Software Engineering for Self-Adaptive Systems & Self-Aware Computing


Holger Giese
Head of the System Analysis & Modeling Group
Hasso Plattner Institute for Software Systems Engineering
University of Potsdam, Germany
holger.giese@hpi.uni-potsdam.de
Outline

1. **MECHATRONIC UML**
2. **ExecUtable Runtime MegA models (EUREMA)**
3. **Challenges Ahead**
4. **Outlook**
1. **Mechatronic UML**

At the level of code it seems impossible to build trustworthy advanced system of systems:

Modeling separately

- the integration of intelligent behavior,
- the integration with control theory,
- the real-time coordination, and
- the reconfiguration at the level of agents.

- Analyze the models in a compositional manner
- Synthesize the code
Application Example: Railcab System (1/2)

A system of *autonomous shuttles* that operate on demand and in a decentralized manner using a *wireless network*.

System of systems
- Hard real-time
- Safety-critical
- Self-Optimization
Application Example: Railcab System (2/2)

Domains:

- Logistic
- Real-time coordination
- Local control
- Electronics
- Mechanics

⇒ Integration of the different worlds
⇒ Self-optimization at multiple levels
⇒ Self-adaptation/self-coordination via software
Micro and Macro Architecture

- **Autonomous subsystems (shuttles)**
- Within: strict hierarchies
- Outside: complex coordination
Micro Architecture

- **Autonomous subsystems (shuttles)**
- Within: strict hierarchies
- Outside: complex coordination
Micro Architecture

- **Operator-Controller Module [ICINCO04]**
  - Cognitive operator ("intelligence")
    - decoupled from the hard real-time processing
  - **Reflective operator**
    - Real-time coordination and reconfiguration
  - **Controller**
    - Control via sensors and actuators in hard real-time
OCM & Reference Architecture
MECHATRONIC UML: Components

- Model the structure of the Software with hybrid UML components with
  - Hybrid behavior
    - Regular ports (discrete)
    - Continuous ports
    - Hybrid ports
  - Reconfiguration
    - Permanent ports
    - potential ports
Integration Reflective Operator & Controller

- Hybrid components
  - UML components (Fujaba)
  - Block diagrams (CAMeL)
- Hybride Statecharts can embed subordinated hybrid components
  - Controller or
  - The reflective operator of subordinated OCMs
- Interface statecharts enable **modular reconfiguration** across the boundaries of hybrid components
- Automatic **check** for correct embedding

[FSE04]
The cognitive operator is decoupled from the rest:

- We **check** that the reflective operator realizes a "**Filter**" which excludes unsafe reactions.
- The cognitive operator can "**guide**" the reflective operator as long as the commands given are considered to be safe and occur in time.
Strict Hierarchies

- **Concepts [FSE04]:**
  - Hybrid components: UML components or block diagrams
  - Hybride Statecharts embed hybrid components (controller or the reflective operator of subordinated OCMs)
  - Interface statecharts enable **modular reconfiguration** across the boundaries of hybrid components
Strict Hierarchies & Reference Architecture

- **Difference:**
  - Hierarchy of parts which include change management functionality ⇒ self-adaptation at multiple levels
  - Reflective operator includes functionality as well as change management ⇒ separation is less strict!

Distributed over the cognitive operators (may build a hierarchy)
Distributed over the reflective operators (strict hierarchical coordination)
Distributed over the controllers and reflective operators (may build a hierarchy)
Macro Architecture

- **Autonomous subsystems (shuttles)**
- Within: strict hierarchies
- Outside: complex coordination
Macro Architecture: Coordination

- **Real-time coordination via pattern** [ESEC/FSE03]
  - Real-time protocol state machines for each role
  - Real-time state machines for each connector

- **Rule-based reconfiguration (self-coordination)** [ICSE06]
  - Rules for instantiation and deletion of patterns

![Diagram](image)
**Complex Coordination & Reference Architecture**

- **Difference:**
  - Pattern capture component interaction as well as its instantiation ⇒ self-coordination
  - No new change plans but only choices which can be made by the local cognitive operators

Only implicit in the degrees of freedom for the rule-based configuration

Rule-based configuration distributed over the patterns and the components realizing the pattern roles
Real-Time Coordination via Patterns

Pattern (Distance Coordination):
- **Model**: Statecharts for roles and connector
- **Specification**: required OCL RT properties

Components (Shuttles):
- **Model**: Statecharts for ports (refined roles) and synchronization
- **Specification**: local OCL constraints
Complex Coordination: Role Protocols

**Role: FrontRole**

- **noConvoy** / rearRole.convoyProposalRejected
  - default → wait
  - answer

- **convoy**
  - rearRole.breakConvoyProposal / rearRole.breakConvoy
  - default

- **wait** after (9 msec)
  - / rearRole.frontRoleData

**Role: RearRole**

- **noConvoy**
  - frontRole.convoyProposalRejected
  - default
  - / frontRole.convoyProposal

- **convoy**
  - frontRole.breakConvoyProposal
  - default

- **wait**
  - / frontRole.breakConvoyProposalRejected

- **incoming**
  - frontRole.frontRoleData
  - emergency after ([1, x] msec)

**Connector:**
- buffer with maximal delay of 5 msec
- modeled faults: only full communication break down
Problem:
- Shuttles move and create resp. delete Distance Coordination patterns
- Arbitrary large topologies with moving shuttles

Solution:
- State = Graph
- Reconfiguration rules = graph transformation rules
- Safety properties = forbidden graphs
  ⇒ Formal Verification possible
Rule-Based Reconfiguration (2/2)

Apply Graph Transformation Systems

- Map the tracks
- Map the shuttles
- Map the movement of shuttles to rules
- Map the reconfiguration to rules

Rule:

\[
\begin{align*}
&t1:Track \rightarrow t2:Track \\
&s1:Shuttle \rightarrow s1:Shuttle
\end{align*}
\]
Application Example: Self-Coordination

- **Cognitive Operators:** do self-optimization
  - Maneuver planning
  - Convoy planning
  - Shuttle planning

- **Reflective Operator:** switch to guarantee safety
  - Realize maneuvers planned by the cognitive operator(s)
  - Recognize timeouts and enforced related safety maneuvers
  - Detect problems of controllers and enforced related safety maneuvers
Observation: Very difficult formal analysis techniques can guarantee some safety goals (validity of the models is guaranteed to some extent by synthesis).
Application Example: Self-Optimization

- **Cognitive Operators:** do distributed self-optimization
  - Distributed learning of a model of the track (environment)
  - Local learning of a model of the shuttle (system hardware)
  - Planning an adaptation in form of an optimal trajectory
- **Reflective Operator:** switch to robust local control if necessary
Models at Run-Time (2/2)

Development time:
- Goals
- Function
- Context
- Adaption

Run-time:
- Function
- Context
- Goals

Mathematical algorithms guarantee optimal trajectories (in case of invalid models a related diagnosis and fallback to a robust control is used as backup)

Possible benefits: Up-to-date context models are available earlier and better validated when multiple subsystems exchange data about their context.

Observation:
2. ExecUtable Runtime MegA models (EUREMA)

- Executable EMF megamodels kept alive at runtime with
  - Multiple runtime models
  - Activities are model operations (e.g., monitor + execute for EJBs with TGG)
- Multiple loops
- Multiple layers
- Runtime interpreter for adaptation engines permits high degree of flexibility
- Leverages the co-existence of self-adaptation and evolution
- Modules and runtime models can to some extent be reused


EUREMA: Knowledge & Runtime Models

Mega Model = „Model of Models and Operations on Models“

Idea:
- **Runtime models** are maintained at runtime
- **Runtime mega models** describe adaptation activities (MAPE)
- **Runtime interpreter** for runtime mega models

Giese | Dagstuhl 15041 | Model-driven algorithms and architectures for self-aware computing systems
EUREMA: Use MDE for Model Operations

Options for using MDE for model operations:

- **Monitor/Execute**: techniques for model synchronization can be employed (e.g., Triple Graph Grammar (TGG))

- **Analysis**: techniques that can operate on models with meta models such as OCL, model transformations, etc. can be employed.

- **Plan**: techniques that can operate on models with meta models such as OCL, model transformations, etc. can be employed.
EUREMA: Use MDE for Model Operations (1/2)

MDE for Monitor/Execute:

- Employ Triple Graph Grammar (TGG) for the model operations monitor and execute (at once)

- Synchronize runtime models incrementally between the modules and the managed element (faster as manual implementations)
  
  □ Extract abstract runtime models for different modules as required from unchanged EJB applications

  □ Adapt managed subsystem incrementally via model (parameter and structural adaptation)


**EUREMA: Use MDE for Model Operations (2/2)**

**Benefits:**
- The supported incremental processing provides low overhead monitoring and executing solution
- Permits sensors and effectors at a higher level of abstraction
- Model transformation technique can be used to map this high level information to analysis models used by the EUREMA modules

<table>
<thead>
<tr>
<th>Target Model</th>
<th>Model-Driven Approach</th>
<th>NIA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simpl. Architectural Model</td>
<td>9 7.44 15259 357</td>
<td></td>
</tr>
<tr>
<td>Performance Model</td>
<td>4 6.25 5979 253</td>
<td></td>
</tr>
<tr>
<td>Failure Model</td>
<td>7 7.14 12133 292</td>
<td></td>
</tr>
<tr>
<td>Sum</td>
<td>20 7.14 33371 902</td>
<td></td>
</tr>
</tbody>
</table>

**Limitations:**
- One generic adapter to the model world is initially required that requires usually more effort than an ad hoc monitoring effort

<table>
<thead>
<tr>
<th>Size</th>
<th>NIA</th>
<th>Model-Driven Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$S$</td>
<td>$B$ n=0 n=1 n=2 n=3 n=4 n=5</td>
</tr>
<tr>
<td>5</td>
<td>8037</td>
<td>20967 0 163 361 523 749 891 10733</td>
</tr>
<tr>
<td>10</td>
<td>9663</td>
<td>43054 0 152 272 457 585 790 23270</td>
</tr>
<tr>
<td>15</td>
<td>10811</td>
<td>72984 0 157 308 472 643 848 36488</td>
</tr>
<tr>
<td>20</td>
<td>12257</td>
<td>105671 0 170 325 481 623 820 55491</td>
</tr>
<tr>
<td>25</td>
<td>15311</td>
<td>142778 0 178 339 523 708 850 72531</td>
</tr>
</tbody>
</table>
Layer Diagrams: Example

Layer-0
:Adaptable Software

Layer-1
:MAPE-K
<trigger>

Layer-2
MAPE
:Planning

Layer-1
MAPE
:Adaptation

Layer-0
:Application
Main concepts:

- **Layers:**
  - Layer 0: core software
  - Layer 1: adaptation engine
  - Layer 2: higher-order adaptation behavior (e.g., planning)

- **Modules:**
  - megamodel modules: FDLs
  - software modules: legacy software

- **Relationships:**
  - Sense: trigger modules
  - Effects: effects of the modules
  - Use: use of megamodel elements of a module
Feedback Loop Diagrams (FLDs): Example
### Feedback Loop Diagrams (FLDs)

#### Concepts:
- **Helper states:**
  - Initial state: start of the execution
  - Final state: end of the execution
  - Destruction state: end of the execution and termination of the module

- **Model operations:**
  - Simple model operations: mapped to software or other modeling techniques (e.g., TGGs)
  - Complex model operations: mapped to modules

- **Models:**
  - Runtime models
  - EUREMA models

---

<table>
<thead>
<tr>
<th>Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>● Initial state</td>
</tr>
<tr>
<td>○ Final state</td>
</tr>
<tr>
<td>× Destruction state</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Model Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
</tr>
<tr>
<td>t2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Complex Model Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>t1</td>
</tr>
<tr>
<td>t2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Models</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
</tr>
<tr>
<td>FLD Model</td>
</tr>
</tbody>
</table>
Feedback Loop Diagrams (FLDs)

- **Concepts:**
  - **Control flow:**
    - Arrow: ordering
    - Rhombus: alternative flows of control
  - **Model usage:**
    - r: read
    - w: write
    - a: append
EUREMA: Self-Repair Example

Self-repair

Failure analysis rules

Check for failures

Adapted

Repair strategies

Repaired

Architectural Model

Reflection Model

Execute

Update

Monitor

MAPE

RtException; 10s; Monitor;

Adaptable Software
EUREMA: Modular Self-Repair Example

Modeling a Feedback Loop

Variability modeling. Parts of a feedback loop that are specified in dedicated FLDs can be reused in other feedback loops or they can be replaced by alternative parts specified in other FLDs. The latter leverages the modeling of variability for feedback loops in an adaptation engine.

For instance, the analysis activity of the Self-repair feedback loop depicted in Figure 3.3 can be abstracted and specified in its own FLD called Self-repair-A as shown in Figure 3.6. This analysis activity has one initial state (Start) and two final states reflecting whether critical failures have been identified (Failures) or not (OK).

A complex model operation defines a signature to invoke a megamodel module defined by an FLD. In general, a signature refers to the initial and final states of the corresponding FLD and to runtime models that have to be provided as parameters of the invocations. In the example, based on the initial and final states of the Self-repair-A FLD (Figure 3.6), the complex model operation shown in Figure 3.7a has one entry compartment called Start and two exit compartments called Failures and OK. Thus, initial and final states of an FLD are mapped to entry and exit compartments of a complex model operation, respectively. This ensures that the feedback loop using a complex model operation can properly invoke a module by defining the initial state for starting the execution of the module, and that it can properly resume execution after the invoked module has terminated by referring to the final states. If an FLD specifies exactly one initial or final state, the entry or exit points for execution are uniquely defined such that the entry or exit compartments of the complex model operation can be omitted. This is depicted in Figure 3.7b that shows the complex model operation for the Self-repair-A FLD, which omits the explicit entry compartment since the FLD has exactly one initial state. In general, the entry (exit) compartments of a complex model operation have to be a subset of the initial (final) states of the corresponding FLD. However, all those final states that are reachable from the initial states selected for entry compartments must be selected for exit compartments. Moreover, the signature of an FLD defines which runtime models have to be provided as parameters when invoking a module. In the example, it is the Architectural Model as shown in Figure 3.7, while the other runtime models used in the Self-repair-A megamodel module, namely Failure analysis rules and Deep analysis rules, are provided by the module itself.

Finally, a complex model operation is labeled with an icon, a small rounded rectangle, to distinguish it from the other type of model operations in FLDs and to reveal that it uses and invokes another megamodel module defined by an FLD. Moreover, when using a complex model operation in an FLD, it must be given a name for the variable \(<\text{var}>\) (cf. Figure 3.7), which is bound to a concrete megamodel module that actually should be invoked.
EUREMA models can already include alternatives (variability) that can be activated by adjusting the EUREMA model at runtime.

- Module-level
- Software-level
4.2 Coordination of Multiple Feedback Loops

Though the concurrent and independent execution of multiple feedback loops is conceivable, there are usually interferences or interactions between feedback loops that have to handled by coordinating adaptations of the different loops. For example, there are typical concerns that are competing, like failures and performance, which might result in conflicting adaptations. The self-repair feedback loop might perform an adaptation to heal a failure in the adaptable software, which might degrade the software's performance controlled by the self-optimization feedback loop. The other way around, an adaptation to optimize the performance might cause failures in the adaptable software. Such potential conflicts require the coordination of the feedback loops.

In EUREMA, the coordination of multiple feedback loops is explicitly modeled with FLDs. Such mega-model modules coordinate other modules that specify individual feedback loops by synchronizing their execution. In the following, we discuss two basic design alternatives for coordinating two feedback loops, in particular for self-repair (Figure 3.8) and self-optimization (Figure 4.1). Both alternatives are modeled with FLDs and they ensure the coordinated execution of the feedback loops either by sequencing complete feedback loops or individual adaptation activities of the different feedback loops.

Solution:

- Use independent triggers for both loops
- Sequential execution will ensure that loops do not overlap
4.2 Coordination of Multiple Feedback Loops

A simple way to coordinate two feedback loops is to execute them sequentially. This is specified in the FLD depicted in Figure 4.3, which uses complex model operations to synchronously invoke the individual feedback loops. Covering multiple capabilities, like self-repair and self-optimization, leads toward self-management, which motivated the name of the FLD. Explicitly coordinating multiple feedback loops facilitates the sharing of runtime models or even adaptation activities among the loops. As specified by the FLD depicted in Figure 4.3, both feedback loops use the same instances of the Architectural Model and TGG Rules. Moreover, they share the monitoring activity in certain situations where one feedback loop also performs the monitoring for the other loop as discussed in the following.

**Solution:**

- Extra module enforces the sequential execution such that the loops do not overlap.
Solution:
- Join monitor and execute activities
- Extra module enforces the sequential execution
EUREMA: Multiple Layers

Layer_0

Layer_1

Layer_2

Adaptable Software
**EUREMA: Reflection via the Megamodels**

**Benefits:**
- No extra model has to be developed
- Causal connection is guaranteed by construction

**Disadvantages:**
- No abstraction of the underlying layer
- No temporal decoupling as no copy is maintained
Complex Behavior of Self-Adaptation Activities

Fig. 5.5: Sequence diagram describing the logical behavior of layered feedback loops. Any adaptation performed by the higher-layer feedback loop is immediately enacted to the lower-layer feedback loop.
**EUREMA: User-Defined Reflection**

**Benefits:**
- Abstraction of the underlying layer
- Temporal decoupling

**Disadvantages:**
- Extra model has to be developed
- Causal connection has to be maintained explicitly
Co-Existence of Self-Adaptation & Maintenance

Development and Maintenance Environment

Adaptable Software

Evaluation Models
Change Models
Plan
Reflection Models
Monitor Execute
Monitoring Models Execution Models

Evaluation Models
Change Models
Plan
Reflection Models
Monitor Execute
Monitoring Models Execution Models

Giese | Dagstuhl 15041 | Model-driven algorithms and architectures for self-aware computing systems
Coordinated external **ad hoc adaptation** of the EUREMA model by adding a module on a higher level.

External Adaptation:
Add **self-adaptation layer** in an EUREMA model on the fly.

**External Adaptation:**

- **Layer-0**
  - :Adaptable Software

- **Layer-1**
  - :Self-repair

- **Layer-2**
  - :Self-repair-strategies
    - MAPE
    - feedbackLoopModel
    - ++
    - ++
    - After[Deep check for failures]: Adapt;
    - M..PE

**Self-repair-strategies**

- **<<EvaluationModel>>**
  - Repair strategies analysis rules

- **<<Monitor>>**
  - <<Analyze>>
    - Check success rate

- **<<Plan>>**
  - <<Execute>>
    - Synthesize new repair strategies

- **<<ReflectionModel>>**
  - feedbackLoopModel

- **Adapt**

- **Adapted**
- **Options:**
  - Only model native triggering with EUREMA (no evolution is possible later)
  - Model and realize triggering with EUREMA (evolution is possible later !)
The general separation of the adaptation behavior into runtime models and activities can be captured. Emulation would in addition permit evolution due to the co-existence with offline maintenance.
EUREMA: Discussion

- Model-driven engineering approach for adaptation engines
- Domain-specific modeling language for layers, modules, and control flow
- Leverages advanced solutions, like layered feedback loops
- Executable megamodels are kept alive at runtime
- Runtime models are employed at runtime
- Runtime interpreter for adaptation engines permits high degree of flexibility
- Leverages the co-existence of self-adaptation and off-line adaptation for evolution
- Modules and runtime models can to some extent be reused

Limitations:
- Concurrency and a distributed setting are not supported yet
- ...
3. Challenges Ahead - enable self-aware computing

- Capture goals explicitly
- Guide tradeoffs
- ...

- Capture possible changes
- ...

- Predictions
- ...

- Learning model from observations
- Abstract from details
- ...

- Selection
- Synthesis
- ...

- Enact runtime model for the adaptable software
- Add details
- ...

REUSE!

Loop:
- Speed vs. accuracy
- exploration vs. exploitation
- limits/guarantees for model learning
- proper treatment of uncertainty

Giese | Dagstuhl 15041 | Model-driven algorithms and architectures for self-aware computing systems
Challenges Ahead
- long term (meta self-aware?)

- Executable mega models kept alive at runtime with
  - Multiple runtime models
  - Activities are model operations (e.g., TGG)
  - Multiple loops
  - Multiple layers
  - Runtime interpreter for adaptation engines permits high degree of flexibility
- Leverages the co-existence of self-adaptation and evolution
- Modules and runtime models can to some extent be reused

Coordinate multiple loops for:
- Multiple goals
- Multiple runtime models
- Multiple activities (runtime model operations)
  - Learning strategies
  - Prediction strategies
  - Synthesis strategies
- ...

Higher layer must steer:
- diverse runtime models: number + selection
- diverse activities (runtime model operations): number + selection
- speed / accuracy
- exploration / exploitation
- ...

Giese | Dagstuhl 15041 | Model-driven algorithms and architectures for self-aware computing systems
4. Outlook: Beyond centralized MAPE-K ...

(Linked) Cyber-Physical Systems

Smart Factory - E.g. Industry 4.0

Smart Logistic

Micro Grids

Internet of Things

Smart City

System of Systems

Ultra-Large-Scale Systems

Smart Home

E-Health

Ambient Assisted Living

Collaborative self-aware computing

- Exchange runtime models