Applying UML and software simulation for process definition, verification, and validation

Nien-Lin Hsueh, Wen-Hsiang Shen, Zhi-Wei Yang, Dong-Lin Yang

IST ‘08

2008. 8. 6

Park, Seung-Hun

© KAIST SE LAB 2008
Introduction (1/2)

- Process definition, verification and validation on the CMMI-based SPI
  - Define a clear and concise process that is readable for all process performers
  - Confirm if the processes are sufficient for CMMI requirements
  - Confirm that the processes are consistent with one another and executable in the organizations
  - Confirm if the processes proposed by EPG are suitable for stakeholders
Proposition

- UML-based approach to define and verify an organization’s process
  - Provides different views to present a process
  - Offers an extension mechanism to allow notion extension
  - Uses OCL to help model verification
- Process validation in a simulation environment prior to the pilot project execution
Overall approach

- Process model definition
- Pilot project execution
- Verification rules with OCL (include CMMI rules)
- Process model verification
- Process education (in SimSE)
- Process validation (in SimSE)
- UML profile technique

Flowchart:

1. Process model definition
2. Pilot project execution
3. Process model verification
4. Process education (in SimSE)
5. Process validation (in SimSE)
6. Model transformation
7. Transformation rule (semi-automatically)
8. Verified process model
9. UML profile technique
10. Revise (incorrect)
Process definition (1/5)

- Process structure
  - Metamodel of the static process structure with CMMI framework
### Process definition (2/5)

**Process structure**

- Apply part of SPEM profile to construct a process profile, called Process Model Profile (PMP)

<table>
<thead>
<tr>
<th>Stereotype</th>
<th>Base class</th>
<th>Stereotype parent</th>
<th>Notation</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>WorkArtifact</td>
<td>Core::Class</td>
<td></td>
<td></td>
<td>Customer requirements</td>
</tr>
<tr>
<td></td>
<td>ActivityGraph::ObjectFlowState</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EntryCriteria</td>
<td>Core::Constraint</td>
<td></td>
<td></td>
<td>Do “Requirement Analysis” before requirements drafted</td>
</tr>
<tr>
<td>ExitCriteria</td>
<td>Core::Constraint</td>
<td></td>
<td></td>
<td>Stop “Requirement Analysis” after requirement document finished</td>
</tr>
<tr>
<td>WorkProcess</td>
<td>Core::Operation</td>
<td></td>
<td></td>
<td>Product development</td>
</tr>
<tr>
<td></td>
<td>ActivityGraph::ActionState</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WorkPractice</td>
<td>Core::Operation</td>
<td>Work Process</td>
<td></td>
<td>Requirement analysis</td>
</tr>
<tr>
<td></td>
<td>ActivityGraph::ActionState</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>WorkStep</td>
<td>Core::Operation</td>
<td>Work Practice</td>
<td></td>
<td>Baseline requirements</td>
</tr>
<tr>
<td></td>
<td>ActivityGraph::ActionState</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ProcessMember</td>
<td>UseCases::Actor</td>
<td></td>
<td></td>
<td>Project leader</td>
</tr>
<tr>
<td>ProcessOwner</td>
<td>UseCases::Actor</td>
<td>ProcessMember</td>
<td></td>
<td>Process Definition Team</td>
</tr>
<tr>
<td>ProcessPerformer</td>
<td>UseCases::Actor</td>
<td>ProcessMember</td>
<td></td>
<td>Requirement Analyzer</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CMMI_ML</td>
<td>Core::Class</td>
<td></td>
<td></td>
<td>Maturity level 2</td>
</tr>
<tr>
<td>CMMI_PA</td>
<td>Core::Class</td>
<td></td>
<td></td>
<td>REQM, CM, PPQA</td>
</tr>
<tr>
<td>CMMI_Goal</td>
<td>Core::Class</td>
<td></td>
<td></td>
<td>Manage requirements</td>
</tr>
<tr>
<td>CMMI_Practice</td>
<td>Core::Class</td>
<td></td>
<td></td>
<td>Manage requirements changes</td>
</tr>
</tbody>
</table>
Process definition (3/5)

- Process structure (cont’d)
  - Example of a static process structure with CMMI framework

![Diagram of CMMI framework with REQM practices and development process](image-url)
Process definition (4/5)

- Process modeling with UML
  - Static process structure modeled by Class diagram
    - Shows process elements and the relationships among them
  - Behavior of process elements modeled by Statechart diagram
    - All transitions must be a work process defined in the class diagram
  - Dynamic process sequence modeled by Activity diagram
    - Shows the flows from activity to activity and the input/output WorkArtifacts for the activities
Process definition (5/5)

- Process modeling with UML (cont’d)
  - Example

**Class diagram**

- EPG
- Implementation
- Product Development
- Source Code
- System Proposal
- Customer Requirements
- System and Software Analysis

**Activity diagram**

- Elicitor: REQ Elicitor
- Analyzer: REQ Analyzer

**Statechart diagram**

- Customer Requirement Elicitation
- Integration and Delivery
- None
- Approved
- Accepted
Process verification (1/2)

- **Process rule**
  - Every element of process profile has an associated verification rule which inherits from UML metamodel
  - Each rule is automatically verified by OCL
  - There are two kinds of rules
    - Process related rule
      - Used to constrain process structure
    - CMMI-related rule
      - Used to verify whether the developed process model can meet the essential requirements of CMMI

Example of a process related rule

Example of a CMMI related rule

**Rule:** Each CMMI\_Practice should be realized by at least one WorkPractice.

```
metaOCL express:
{self.Relationship->select(oclIsTypeOf(Dependency)) -> select(k|
k.TaggedValueSetReference->exists(Name = 'Realized by' and
oclIsTypeOf(Stereotype ))).oclAsType(Dependency).NonOwningEnd
->select(oclIsTypeOf(Class)) -> select(p| p.TaggedValueSetReference->exists(Name =
'Work\_Practice' and oclIsTypeOf(Stereotype ))) ->size()>=1
}
```
Process verification (2/2)

- Process rule (cont’d)
  - Example
Process validation (1/4)

- Simulation model for process validation
  - Simulation model is built in two steps
    - Model transformation step
      - Developed process model is automatically transformed into a semi-finished SimSE model
    - Model refining step
      - Developers refine the semi-finished SimSE model manually
SimSE model

- **Object types**: Define templates for all objects that participate in the simulation. Every object type defined must descend from one of five meta-type: Employee, Artifact, Tool, Project, or Customer.

- **Start state**: Collection of objects that are present at the beginning of a simulation.

- **Actions**: Represent the set of activities in which the objects in the simulation can participate.

---

**Action Coding**

```javascript
{  
  name: String, 
  type: String, 
  key: true, 
  visible: true, 
  ... 
}
```

**Participant Coder**

```javascript
{  
  quantity: at least 1, 
  allowable types: Programmer, Tester 
}
```

**Participant CodeDoc**

```javascript
{  
  quantity: exactly 1, 
  allowable types: Code 
}
```

**Participant IDE**

```javascript
{  
  quantity: at least 1, 
  allowable types: Eclipse, JPad 
}
```

**Trigger userTrigger**

```javascript
{  
  type: User-initiated, 
  menuText: “Start coding” 
}
```

**Destroyer autoDestroyer**

```javascript
{  
  type: Autonomous, 
  overheadText: “I'm finished coding!” 
}
```

**Destroyer userDestroyer**

```javascript
{  
  type: User-initiated, 
  menuText: “Stop coding” 
}
```

---

**Project**

```javascript
{  
  name: String, 
  type: Autonomous, 
  overheadText: “I'm finished coding!” 
}
```

**SEProject**

```javascript
{  
  name: “Rocket Software”, 
  COFOfCode = 10000.00, 
  ... 
}
```

---

**Programmer Employee**

```javascript
{  
  name: String, 
  type: String, 
  key: true, 
  visible: true, 
  ... 
}
```

**Object Programmer**

```javascript
{  
  name = “Programmer”, 
  energy = true, 
  productiveTime = true, 
  errorRate = true, 
  minWal: 0.0, 
  maxWal: 1.0, 
  minDigits: 0, 
  maxDigits: 10, 
  hire: true, 
  hireRate: true, 
  visible: true, 
  ... 
}
```

**Object IDE**

```javascript
{  
  Eclipse: 
  purchased = true, 
  licenseValid = true, 
  JPad: 
  purchased = true, 
  licenseValid = true 
}
```
SimSE model (cont’d)

- Rule: Defines an effect of an action – how the simulation is affected when that action is active.
  - Distinguishes three types of rules:
    - Create objects rules,
    - Destroy objects rules,
    - Effect rules.

- Graphics: Refers to the graphical representations of all objects and the layout of the virtual office.
Performing process validation

- Obtain a score, indicating one’s understanding and familiarity of the process
  - Score is obtained by calculating the percentage of process understanding using action time
    \[ P_i \text{Understanding} = \frac{P_i \text{Best Time Elapsed}}{P_i \text{Action Time Elapsed}} \times 100 \]
  - If the player makes the right process, he/she will get high score, and vice versa
    - Find which parts are problematic
Conclusion

Summary

- Adopted a CMMI approach to guide the process definition, verification, and validation
  - In the process definition stage, proposed a specification to define process to explicitly represent the process for CMMI framework
  - In the process verification stage, process rules were represented as OCL and embedded in the process model
  - In the process validation stage, simulation model was transformed from process models as the core of simulation running in the simulation environment to be validated by process stakeholders

Future work

- Focus on process modeling applications in a distributed environment
My previous works

- Process modeling and tailoring based on the CMMI dependencies


- My previous works (cont’d)
  - Process modeling and tailoring based on the CMMI dependencies

Extended SPEM Process Structure package

![Diagram of SPEM Process Structure package]

19/26
My previous works (cont’d)

- Process modeling and tailoring based on the CMMI dependencies

The extended Dependencies package

The Basic Dependency package

The Derived Dependency package

- Uses dependency: one or more work products of target practice are used as inputs of source practice
- Implements dependency: one or more direct artifacts and one or more indirect artifacts or affirmations of the target practice are also the work products of the source practice
- Includes dependency: a part of the direct artifacts of the target practice are the work products of the source practice
My previous works (cont’d)

- Process modeling and tailoring based on the CMMI dependencies

Tailoring rule using OCL

```
context Classifier
inv ?IL_constrain:
let streamss(stn: Classifier) : Set(String) = stream(stn.stereotype)->collect(name)->asSet()
let st(stc: Classifier) : String =
  if streamss(stc)->size() = 1 then streamss(stc)->asSequence()->first().lower() else "" endif
let associatedSet(c: Classifier, outGoing: Boolean, checked: Set(Classifier), srcst: String, tarst: String):
  Set(Classifier) = let tarsts: Set(String) =
    if st(c) = srcst then Set(srcst, tarst) else Set(tarst); endif
  if outGoing = true then c.association.association.connection->select(isNavigable = true)->
    collect(participant0)->select(tac: Classifier | ac <> c and checked->
      excludes(ac) and tarsts->includes(st(ac))->asSet())
  else
    c.association->select(isNavigable = true).association.connection->collect(participant)->
    select(ac: Classifier | ac <> c and checked->excludes(ac) and tarsts->includes(st(ac)))->asSet()
  endif
let connectedSet(c: Classifier, outGoing: Boolean, visited: Set(Classifier), startingST: String, endingST: String):
  Set(Classifier) = let associated = associatedSet(c, outGoing, visited, startingST, endingST)
  if associated->size() = 0 then visited->including(c) else
    let newVisited = visited->including(c)
    associated->select(adc: Classifier | resultSet = visited->including(c))
  endif
  let ad2 = connectedSet(adc, outGoing, newVisited, startingST, endingST)
  resultSet->union(ad2)
endif

\(s(wd) = "specificpractice" or s(wd) = "genericpractice") \implies
connectedSet(self.true, Set), "specificpractice", "indirectpractice" ->
exists(c: Classifier | c="indirectpractice") or
connectedSet(self.true, Set), "specificpractice", "affirmations" -> exists(c: Classifier | c="affirmations") or
connectedSet(self.true, "genericpractice", "affirmations") -> exists(c: Classifier | c="affirmations") or
connectedSet(self.true, Set), "genericpractice", "indirectpractice" ->
exists(c: Classifier | c="indirectpractice") or
connectedSet(self.true, Set), "genericpractice", "affirmations" -> exists(c: Classifier | c="affirmations")
```

Process verification with OCLE
My previous works (cont’d)
- Developing a simulation model using SPEM-based process model

Step 1
Scoping

Step 2
Modeling

Step 3
Transforming

Identifying the scope of a simulation
Portion of project lifecycle

Developing a simulation model structure using UML
- Use Case Diagram
- Activity Diagram
- Activity Diagrams for each activity

Developing qualitative and quantitative models
- Cause-effect Diagram
- Quantitative equations and parameters

Generating a DEVS-Hybrid simulation model
- DEVS-Hybrid simulation model

Structural transformation
Behavioral transformation

Integration
Discussion (6/9)

- My previous works (cont’d)
  - Developing a simulation model using SPEM-based process model

UseCase diagram

Activity diagram
Discussion (7/9)

- My previous works (cont’d)
  - Developing a simulation model using SPEM-based process model

**Structural transformation algorithm**

```plaintext
Input: a UseCase diagram, an Activity diagram describing the overall structure
Output: Coupled model

For all use cases UC_i in the UseCase diagram
if UC_i's stereotype = “Phase” or “WorkDefinition”
  
  UC_i = Coupled Model

  For all UC_j ∈ S { 
    AC_j = UC_j
    /* Construct outgoing relations from an activity */
    For all AC_k ∈ T {
      UC_k = AC_k
      If (UC_k ∈ S) // if two activities are performed in the same phase
        IC = W_p
      Else { // if two activities are performed in the different phase
        EOC = W_p
        If (!(W_k ∈ Y or Y_phase)) // W_q does not already exist as an output
          // W_q does not already exist as an output
          Y = W_p
        }
    }
    /* Construct incoming relations to an activity */
    For all AC_k ∈ Z { 
      UC_k = AC_k
      If (AC_k ∈ S) // if two activities are performed in the same phase
        IC = W_q
      Else { // if two activities are performed in the different phase
        EOC = W_q
        If (!(W_k ∈ X)) // W_q does not already exist as an input
          X = W_q
        }
    }
  }
Else if UC_i's stereotype = “Activity”
  UC_i = Atomic Model
```

**Behavioral transformation algorithm**

```plaintext
Input: a UseCase diagram, Activity diagrams describing the behavior of the activities
Output: Atomic model

For all use cases in the UseCase diagram
if UC_i's stereotype = “Activity” {
  For the activity diagram which is mapped to UC_i {
    S_{step} = S_{step}
    X = E
    If (guard condition on a transition) {
      Y_{phase} = A
      C_{phase} = guard condition
    }
    Else
      Y = A
    
    For all events e in the Activity diagram {
      If (e = “after”) { // if the event is a TimeEvent
        \( \delta_{int} \) = (the source state, the target state)
        \( \lambda \) = (the source state, AE)
        \( t_{a} \) = time of "after"
      }
      Else {
        \( \delta_{int} \) = (the source state, the target state, e)
        \( t_{a} \) = infinity
      }
    }
  }
}
```
My previous works (cont’d)

- Developing a simulation model using SPEM-based process model
## Comparison

<table>
<thead>
<tr>
<th></th>
<th>Hsueh’s work</th>
<th>My work</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Process modeling</strong></td>
<td>• UML based on the part of SPEM profile including CMMI</td>
<td>• UML based on the SPEM profile including CMMI</td>
</tr>
<tr>
<td><strong>Process verification</strong></td>
<td>• Process related rules for constraining process structure</td>
<td>• Process tailoring rules for preserving the CMMI maturity level</td>
</tr>
<tr>
<td></td>
<td>• CMMI related rules for satisfying requirements of CMMI</td>
<td></td>
</tr>
<tr>
<td><strong>Process validation</strong></td>
<td>• Confirming the execution of a process</td>
<td>• Analyzing the various project parameters such as effort and schedule</td>
</tr>
<tr>
<td><strong>Advantages</strong></td>
<td>• Enable to confirm whether the process is executable or not</td>
<td>• Provide tailoring guidance for CMMI</td>
</tr>
<tr>
<td></td>
<td>• Help project members to learn and understand the process</td>
<td>• Help to quantitatively analyze the CMMI-based SPI strategies</td>
</tr>
<tr>
<td><strong>Limitation</strong></td>
<td>• Do not support the quantitative evaluation</td>
<td>• Require much additional effort to develop a simulator</td>
</tr>
</tbody>
</table>
## Process validation

### Model transformation

- Process model is transformed into a SimSE model by performing a series of model element mapping

<table>
<thead>
<tr>
<th>Source (PMP)</th>
<th>Transformation rule/guidance</th>
<th>Result (SimSE)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Class diagram</td>
<td>(E1) <strong>Implementation is a WorkPractice</strong></td>
<td>Transform a class stereotyped as a <strong>WorkPractice</strong> into an <strong>Action</strong></td>
</tr>
<tr>
<td></td>
<td>(E2) <strong>Implemenier is a ProcessPerformer, and can perform Implementation</strong></td>
<td>Transform a class stereotyped as a <strong>ProcessPerformer</strong> into a subtype of <strong>Employee</strong></td>
</tr>
<tr>
<td></td>
<td>(E3) <strong>Unit Test is a WorkArtifact, and is generated after Implementation</strong></td>
<td>Transform each <strong>WorkArtifact</strong> into a subtype of <strong>Artifact</strong> Transform each artifact generation into a rule</td>
</tr>
<tr>
<td>State-chart diagram</td>
<td>(E4) <strong>System Acceptance Form (SAF) has the states None, Approved, and Accepted</strong></td>
<td>(M) Transform states into a conditions</td>
</tr>
<tr>
<td></td>
<td>(E5) <strong>SAF transits into Accepted state from Approved when Integration and Delivery is performed</strong></td>
<td>(M) Transform state transitions into an effective rule</td>
</tr>
<tr>
<td>Activity diagram</td>
<td>(E6) <strong>System and Software Analysis (SSA) is performed after Customer Requirement Elicitation (CRE)</strong></td>
<td>Transform a sequence of activities into a trigger condition of the action</td>
</tr>
<tr>
<td></td>
<td>(E7) <strong>The Elicitor performs the Customer Requirement Elicitation (CRE) and the Analyzer performs the System and Software Analysis (SSA)</strong></td>
<td>(M) Transform swimlanes into rules to represent “improper performer generates more errors to its artifact after that needs more effort to correct”</td>
</tr>
<tr>
<td></td>
<td>(E8) <strong>The System and Software Analysis is performed after Customer Requirement Elicitation</strong></td>
<td>An effective rule “the efficiency of a proper performer is better than that of an improper one” is added to the rule of the action SSA (see Fig. 11)</td>
</tr>
</tbody>
</table>