Outline

I  Motivation
   - Why self-adaptiveness?

II  Foundations
   - What is self-adaptiveness?

III  Construction
   - How to build them?

IV  Quality Assurance
   - How to ensure their quality?

V  Conclusion
Motivation: Software Evolution & Aging

Software Evolution [Lehman&Belady1985,Lehman1997]:
- Programs always include explicit and implicit assumptions about the real world domain
- The real world domain and the program (and its explicit and implicit assumptions) must be maintained compatible and valid with one another
- Developing software is a complex feedback system

Two types of software aging [Parnas1994]:
- Lack of Movement: Aging caused by the failure of the product’s owners to modify it to meet changing needs.
- Ignorant Surgery: Aging caused as a result of changes that are made.
- This “one-two punch” can lead to rapid decline in the value of a software product.
Motivation: Software Complexity & Administration

Autonomic Computing [AC2001]:

- Evolution via automation also produces complexity as an unavoidable byproduct (especially true for IT systems: incredible progress in speed, storage and communication; extreme growth software with >30 million loc and > 4,000 programmers)

- In fact, the growing complexity of the I/T infrastructure threatens to undermine the very benefits information technology aims to provide, because systems cannot be managed any more.

Proposed solution:

- make things simpler for administrators and users of I/T by automating its management (Paradoxically, it seems we need to create even more complex systems).

- Inspiration is the massively complex systems of the human body: the autonomic nervous system.
I Motivation: Software Landscapes vs. Applications

- Characteristics: large-scale, heterogeneous, distributed, ad hoc evolution, no central authority
- May include: Server backends, embedded subsystems, wireless ad hoc networks, mobile devices, ...

The software must resolve adaption needs due to changes in the context and platform itself to be able to work at all
Motivation: Future Software Landscape Example

A shuttle system that builds convoys to optimize the energy consumption

http://www.railcab.de/
II Foundations: Self-Adaptiveness

What do we need?

Software (Model) runs on Platform

Environment

Original

abstract

Who?

The software itself!
II Foundations: Adaptation Loop

Environment sensors
Application requirements
Instrumentation
User context

Collect

Inform users and administrators
Alter system configuration
Record strategies
Manipulate effectors

Act

Hypothesis generation
Planning
Risk analysis
Decision theory

Decide

Game theory
Regulations, rules, and policies
Bounds and envelopes
Symptoms database
Inference
Uncertainty reasoning
Economic models

Analyze

[Brun+2009]
II Foundations: Architecture & Self

Internal Approach

Software

Context

External Approach

Adaption Engine

Function

Context

[Salehie&Tahvildari2009]
II Foundations: Models and Adaptation

Adapt “without” models:
- Still explicit or implicit **design-time models** are used to guide adaptation processes
- **Limitation:** covers only changes covered by one model of the software’ + context (potentially including some parameters or structural changes that can be observed)

Adapt with runtime models:
- Explicit **runtime models** are used to guide adaptation processes
- **Limitation:** covers only changes captured by the runtime models (multiple!); requires correct adjustment of them from the observations
II Foundations: Top-Down Architecture

Reference Architecture for Self-Management:

- Layers for different purposes
- Decoupling of the layers in time

[Kramer&Magee2007]
Self-organization is a process in which structure and functionality at the global level of a system emerge solely from numerous interactions among the lower-level components.

Characteristics:
- No central control
- Emerging structure
- Resulting complexity
- High scalability

Emergence is an apparently meaningful collaboration of components (individuals) resulting in capabilities of the overall system (far) beyond the capabilities of the single components.
But how can we systematically build the software for such systems (construction)?
III Construction: Control Loop

Development time:

Runtime:

Challenges:

(1) How to design the adaptation algo.?

(2) How to architect systems with control loops?

(3) How to develop the necessary elements of the loop?
Complex development time models:

- Application: Monitoring and Restart of Services
- Instance of the MIAC scheme
- Identification of required reliability and availability parameters via monitoring
- An development time **availability model** in form of an Stochastic Petri Net is used to **precompute** values for the required parameter adaptation (using interpolation)
III Construction: (2) Architecting Loops

Problem [Brun+2009]:
- Control loops are not directly supported when architecting

Proposal: A UML Profile for feedback loops:
- Identify loop elements and mark them using stereotypes
- Identify whether loops overlap in undesired ways
- Identify control related effects
III Construction: Control Loop & Layers

Development time:

- Goal Management
- Change Management
- Function
- Context

Runtime:

- Adaption:
  - Goal Mgt.
  - Change Mgt.
- Function
- Context

Challenges:

(1) Support layers
(2) Provide decoupling between layers

16.02.2010 | Holger Giese | Model-Driven Software Engineering of Self-Adaptive Systems
III Construction: Micro Architecture

Operator-Controller Module

- **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
III Construction: Relation to the Reference Architecture

(1) Support layers
(2) Provides decoupling between layers
III Construction: Control Loop & Architecture

Relevant cases:
(1) Hierarchies
(2) Self-organizing
III Construction: Complex 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
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
Apply Graph Transformation Systems

- Map the tracks
- Map the shuttles
- Map the shuttle movement to rules (movement equals reconfiguration)

Rule:

\[ \text{t1:Track} \rightarrow \text{t2:Track} \]

\[ \text{s1:Shuttle} \rightarrow \text{t1:Track} \]
### III Construction:

(2) Self-organizing

**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

**Self-organizing** (degrees of freedom for the local rule-based configuration)

Rule-based configuration

distributed over the patterns and the components realizing the pattern roles
III Construction: Runtime Models

Development time:

Runtime:

Challenge:

(1) Efficient and cost-effective realization of the runtime models
(2) Efficient and cost-effective realization of the function update ($f$)
III Construction: (1) Runtime Models: MDE

- Supports **adaptation loops** for models using “meta-models” (EMF) and bidirectional model transformation techniques (Triple Graph Grammars) for an EJB application server.

- Extract abstract runtime models for different autonomic managers for **monitoring** EJB applications (**unchanged**).

- **Adapting** managed subsystem via extracted runtime models (parameter and structural adaptation; not as easy as monitoring!)

- Synchronize runtime models **incrementally** (faster as non incremental manual implementations).

[ICAC2009]
III Construction: (2) Function Update (Distributed)

- 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

But how can we guarantee that they have a sufficient quality (quality assurance)?
IV Quality Assurance: Development time models

**Bottom line:** Self-adaptive systems must simply be “better” and not “worse”

(1) Correct working adaptation algorithm
(2) Correct adaptation implementation
   a. Correct monitoring: handle measurement failures; ...
   b. Correct system analysis: consistent with real changes; ...
   c. Correct adaptation decisions: fits to real changes; guarantees required properties; ...
   d. Correct execution of the adaptation: consistent update; timing, ...
IV Quality Assurance: Control Loop & Layers

- Correct working adaptation algorithm (1) ⇒ if simple properties, abstract models can be formally verified
- Correct adaptation implementation (2) ⇒ Can be tackled to some extend if we abstract from adaptation details (consider only change management)
IV Quality Assurance: Correct (1) + (2)

Operator-Controller Module (OCM) for
- Cognitive operator (CO)
- Reflective operator (RO)
- Controller (C)

Formal verification ("RO part" only):
- Formal model covers possible pre-planned configuration steps
- Only consistent and steps of the controller that the reflective operator can do within required time bounds occur (correct (1))

Code generation:
- guarantees functional and timing properties (correct (2))
Correct working adaptation algorithm (1) $\Rightarrow$ simple properties for abstract models can be formally verified, if we abstract from adaptation details (consider only change management) and decomposing the problem (apply a modular or compositional reasoning schemes)
Decompose verification:
- Verification guarantees properties for the collaborations
- Verification guarantees conformance for components (ports refine roles)

Compositional result: Properties hold for all collaborations in correctly composed component deployments

But, it is yet not guaranteed that shuttles nearby are connected via a collaboration!
IV Quality Assurance:
(1) Correct adaptation algo. (2/3)

Forbidden Graph

Correctness: all reachable system graphs do not match the forbidden graph pattern

Problems:
- there could be infinite many reachable system graphs
- fixed initial topology not known (may change)

Now, both results together would guarantee the absence of collisions!

[Monterey2007]
IV Quality Assurance:
(1) Correct adaptation algo. (3/3)

Verification:
- Analyze whether structural changes can lead from safe to unsafe situations (inductive invariants)

Checking Options:
- Model Checking (mapping to GROOVE; only debugging)
  - Limited to small configurations and finite models
  - Extension for continuous time have been developed
- Invariant Checker (our own development)
  - Supports infinite many start configurations specified only by their structural properties
  - Supports infinite state models
  - Extension of time and discrete variables exist
  - Incremental check for changed rules
  - Extension of hybrid behavior (recently!)

[ICSE2006, ISORC2008]
Guarantee correct working adaptation algorithm (1)?

- If solver for $f$ exists, correctness can be derived

- Correct adaptation implementation (2) ⇒ Can be tackled by MDE
IV Quality Assurance:
(1) Correct adaptation algo.

- 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
- Trajectory synthesis establishes required guarantees for $\mathbf{f}$
- Backup for the case of data errors!
Correct adaptation implementation (2):

a. Correct system model updates: **valid abstraction by construction**

b. Correct system model analysis

c. Correct adaptation decisions

d. Correct execution of the adaptation (special case: propagate changes in updates system model): **functional correctness by construction; timing?**
V Conclusion & Outlook

- **Self-adaptive systems** promises to automate the efforts required today to adapt the software (by the developer and admin) as well as enables software landscapes not feasible without. However, it also makes the software even more complex.

- Therefore, techniques for the systematic and cost-effective **software engineering of self-adaptive systems** are crucial for the while vision.
  - **Construction** (adaptation algo., loops, layers, hierarchies, self-organizing, runtime models, function updates, ...)
  - **Quality assurance** (adaptation algo., loops, layers, hierarchies, self-organizing, runtime models, function updates, ...)
Models and model-driven engineering can play a major role for the cost-effective construction and quality assurance of such systems.

- Development time models permit to construct such systems and verify the correctness of the adaptation algo.
- In case of runtime models, suitable function updates can be constructed and verified to show the correctness of the adaptation algo.
- Model-driven engineering can often assure via code generation that the verified properties also hold for the running system.
- In case of runtime models, model-driven engineering can in addition be employed to provide a basis for structural adaptation that guarantees correct implementation.

But much left to be done ...
Invitation to Participate:

SEAMS 2011

6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems

Submission deadline: 12th Dec
Author notification: 15th Feb
Camera ready copy: 1st March

Sponsored by ACM SIGSOFT and IEEE TCSE
Waikiki, Honolulu, Hawaii, USA
May 23-24, 2011

at ICSE 2011
SEAMS 2011
See you in Hawaii
References (1/2)


References (2/2)


Own References


Additional Slides
I Motivation: Future Software Landscapes

**Prognoses:**
- “In the near future, software-intensive systems will exhibit **adaptive** and **anticipatory behavior**; they will process knowledge and not only data, and **change their structure dynamically**. Software-intensive systems will act as global computers in **highly dynamic environments** and will be based on and **integrated** with service-oriented and pervasive computing.”
  

- “The sheer scale of ULS systems will change everything. ULS systems will necessarily be **decentralized** in a variety of ways, developed and used by a wide variety of stakeholders with conflicting needs, **evolving continuously**, and constructed from heterogeneous parts.

- **Adaptation** is needed to compensate for changes in the mission requirements (...) and operating environments (..)

The Basic Case: Engineering & Design Models

Question: How do engineers develop complex systems?

Solution: design models
- used as representations for real or imaginary systems
- Allow to try out alternatives
- Allow reliable predictions

Characteristics of design models
- Complete coverage of the problem
- Accurate representation
- Constant