Engineering Self-Adaptive Software Systems with Runtime Models

Symposium on Future Trends in Service-Oriented Computing
Potsdam, Germany, June 14-15, 2012

Thomas Vogel
HPI Research School
System Analysis and Modeling Group
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
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)
(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@run.time for engineering adaptation engines:** (1)-(4)
Feedback Loop consisting of

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

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

- **MAPE-K** [Kephart and Chess, 2003]

Adaptation Engine

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

Metamodel of a PSM
Metamodel of a PSM
Simplified
Knowledge — Reflection Models

Metamodels for PIMs

Failures

- ComponentPlatform
  - ComponentType
    - instantiate()
  - PropertyType
  - Component
    - state : ComponentLifeCycle
    - properties
      - Property
      - value : EJavaObject
  - failures

Performance

- Interface
  - provides 1..*
  - requires 0..*
- Connector
  - source
  - target
  - inConnectors
  - outConnectors
- Server
  - getRunningIntances() : ELong
  - getInstanceCount() : ELong
  - getInvocationCount() : ELong
  - getTotalInvocationTime() : ELong
- Component
  - uid : EString
  - runningInstances : ELong
  - instanceCount : ELong
  - startTme : ELong
  - runningInstancesMax : ELong
  - name : EString
  - getInvocationCount() : ELong
  - getMaxOfMaxTime() : ELong
  - getMinOfMinTime() : ELong
  - getTotalInvocationTime() : ELong
- Connector
  - uid : EString
  - name : EString
  - invocationCount : ELong
  - maxTime : ELong
  - minTime : ELong
  - totalTime : ELong
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$_2$, $\ldots$
  (Model synchronization engine based on Triple Graph Grammars (TGG))

[MRT09,MiSE10]
Monitor — TGG Rules

TGG rule for PSM → PIM\textsubscript{failures}

- Overall, 11 rules for PSM → PIM\textsubscript{failures}
Monitor — Development costs

### Proposed solution

<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 → PSM: 2685 LOC for the reusable adapter
  - PSM → 3 PIMs: 20 TGG rules (generated >33k LOC)
- **Batch** — creates PIMs directly from scratch (**non-incremental**)
  - 902 LOC (≈ 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>

<table>
<thead>
<tr>
<th></th>
<th>System → PSM</th>
<th>0%</th>
<th>92.8%</th>
<th>94.1%</th>
<th>95.6%</th>
<th>95.2%</th>
<th>96.3%</th>
<th>-</th>
</tr>
</thead>
<tbody>
<tr>
<td>PSM → 3 PIMs</td>
<td>0%</td>
<td>7.2%</td>
<td>5.9%</td>
<td>4.4%</td>
<td>4.8%</td>
<td>3.7%</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

- **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**

- \( f_1 : \text{Failure} \)
  - name = InvalidTX
  - failures

- \( i_2 : \text{Interface} \)
  - name = IWarehousing
  - failures

- \( f_2 : \text{Failure} \)
  - name = InvalidTX

- \( f_3 : \text{Failure} \)
  - name = InvalidTX

**OCL expression**

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

**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

**Story Pattern**

```
c1:Component
  name = Shop
  requires

i1:Interface
  name = IWarehousing

co1:Connector
  name = c1

i2:Interface
  name = IWarehousing
  provides

co2:Connector
  name = c2

i3:Interface
  name = IWarehousing
  provides

co3:Connector
  name = c3

name = Warehousing
name = Shop

requires

name = IWarehousing
name = c1

name = IWarehousing
name = c2

name = Warehousing2
name = Warehousing
```
Synchronizing changes of reflection models to the system: PIMs → PSM → System

- PIM → 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 → System
  - Observing PSM changes performed by the model synch. engine
  - Incrementally enacting these changes through effectors (management APIs)
TGG rule for PSM $\leftrightarrow$ PIM$^\text{failure}$

- Overall, 11 rules and 1 factory for PSM $\leftrightarrow$ PIM$^\text{failure}$

**Factory required!**
Interplay of all those models?

[MRT10, MiSE11, SEAMS12]
Specifying and executing feedback loops

Specification — Modeling language

- Capturing the interplay of multiple runtime models
  [Vogel et al., 2010b, 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

- Coordinated execution/usage of multiple runtime models
- **Flexible** solutions and structures for feedback loops
  - Adaptable feedback loops (adaptive control)
  - 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., 2010b, 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**
- Coordinated execution/usage of multiple runtime models
- **Flexible** solutions and structures for feedback loops
  - Adaptable feedback loops (adaptive control)
  - State-of-the-art frameworks often prescribe static solutions to single feedback loops (e.g., [Garlan et al., 2004, Schmidt et al., 2008])

Executable Runtime Megamodels
Megamodels

Definition (Megamodel)

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

In general:

![Diagram showing model operations](image)

Model-Driven Architecture (MDA) example:

- 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., 2010b]
An Example: Self-repair

Legend
(concrete syntax)

- Initial state
- Final state

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

Legend (concrete syntax)

- Initial state
- Final state

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

Analyzed
Start

Legend
(concrete syntax)

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

Legend
(concrete syntax)

- Initial state
- Final state
- Model
- Operation
- Control flow
- [condition]
- [else]
- [t1]
- [t2]
- [c since ‘no failures’ > 5]

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

Legend (concrete syntax)

- Initial state
- Final state
- Model Operation
- Control flow

Remark: Abstract syntax defined by a metamodel [Vogel and Giese, 2012a]
Modeling Interacting Feedback Loops

Self-repair

Self-optimization

One solution: Linearizing Complete Feedback Loops
Modeling Interacting Feedback Loops

Self-repair

Self-optimization

One solution: Linearizing Complete Feedback Loops
Complex model operations

One solution: Linearizing Complete Feedback Loops
Modeling Interacting Feedback Loops

Self-repair

- <<ReflectionModel>> Architectural Model
- TGG Rules
- <<ExecutionModel>>
- Effect done

Self-optimization

- <<ReflectionModel>> Architectural Model
- TGG Rules
- <<ExecutionModel>>
- Effect done

Complex model operations

One solution: Linearizing Complete Feedback Loops

Modeling Interacting Feedback Loops

Self-repair

1. **Start**
2. **Monitor**
   - **Update** (architectural model)
   - **Check for failures**
   - **Analyze**
     - **Analysis rules**
   - **Reflection Model**
     - **TGG Rules**
   - **Repair**
   - **Effect**
3. **Done**

Self-optimization

1. **Start**
2. **Monitor**
   - **Update** (architectural model)
   - **Analyze**
   - **Queueing Model**
   - **Bottleneck identification**
   - **Plan**
     - **Adjust params**
   - **Effect**
3. **Done**

Shared runtime model

One solution: Linearizing Complete Feedback Loops

Other Solutions...

<table>
<thead>
<tr>
<th>Generic</th>
<th>Self-repair</th>
<th>Self-optimization</th>
<th>Composition</th>
</tr>
</thead>
</table>

- **Generic**
  - M
  - A
  - P
  - E

- **Self-repair**
  - M
  - A
  - P
  - E
  - Analyzed
  - Start
  - Effected

- **Self-optimization**
  - M
  - A
  - P
  - E
  - Analyzed
  - Analyze
  - Start
  - Effected

- **Composition**
  - M
  - A
  - P
  - M
  - A
  - P
  - M
  - A
  - P
  - E

~~ Patterns for control in self-adaptive systems [Weyns et al., 2012]
Modeling Hierarchies of Feedback Loops

\( \text{Layer}_0 \)

Running System
Modeling Hierarchies of Feedback Loops

Layer 1

Layer 0

Running System
Modeling Hierarchies of Feedback Loops

Layer 0

Running System

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
  - <<Monitor>>: Check success rate
  - <<Analyze>>: Synthesize new repair strategies
  - <<Plan>>: Replace strategies
  - <<Execute>>: Replaced
  - <<ReflectionModel>>: Self-repair
  - Adapted
  - Adapt

Layer 1

- **Self-repair**
  - <<EvaluationModel>>: Failure analysis rules
  - <<Analyze>>: Check for failures
  - <<Plan>>: Repair strategies
  - <<Execute>>: Effected
  - <<ReflectionModel>>: Adapted
  - Architectural Model
  - TGG Rules
  - [c since 'no failures' > 5]

Layer 0

- **Running System**
- **Causal connection**
  - sensors + effectors required
  - implementation efforts!
Modeling Hierarchies of Feedback Loops

Layer 2

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
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)
References

Software engineering processes for self-adaptive systems.
Springer-Verlag.
(to be published).

Traceability and Provenance Issues in Global Model Management.

MDA components: Challenges and Opportunities.
In First Intl. Workshop on Metamodelling for MDA, pages 23–41.

On the Need for Megamodels.

Models@run.time.

Comprehensive support for management of Enterprise Applications.

Engineering Self-Adaptive Systems through Feedback Loops.


(to be published).

In Language Engineering for Model-Driven Software Development, number 04101 in Dagstuhl Seminar Proc. IBFI.
References II

Model-driven Development of Complex Software: A Research Roadmap.

Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure.

Graph Transformations for MDE, Adaptation, and Models at Runtime, volume 7320 of LNCS.
Springer.
(to be published).

A Model-Driven Configuration Management System for Advanced IT Service Management.

The Vision of Autonomic Computing.

Program evolution: processes of software change.
Academic Press Professional, Inc., San Diego, CA, USA.

Visibility of control in adaptive systems.

Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA.

Software aging.

Simplifying autonomic enterprise Java Bean applications via model-driven engineering and simulation.

Autonomous Reconfiguration Procedures for EJB-based Enterprise Applications.
Adaptation and Abstract Runtime Models.

Language and Framework Requirements for Adaptation Models.


Requirements and Assessment of Languages and Frameworks for Adaptation Models.
In MoDELS 2011 Workshops, volume 7167 of LNCS, pages 167–182. Springer.

Incremental Model Synchronization for Efficient Run-time Monitoring.

Model-Driven Architectural Monitoring and Adaptation for Autonomic Systems.
In Proceedings of the 6th IEEE/ACM International Conference on Autonomic Computing and Communications (ICAC 2009), Barcelona, Spain, pages 67–68. ACM.

Incremental Model Synchronization for Efficient Run-Time Monitoring.
In MoDELS 2009 Workshops, volume 6002 of LNCS, pages 124–139. Springer.

Toward Megamodels at Runtime.
(best paper).

The Role of Models and Megamodels at Runtime.
In MoDELS 2010 Workshops, volume 6627 of LNCS, pages 224–238. Springer.

(to be published).