cases/Activity Diagrams

Target:
- Software Components Unit / Integration testing
  - Diagram used: Statechart(s)/Interaction Diagrams

Diagram used:
- TDE, UML

Approach

Step 1: Modeling Component Behavior

- UML Statecharts describing behavior of each component
  - Representing the normal and erroneous behavior of the component
  - UML extended by a new transition labeling scheme

- Assumptions:
  - Point-to-point communication semantics between components rather than a shared event model
  - Communications are synchronous (blocking)
  - Implementation is deterministic

Tools:
- UML Case Tool (e.g., Rational Rose)
- Execution Tools
- Test Scripts
TDE/UML

An Example: A Communication Protocol

- Focus on integration testing
- Generate test cases to validate component interaction
- Consider subsystem A
- External interfaces:
  - tuser
  - timer
  - txport
- Internal interfaces:
  - timing

Describing Component Behavior and Interaction

Matching Send and Receive Events Between Components
Step 2: Generating Test Cases

(1) Normalizing the UML-based Models
   - Resolving transitions with multiple events

(2) Computation of a global behavioral model based on matching send and receive events
   - Composing the Global Behavioral Model (complexity is better than exponential)
   - Determining a composition order (based on subsystem definition)

(3) Generating test cases from the global behavioral model
   - Default coverage criterion: all transitions (within/between components)
   - Using the data variations specified during the initial modeling phase
Composing a Global Behavioral Model...

Test Case Generation

Test Cases for Subcomponent:

For integration testing: all transitions between components are covered.
Test Generation and Execution

Target: System level functional testing
Diagram used: Use cases/Activity Diagrams

Target: Software Components Unit / Integration testing
Diagram used: Statechart(s)/Interaction Diagrams

Execution Tools

Test Scripts

UML Case Tool (e.g., Rational Rose)

TDE/UML for System Testing

System Specification
- Use Case Specification
- Flow of Events
- Business Rules

UML Diagrams

TDE

Test Scripts

Capture/Replay Tool

Application GUI

SUT

(TSL) Annotations

Test Designer
TDE/UML for System Testing

- Why "Use Cases" and "Activity" Diagram?
  - They offer systematic and intuitive means of capturing functional requirements with a focus on value added to the user
  - They describe system behavior from a user point of view
  - Unfortunately not formal enough for test automation

- Describing Use Case with Activity Diagram
  - Annotations
    - UML stereotypes and notes
    - Activities organized in swim lanes
      - "Happy path" (ActorAction, SystemResponse) and alternative courses
  - Refinements
    - Activity and variables - Strategy: be as abstract or concrete as you like

Modeling Use Cases with Activity Diagrams

<<Name>>
CancelEncounter
{AbstractLevel}
InvokeCancellation
<<UserAction>>
Fill out Cancellation Form
<<SystemResponse>>
PresentEncounterInformation
<<precondition>>
1. A valid user is logged into the system.
2. An encounter in a "Attended" or "Checked-In" status has been identified
3. Context of the encounter is known

PresentCancellationForm
<<SystemResponse>>
PresentEncounterInformation
<<SystemResponse>>
PrintArtifacts
<<include>>
Registrar
UpdateEncounter
CancelEncounter
<<include>>
Annotations to an Activity Diagram

- **Structural**
  - Activity: `<ActorAction>`, `<SystemResponse>` or `<Include>`
  - Variables: `<define>`
  - Branches: TSL guard condition (using variables)

- **Test Related**
  - Partitions
  - Coverage statements

```
<define>
ExistingServicesProfileOption
ServicesAttached
</define>

partition ExistingServicesProfileOption {
* Warning value "warning";
* Error value "error";
}
```

```
PresentEncounterInformation
<<SystemResponse>>
PresentCancellationForm
<<UserAction>>
FilloutCancellationForm
```

```
[ ServicesAttached && (ExistingServicesProfileOption==#Error) ]
```

```
InvokeCancellation
<<UserAction>>
```

```
PresentEncounterNotEligibleMessage
<<SystemResponse>>
```

```
ExistingServicesProfileOption
```
Annotations to an Activity Diagram

- Structural:
  - Activity: <ActorAction>, <SystemResponse> or <Include>
  - Variables: <define>
  - Branches: TSL guard condition (using variables)

- Test Related:
  - Partitions
  - Coverage statements

Activities may be refined using subdiagrams
Transparent abstraction levels for test generation
Be as abstract as you want
Test Case Generation from within Rational Rose

Figure 5: Generating System Tests from within Rational Rose

Test Generation

- Related to a specific Use Case
- Steps
  - Refinements are expanded
  - Happy paths from included Use Cases are analyzed
  - TSL test design is generated from the activity diagram
    - Mapping activities and transitions to TSL partitions and choices.
  - TDE creates test cases in order to satisfy the coverage requirements
    - Default - all data variations and branches are covered

<table>
<thead>
<tr>
<th>Steps</th>
<th>Preconditions</th>
<th>Postconditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>InsertSale</td>
<td></td>
<td></td>
</tr>
<tr>
<td>UpdateSale</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DeleteSale</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Purchase</td>
<td></td>
<td>Purchase</td>
</tr>
<tr>
<td>CreditCheck</td>
<td></td>
<td>CreditCheck</td>
</tr>
<tr>
<td>CustomerProof</td>
<td></td>
<td>CustomerProof</td>
</tr>
</tbody>
</table>
Test Script Output

```xml
<Test name="CancelEncounter_SystemTest_1">
  <UserAction name="FilloutCancellationForm">
    <Choice name="InvalidDate" value="T" />
    <Choice name="FilloutCancellationForm_cancel" value="F" />
  </UserAction>
  <SystemResponse name="ValidateEntries" />
  <SystemResponse name="PresentInvalidMessage" />
</Test>
```

Benefits

- **Modeling system behavior.**
  - Results in better, consistent and complete system models
  - Enables test designer to identify and compactly document a greater variety of test scenarios
  - Effectiveness of test design phase can be increased

- **Generating test procedures.**
  - Supports the automatic and systematic derivation of test cases
  - Notion of test adequacy or coverage with respect to the system functionality is addressed

- **Executing test scripts.**
  - Support for automated test execution
  - Promotes script reuse and simplifies script maintenance through ‘test snippets’
Conclusion and Future Work

Conclusion
- Presented ongoing research project at SCR
- UML-based approach on unit/integration and system testing
- Benefits from both combining COTS and in-house tools

Future Work
- Empirical Study
- Tool Enhancements (Wizard, Advisor …)
- Integration of
  - UML 2.0 Testing Profile
  - TTCN-3
  - Product Lines