Engineering Self-Adaptive Software Systems with Runtime Models

Seminar on QoS Attributes in Service- and Cloud-based Systems

Thomas Vogel
System Analysis and Modeling Group
Hasso Plattner Institute
University of Potsdam, Germany
Motivation

- Need to continuously change software
  - Lehman’s laws of software evolution [Lehman and Belady, 1985]
  - Software aging [Parnas, 1994]

⇒ **Software evolution and maintenance**
Motivation

- Need to continuously change software
  - Lehman’s laws of software evolution [Lehman and Belady, 1985]
  - Software aging [Parnas, 1994]

⇒ Software evolution and maintenance

- Software systems that are . . .
  - self- or context-aware
  - mission-critical
  - ultra-large-scale (ULS)
  - . . .
Motivation

• Need to continuously change software
  • Lehman’s laws of software evolution [Lehman and Belady, 1985]
  • Software aging [Parnas, 1994]

⇒ Software evolution and maintenance

• Software systems that are…
  • self- or context-aware
  • mission-critical
  • ultra-large-scale (ULS)
  • …

“Evolution in ULS systems will rarely occur in discrete, planned steps in a closed environment; instead it will be continuous and dynamic. The rules for continuous evolution must therefore be built into ULS systems […] so that they will be […] able to cope with dynamically changing environments without constant human intervention. Achieving this goal requires research on in situ control, reflection, and adaptation to ensure continuous adherence to system functional and quality-of-service policies in the context of rapidly changing operational demands and resource availability.”

[Northrop et al., 2006, p.33]
Motivation

• Need to continuously change software
  • Lehman’s laws of software evolution [Lehman and Belady, 1985]
  • Software aging [Parnas, 1994]

⇒ Software evolution and maintenance

• Software systems that are . . .
  • self- or context-aware
  • mission-critical
  • ultra-large-scale (ULS)
  • . . .

⇒ Self-adaptive Software [Cheng et al., 2009, de Lemos et al., 2012]

⇒ Autonomic Computing [Kephart and Chess, 2003]

Remark: Co-existence of evolution/maintenance and self-adaptation
(1) Cost-effective development
(2) Reflection capabilities
(3) Making feedback loops explicit
(4) Flexible (runtime) solutions

Related approaches, e.g.:

- *Rainbow* [Garlan et al., 2004] : (1), (2), (3), (4)
- *J3 Toolsuite* [Schmidt et al., 2008] : (1), (2), (3), (4)
Engineering Self-adaptive Software

(1) Cost-effective development
(2) Reflection capabilities
(3) Making feedback loops explicit
(4) Flexible (runtime) solutions

Related approaches, e.g.:

- **Rainbow** [Garlan et al., 2004] : (1), (2), (3), (4)
- **J3 Toolsuite** [Schmidt et al., 2008] : (1), (2), (3), (4)

Models at runtime for engineering adaptation engines: (1)-(4)
Feedback Loop consisting of

- **Adaptation steps**
  Monitor, Analyze, Plan, Execute

- **Knowledge**
  about the managed system and its context

General goal: leverage MDE techniques and benefits to the runtime environment [France and Rumpe, 2007, Blair et al., 2009]

⇒ **Models@run.time for adaptation steps & knowledge**
Knowledge

Models causally connected to the running system

- Typically, **one** model is employed (often an architectural model emphasizing one concern)
  (cf. related work in [Vogel and Giese, 2010])

- Simultaneous use of multiple runtime models
  → **abstraction levels** — PSM vs. PIM (solution vs. problem space)
    - PSM: easier to connect to the running system
    - PIM: easier to use by adaptation steps

- **concerns** — failures, performance, architectural constraints, . . .

- Different views on a running system
  → **reflection capabilities** enabled and used by adaptation steps
Knowledge — Reflection Models

Metamodels for PIMs

 Failures

- ComponentPlatform
  - ComponentType
    - instantiate()
  - Component
    - state: ComponentLifeCycle
      - DEPLOYED
      - STARTED
      - UNDEPLOYED
      - NOT_SUPPORTED
  - propertyTypes
    - Property
      - value: EJavaObject
  - components
    - 0..*
  - provides 1..*
  - requires 0..*

- InterfaceType
  - provides 1..*
  - requires 0..*

- Failure
  - source
  - target
  - failures

 Performance

- Server
  - getRunningIntances(): ELong
  - getInstanceCount(): ELong
  - getInvocationCount(): ELong
  - getToalInvocationTime(): ELong

- Component
  - uid: EString
  - runningInstances: ELong
  - instanceCount: ELong
  - startTime: ELong
  - runningInstancesMax: ELong
  - name: EString
  - getInvocationCount(): ELong
  - getMaxOfMaxTime(): ELong
  - getMinOfMinTime(): ELong
  - getToalInvocationTime(): ELong

- Connector
  - uid: EString
  - name: EString
  - invocationCount: ELong
  - maxTime: ELong
  - minTime: ELong
  - totalTime: ELong
Monitor

Synchronizing changes in the system to the reflection models

- Keeping runtime models up-to-date and consistent to each other
- Sensors (instrumentation): management APIs
- **Incremental**, event-driven updates: System $\rightarrow$ PSM
  (manually implemented adapter)
- **Incremental** model synchronization: PSM $\rightarrow$ PIM$_1$, PIM$_2$, ... 
  (Model synchronization engine based on Triple Graph Grammars (TGG))
Overall, 11 rules for PSM $\rightarrow$ PIM_{failures}
Monitors — Development costs

generated code from TGG rules

<table>
<thead>
<tr>
<th>PIMs</th>
<th>Proposed solution</th>
<th>Batch LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>#Rules</td>
<td>#Nodes/Rules</td>
</tr>
<tr>
<td>Simpl. Architectural Model</td>
<td>9</td>
<td>7,44</td>
</tr>
<tr>
<td>Performance Model</td>
<td>4</td>
<td>6,25</td>
</tr>
<tr>
<td>Failure Model</td>
<td>7</td>
<td>7,14</td>
</tr>
<tr>
<td>Sum</td>
<td>20</td>
<td></td>
</tr>
</tbody>
</table>

- **Proposed solution** — *incremental* synchronization
  - System $\rightarrow$ PSM: 2685 LOC for the reusable adapter
  - PSM $\rightarrow$ 3 PIMs: 20 TGG rules (generated >33k LOC)
- **Batch** — creates PIMs directly from scratch (*non-incremental*)
  - 902 LOC ($\approx$ 20 TGG rules)
- Declarative vs. imperative approaches

**Remark**: done for slightly different metamodels than shown here
Monitor — Performance

<table>
<thead>
<tr>
<th>Size</th>
<th>n=0</th>
<th>n=1</th>
<th>n=2</th>
<th>n=3</th>
<th>n=4</th>
<th>n=5</th>
<th>Batch [ms]</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>0</td>
<td>163</td>
<td>361</td>
<td>523</td>
<td>749</td>
<td>891</td>
<td>8037</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>152</td>
<td>272</td>
<td>457</td>
<td>585</td>
<td>790</td>
<td>9663</td>
</tr>
<tr>
<td>15</td>
<td>0</td>
<td>157</td>
<td>308</td>
<td>472</td>
<td>643</td>
<td>848</td>
<td>10811</td>
</tr>
<tr>
<td>20</td>
<td>0</td>
<td>170</td>
<td>325</td>
<td>481</td>
<td>623</td>
<td>820</td>
<td>12257</td>
</tr>
<tr>
<td>25</td>
<td>0</td>
<td>178</td>
<td>339</td>
<td>523</td>
<td>708</td>
<td>850</td>
<td>15311</td>
</tr>
</tbody>
</table>

| System → PSM | 0% | 92.8% | 94.1% | 95.6% | 95.2% | 96.3% | -          |
| PSM → 3 PIMs | 0% | 7.2%  | 5.9%  | 4.4%  | 4.8%  | 3.7%  | -          |

- **Size**: number of deployed beans
- Structural monitoring through event-driven sensors
- Processing \( n \) events and invoking **once** the model synchronization engine

**Remark**: done for slightly different metamodels than shown here
Analyzing the running system based on reflection models (PIMs)

- Identifying needs for adaptation (reactively)
- Structural checks expressed in **Story Patterns** (Story Pattern and Story Diagram Interpreter)
- Under certain conditions, **incremental** execution of Story Patterns
- Constraints expressed in the **Object Constraint Language (OCL)** (Existing engine from the Eclipse Model Development Tools)
- Model-based analysis techniques
Identifying failures or violations of architectural constraints

Story Pattern

if self.name = 'TShop'
then self.components.size() <= 1
else true
endif
Planning adaptations based on analysis results

- Changing reflection models (PIMs) (and in the end the system)
- **Story Patterns** defining in-place transformations (Story Pattern and Story Diagram Interpreter)
- Under certain conditions, **incremental** execution of Story Patterns
- **OCL expression** to check and manipulate models (Existing engine from the Eclipse Model Development Tools)
Switching connections between components

- **c₁:Component**
  - name = Shop
  - requires

- **i₁:Interface**
  - name = IWarehousing

- **co₁:Connector**
  - name = c₁

- **c₂:Component**
  - name = Warehousing
  - provides

- **i₂:Interface**
  - name = IWarehousing

- **co₂:Connector**
  - name = c₂

- **i₃:Interface**
  - name = IWarehousing
  - provides

- **c₃:Component**
  - name = Warehousing2

**Story Pattern**
Synchronizing changes of reflection models to the system: PIMs $\rightarrow$ PSM $\rightarrow$ System

- **PIM $\rightarrow$ PSM**
  - **Incremental** model synchronization: same rules as for monitoring due to bidirectionality of TGG
  - Story Patterns for default creation patterns in refinement transformations (*Factories*)

- **PSM $\rightarrow$ System**
  - Observing PSM changes performed by the model synch. engine
  - Incrementally enacting these changes through effectors (management APIs)
• Overall, 11 rules and 1 factory for $\text{PSM} \leftrightarrow \text{PIM}_{\text{failures}}$
Interplay of all those models?

if self.name = 'TShop'
then self.components.size() <= 1
else true
endif

name = InvalidTX
f1:
name = IWarehousing
i2:Interface
Failure
name = InvalidTX
f3:
failures
Failure
name = InvalidTX
f2:
failures
Failure
name = InvalidTX
f1:
name = IWarehousing
i2:Interface
Failure
name = InvalidTX
f3:
failures
Failure
name = InvalidTX
f2:
failures
Failure
name = InvalidTX

m:EjbModule
uid := ib.uid
i:Interface
c:Component
uid := i.uid
ib:EjbInterface
sb:SessionBean
tb:EjbInterfaceType ... ++
++++

if self.name = 'TShop'
then self.components.size() <= 1
else true
endif
Interplay of all those models?

if self.name = 'TShop'
then self.components.size() <= 1
else true
endif

name = InvalidTX
f1: name = IWarehousing
i2: Interface
Failure
name = InvalidTX
f3:
failures
Failure
name = InvalidTX
f2:
failures
Failure
name = InvalidTX
f2:
failures
Failure
name = InvalidTX
f2: failures Failure
failures
name = Shop
c1: Component
name = Warehousing
c2: Component
name = Warehousing
i1: Interface
name = IWarehousing
i2: Interface
name = IWarehousing
i3: Interface
name = IWarehousing
c3: Component
name = c1
co1: Connector
name = c2
co2: Connector
req uires
provides
--
--
++++
provides
++
--
⇒
Specifying and executing feedback loops

Specification — Modeling language

• Capturing the interplay of multiple runtime models
  [Vogel et al., 2010, Vogel et al., 2011]

• Making feedback loops explicit in the design of self-adaptive systems
  [Müller et al., 2008, Brun et al., 2009]

Execution — Model interpreter

• **Flexible** solutions and structures for feedback loops
  
  • Adaptive control [Kramer and Magee, 2007] ⇒ multiple loops
  
  • Uncertainty [Esfahani and Malek, 2012]
  
  • State-of-the-art frameworks often prescribe static solutions to single feedback loops (e.g., [Garlan et al., 2004, Schmidt et al., 2008])
Specifying and executing feedback loops

**Specification — Modeling language**
- Capturing the interplay of multiple runtime models  
  [Vogel et al., 2010, Vogel et al., 2011]
- Making feedback loops **explicit** in the design of self-adaptive systems  
  [Müller et al., 2008, Brun et al., 2009]

**Execution — Model interpreter**
- **Flexible** solutions and structures for feedback loops
  - Adaptive control  
    [Kramer and Magee, 2007] ⇒ multiple loops
  - Uncertainty  
    [Esfahani and Malek, 2012]
  - State-of-the-art frameworks often prescribe static solutions to single feedback loops (e.g.,  
    [Garlan et al., 2004, Schmidt et al., 2008] )

Executable Megamodels
A *megamodel* is a model that contains models and relations by means of model operations between those models.

In general:

```
Model <-> ModelOp <-> Model'
```

Model-Driven Architecture (MDA) example:

```
PIM  Transformation  PSM
```

- Research on model-driven software development (MDA, MDE)
  [Favre, 2005, Bézivin et al., 2003, Bézivin et al., 2004, Barbero et al., 2007]
- “Toward Megamodels at Runtime” [Vogel et al., 2010]
Modeling a Single Feedback Loop

Self-repair

Concrete syntax:

- **Initial state**
- **Final state**

**Remark**: Abstract syntax defined by a metamodel [Vogel and Giese, 2012]
Self-repair

<<EvaluationModel>>
Failure analysis rules

<<Monitor>>
Update

<<ReflectionModel>>
Architectural Model

<<EvaluationModel>>
Deep analysis rules

[else]

<<Analyze>>
Check for failures

<<Analyze>>
Deep check for failures
detailed results

<<Plan>>
Repair

<<Execute>>
edited

Analyzed

Start

Self-optimization

<<EvaluationModel>>
Queueing Model

<<ChangeModel>>
Parameter variability

<<ReflectionModel>>
Bottleneck identification

<<Analyze>>
Bottlenecks

<<Plan>>
Adjust params

<<Execute>>
done

Effected

Start

One solution: Linearizing Complete Feedback Loops
Modeling Interacting Feedback Loops

Self-repair

Self-optimization

One solution: Linearizing Complete Feedback Loops
Modeling Interacting Feedback Loops

Self-repair

Self-optimization

Complex model operations

One solution: Linearizing Complete Feedback Loops
Self-repair

Self-optmization

Complex model operations

One solution: Linearizing Complete Feedback Loops
Modeling Interacting Feedback Loops

Self-repair

Shared runtime model

Self-optimization

One solution: Linearizing Complete Feedback Loops
Modeling Hierarchies of Feedback Loops

Layer 0

Running System
Modeling Hierarchies of Feedback Loops

Layer 0

Running System

Layer 1

Self-repair

<<EvaluationModel>>
Failure analysis rules

<<Analyze>>
Check for failures

<<Monitor>>
updated model

<<ReflectionModel>>
Architectural Model

TGG Rules

<<ExecutionModel>>
TGG Rules

<<ChangeModel>>
Repair strategies

<<Plan>>
Repair

<<Execute>>
Effected

Layer 2
directly uses the megamodel of Layer 1
• no specific sensors and effectors required
• adapts the models or control flow of the Layer 1 megamodel
• interpreter (flexibility)!

Causal connection
• sensors + effectors required
• implementation efforts!

Layer 1

updated model

no failures

[c since 'no failures' > 5]

Layer 0
Layer 0

Running System

Layer 1

<table>
<thead>
<tr>
<th>Self-repair</th>
</tr>
</thead>
<tbody>
<tr>
<td>Failure analysis rules</td>
</tr>
<tr>
<td>Check for failures</td>
</tr>
<tr>
<td>Analyzed</td>
</tr>
<tr>
<td>Adapted</td>
</tr>
<tr>
<td>Repair</td>
</tr>
<tr>
<td>Repair strategies</td>
</tr>
<tr>
<td>Effected</td>
</tr>
</tbody>
</table>

Causal connection

- sensors + effectors required
- implementation efforts!
Modeling Hierarchies of Feedback Loops

Layer 2

- **Self-repair-strategies**
  - <<EvaluationModel>>
  - Repair strategies analysis rules
  - <<ChangeModel>>
  - Repair strategies synthesis rules
  - <<Analyze>>
  - Check success rate
  - checked
  - <<Plan>>
  - Synthesize new repair strategies
  - synthesized
  - <<Execute>>
  - Replace strategies
  - replaced
  - <<ReflectionModel>>
  - Self-repair
  - Adapted

Layer 1

- **Self-repair**
  - <<EvaluationModel>>
  - Failure analysis rules
  - [c since 'no failures' > 5]
  - Self-repair-strategies
  - Adapted
  - <<ChangeModel>>
  - Repair strategies
  - <<Plan>>
  - Repair
  - repaired
  - <<Execute>>
  - Effect
  - done

Layer 0

- **Running System**

Causal connection
- sensors + effectors required
- implementation efforts!

Causal connection
- **Architectural Model**
  - <<MonitoringModel>>
  - TGG Rules
  - <<ExecutionModel>>
  - Start
  - <<Monitor>>
  - Update
  - updated model
  - <<Analyze>>
  - Check for failures
  - no failures
  - <<Analyze>>
  - Analyzed
  - <<ReflectionModel>>
  - Adapted
  - <<Plan>>
  - Repair
  - <<Execute>>
  - Effect
  - done
Modeling Hierarchies of Feedback Loops

Layer_2 directly uses the megamodel of Layer_1

- no specific sensors and effectors required
- adapts the models or control flow of the Layer_1 megamodel
- interpreter (flexibility)!

Causal connection
- sensors + effectors required
- implementation efforts!

Layer_0

Running System
Conclusion

Models at runtime

- Adaptation steps and knowledge
- Single and multiple feedback loops

Discussion

1. Cost-effective development
2. Reflection capabilities
3. Making feedback loops explicit
4. Flexible (runtime) solutions

... while being runtime efficient (incremental, on-line techniques)

Interests:

- Techniques, algorithms, models, and tools for QoS attributes
- Software architecture ↔ multiple QoS attributes
Further Reading

References I


Toward Megamodels at Runtime.
(best paper).

The Role of Models and Megamodels at Runtime.

Used Sources
- Slide 1: Dagstuhl figure from http://www.dagstuhl.de/.
- Slide 2: Ultra-large-scale systems [Northrop et al., 2006].