Scenario-driven Testing of Security-related Domain-specific Language Models

Bernhard Hoisl

June 25, 2013
• **MDD**: software engineering technique, abstracting problem domain via models, M2M/M2T transformations

• **DS(M)L**: a specialized (modeling) language tailored for a particular application domain (e.g. access control, backup policies, or system auditing) in the context of MDD

• **DSML based on the MOF/UML**: de facto standard for software system modeling, standardized modeling extensions, tool support, maintained through the OMG, builds upon a standardized metamodel
Introduction, Definition, Motivation
<table>
<thead>
<tr>
<th>Objectives</th>
<th>Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>An integrated approach for modeling processes and process-related RBAC models (roles, hierarchies, statically and dynamically mutual exclusive tasks etc.)</td>
<td>Business processes, role-based access control (RBAC)</td>
</tr>
<tr>
<td>Integrated modeling of business processes and process-related duties; particularly the modeling of duties and associated tasks in business process models</td>
<td>Business processes, process-related duties</td>
</tr>
<tr>
<td>An approach to provide modeling support for the delegation of roles, tasks, and duties in the context of process-related RBAC models</td>
<td>Business processes, delegation of roles, tasks, and duties</td>
</tr>
<tr>
<td>Modeling of confidentiality and integrity of object flows in activity models</td>
<td>Data confidentiality and integrity</td>
</tr>
<tr>
<td>Modeling support for the notion of mutual exclusion and binding constraints for duties in process-related RBAC models</td>
<td>RBAC (consistency checks for duties)</td>
</tr>
<tr>
<td>Incorporation of data integrity and confidentiality into the MDD of process-driven SOAs</td>
<td>Integrity and confidentiality for service invocations</td>
</tr>
<tr>
<td>Integration of context constraints with process-related RBAC models and thereby supporting context-dependent task execution</td>
<td>Business processes, RBAC, context constraints</td>
</tr>
<tr>
<td>An approach for the definition of audit requirements and specification of audit rules at the modeling-level</td>
<td>Audit rules</td>
</tr>
</tbody>
</table>
• **Use case**: Integration of security-related DSML core language models
  
  - DSML core language model captures all relevant domain abstractions and specifies the relations between these abstractions
  - Integration enables the reuse of DSMLs by composing two or more languages into an integrated DSML to implement a new domain or to integrate domain viewpoints
Scenario-based Testing

- As it is almost impossible to completely test a complex software system, effective means are needed to select relevant tests, to express and to maintain them, and to automate test procedures whenever possible.

- Scenarios can help to reduce the risk of omitting or forgetting relevant test cases, as well as the risk of describing important tests insufficiently.

- Scenarios are used to specify user needs as well as to explore and to define (actual or intended) system behavior.

- Scenarios can be described in different ways at various abstraction levels (e.g., via structured text, graphical models, or formal specifications).
Scenario-based Testing
**Scenario-based Testing of DSML Core Language Models: Process Description**

---

**ad:** A scenario-based testing activity

- **(domain expert)** Create scenario descriptions
- **(DSML engineer)** Translate scenario descriptions into tests
- **(domain expert, DSML engineer)** Review scenario tests
- **(DSML engineer)** Perform meta-modeling action
- **(DSML engineer)** Run scenario tests

[tests succeed] [tests fail]
Scenario-based Testing of DSML Core Language Models: Problem Statement

Create natural-language scenario descriptions

Translate scenarios

Run scenario tests

Review scenario tests

Independent from testing platform

Specific on testing platform & metamodel language

Domain expert

DSML engineer
Scenario-based Testing of DSML Core Language Models: Our Approach

- Natural-language scenario descriptions on the requirements-level
- Executable test scenarios (for a dedicated platform)
- Transformation via linguistic rule-based step definitions
- Tool-support: testing framework based on Eclipse EMF, Epsilon, Xtext

```
Create natural-language scenario descriptions

Domain expert

Independent from testing platform

DSML engineer

Specific on testing platform & metamodel language

DSML engineer

Reuse step definitions

Step definitions

Review scenario tests

Run scenario tests
```
Example: Process Description

- Scenario testing a security-related DSML core language model composition
Example: Security-related DSML Core Language Models
Example: Integration of Security-related DSML Core Language Models
### Test case 2

<table>
<thead>
<tr>
<th>Test scenario 1</th>
<th>An AuditableEvent issued by a Transition must publish at least one Signal.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preconditions</td>
<td>AuditableEvent has all structural features of AuditEvent and Event.</td>
</tr>
<tr>
<td>Expected result</td>
<td>Instances of AuditableEvent must refer to at least one Signal instance.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Primary actors</th>
<th>System auditor, distributed-systems operator</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preconditions</td>
<td>All metamodel constraints for the source DSMLs must hold for DSML C.</td>
</tr>
<tr>
<td>Trigger/Setup</td>
<td>The model-transformation workflow to integrate the metamodels of DSML A &amp; DSML B is executed.</td>
</tr>
</tbody>
</table>

**Example:** Requirements-level Scenario Description (Excerpt)

Ascertain that each triggered AuditableEvent can be sensed by the monitoring facility.
Example: Scenario-based requirements specification language (metamodel & textual concrete syntax)

- **RequirementsSpecification:** "...
  - **Feature:** "...
    - In order to "...
    - As a "...
    - I want "...
  - **Scenario:** "...
    - Given "...
    - When "...
    - Then "...

- **NamedElement**
  - name : String [1]

- **Feature**
  - in_order_to : String [0..1]
  - as_a : String [0..1]
  - i_want : String [0..1]

- **Scenario**
  - given : String [0..*]
  - when : String [0..*]
  - then : String [1..*]
Example: Requirements-level Scenario Description (Alternative Format; Excerpt)

**Requirements Specification:** "DSMLs A and B integration"

**Feature:** "Monitor AuditableEvent"

**In order to** "ascertain that each triggered AuditableEvent can be sensed by the monitoring facility"

**As a** "system auditor and distributed-systems operator"

**I want** "that AuditableEvents shall publish Signals"

**Scenario:** "AuditableEvent shall publish at least one Signal"

**Given** "that EventSystemStateMachine.AuditableEvent has all features of EventSystemAuditEvent and StateMachine.Event"

**When** "in metamodel EventSystemStateMachine metaclass AuditableEvent references metaclass Signal"

**Then** "instances of EventSystemStateMachine.AuditableEvent shall refer to at least 1 Signal instance"

...
var stepDef : Map = Map {
    -- tests if multiplicity >= 1 between two classifiers
    Set {
        "^instances of (\S+)\.(\S+) (?:shall|must) refer to at least (\d+) (\S+) instances?$"
    }
    = "assertFalse("An $1 shall publish at least $2 $3.", $0!EClass.all->selectOne(c | c.name = "$1").eStructuralFeatures->first().lowerBound < $2);"
    -- more step definitions
};
--DSMLs A and B integration

@TestSuite
operation dsmls_a_and_b_integration() {
  --Monitor AuditableEvent:
  --In order to ascertain that each triggered AuditableEvent can be sensed by the monitoring facility
  --As a system auditor and distributed-systems operator
  --I want that AuditableEvents shall publish Signals

@TestCase
operation monitor_auditableevent() {
  --AuditableEvent shall publish at least one Signal

@TestScenario
  --Given that EventSystemStateMachine.AuditableEvent has all features of EventSystem.AuditEvent and StateMachine.
  Event
  $pre EventSystemStateMachine!EClass.all->selectOne(ae | ae.name = "AuditableEvent").eStructuralFeatures.size() =
  EventSystem!EClass.all->selectOne(ae | ae.name = "AuditEvent").eStructuralFeatures.size() + StateMachine!
  EClass.all->selectOne(ae | ae.name = "Event").eStructuralFeatures.size()

operation auditableevent_shall_publish_at_least_one_signal() {
  --When in metamodel EventSystemStateMachine metaclass AuditableEvent references metaclass Signal
  if (EventSystemStateMachine!EClass.all->selectOne(c | c.name = "AuditableEvent").eStructuralFeatures->first().
    eType.name = "Signal") {
    --Then instances of EventSystemStateMachine.AuditableEvent shall refer to at least 1 Signal instance
    assertFalse("An AuditableEvent shall publish at least 1 Signal.", EventSystemStateMachine!EClass.all->
      selectOne(c | c.name = "AuditableEvent").eStructuralFeatures->first().lowerBound < 1);
  }
}
Example: Scenario-test Reports

![Test Results]

- **default.test-gen [432ms]**
  - dsmls_a_and_b_integration [188ms]
  - compose_dsmls [126ms]
    - create_concept_auditableevent [75ms]
    - auditableevent_should_publish_signals [45ms]
  - monitor_auditableevent [47ms]
- **auditableevent_shall_publish_at_least_one_signal [39ms]**

Failure Trace:

```
org.eclipse.epsilon.eol.exceptions.EolAssertionException: An AuditableEvent shall publish at least 1 Signal.
```

![Test Results]

- **default.test-gen [284ms]**
  - dsmls_a_and_b_integration [138ms]
  - compose_dsmls [95ms]
    - create_concept_auditableevent [52ms]
    - auditableevent_should_publish_signals [39ms]
  - monitor_auditableevent [38ms]
- **auditableevent_shall_publish_at_least_one_signal [36ms]**
Concluding Remarks

• Discussion
  - **Ambiguity of requirements**: step definitions help to reduce the number of semantic variation points of ambiguous natural-language requirements
  - **Consistency of requirements**: executable scenario tests add to the conflict-free definition of requirements (otherwise tests will fail)
  - **Singularity of requirements**: pattern-based step definitions recognize conjugated scenario steps
  - **Traceability of requirements**: forward traceability via step definitions; backward traceability via the scenario-test report
  - **Verification of requirements**: evidence collection via executable test scenarios for the proof that system can satisfy requirements

• **Further information**: http://nm.wu.ac.at/modsec