Christian-Albrechts-Universität zu Kiel

# Lecture
# „Intelligent Systems“

## Chapter 2: Intelligent Systems

Prof. Dr.-Ing. habil. Sven Tomforde / Intelligent Systems

Winter term 2020/2021

# About this chapter

Christian-Albrechts-Universität zu Kiel

## Contents

- Motivation
- Examples for the growing complexity
- Organic Computing: Nature as inspiration
- Design of Organic Computing systems
- Example: Organic Traffic Control
- Autonomic Computing
- A reference architecture
- Holonic systems
- Conclusion and references

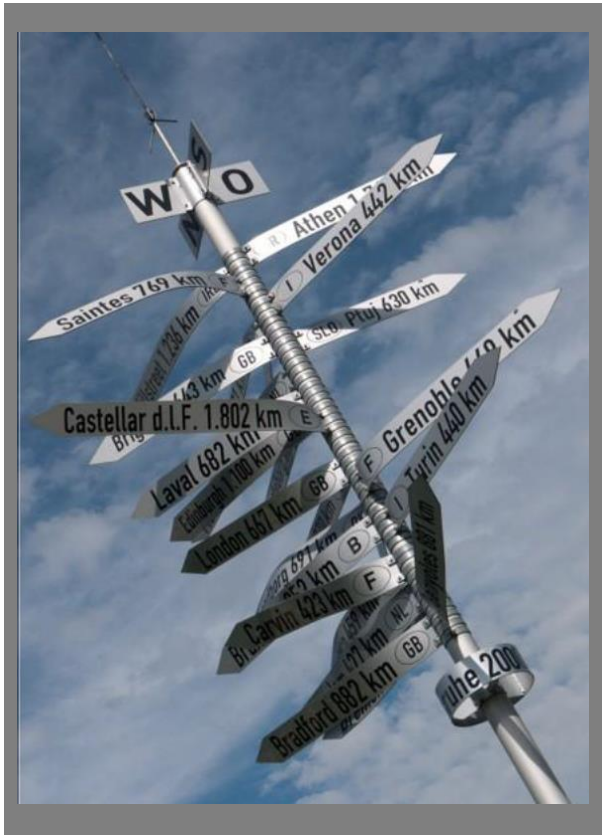
## Goals

- Be able to explain complexity as a most urgent challenge
- Be able to define and compare the terms ‘Organic Computing‘ and ‘Autonomic Computing‘
- Be able to explain the process of an intelligent system using the example of traffic control
- Be able to depict and explain a reference architecture for self-adapting systems with an emphasis on different challenges
- Be able to define the concept of holonic systems

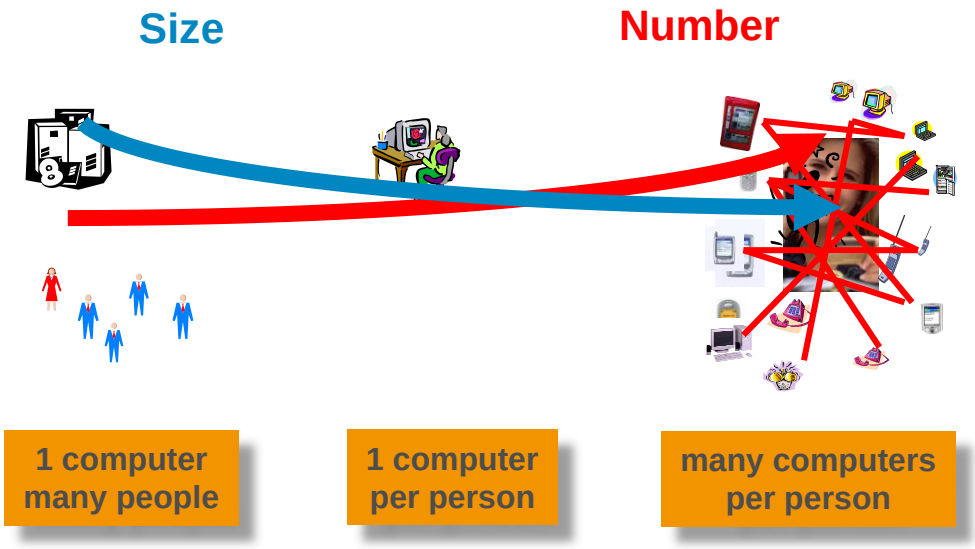
# Agenda

Christian-Albrechts-Universität zu Kiel

- Motivation
- Examples for the growing complexity
- Organic Computing: Nature as inspiration
- Design of Organic Computing systems
- Example: Organic Traffic Control
- Autonomic Computing
- A reference architecture
- Holonic systems
- Conclusion and references



- Computing trends
  - Number of devices
    → increases
  - Computational power
    → increases
  - Malfunctions due to mutual influences




- Observations
  - Failures due to high complexity
    → Manageability of interconnected systems decreases
  - Unknown configuration space (dynamic environment)
    → Impossible to predict all possible situations

# Traditional system design



- Traditional Systems Engineering
  - Relies (mainly) on hierarchical top-down design methods.
  - Everything should be pre-planned and tested at design-time.

- Intelligent Systems
  - Complexity of system tasks is conquered by a distribution of responsibilities.
  - Engineer at design-time is responsible for the basic system design and defining the scope of the system‘s freedom.
  - System becomes self-managing using automated discovery of appropriate decisions.

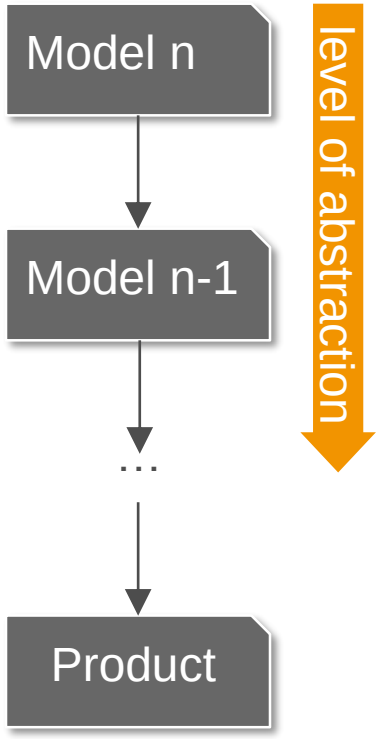
**Mastering of complexity in the presence of permanent change!**

# Traditional system design (2)



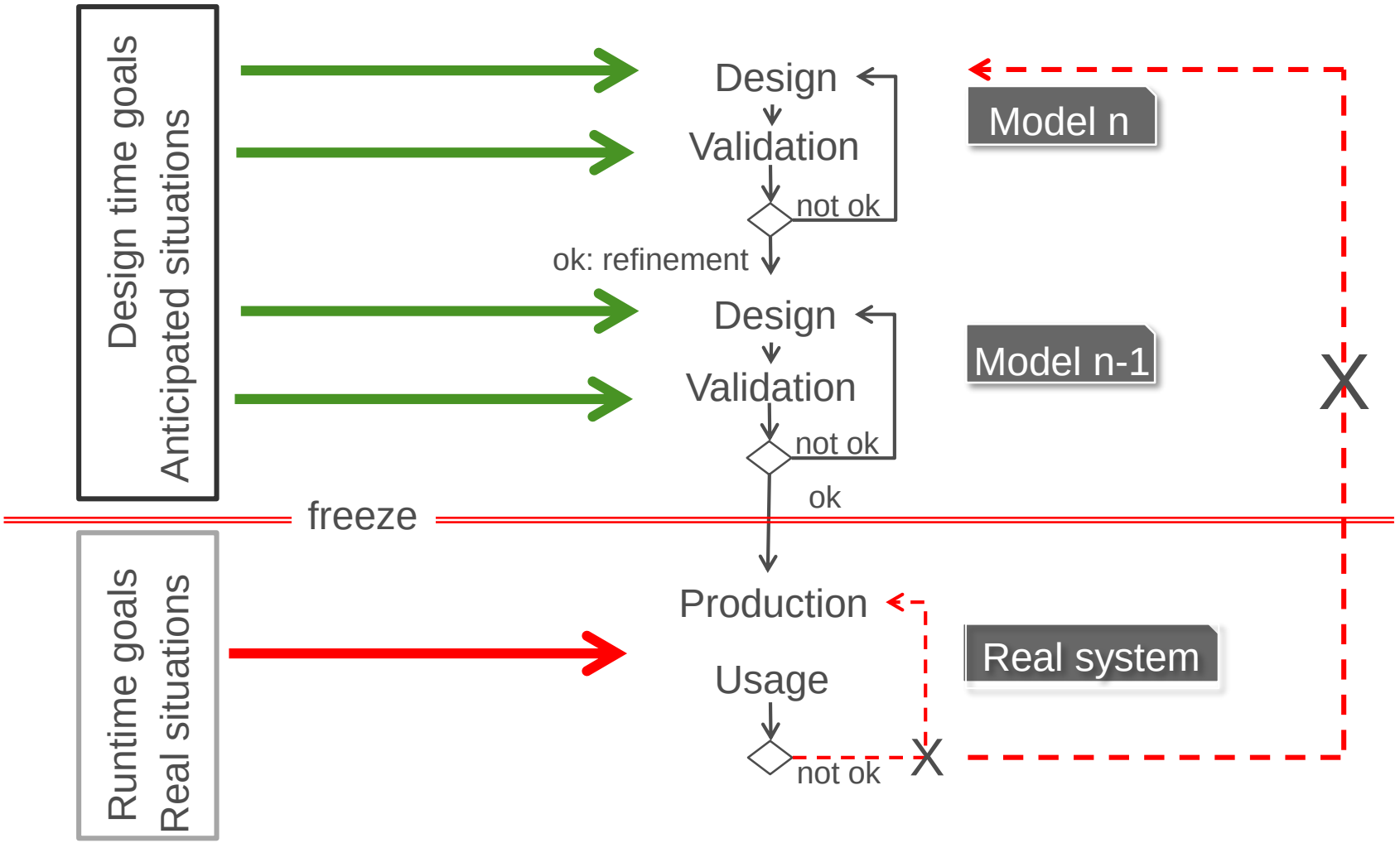
- Traditional Systems Engineering
  - Specifies requirements
  - Modelling at an abstract level
  - Refine at a more detailed level
  - Until the product (system) is finished
  - Test with all foreseen situations

- Traditional design processes
  - Pre-planned and fully tested solutions at design-time
  - No revision of design decisions at runtime



# Traditional design process

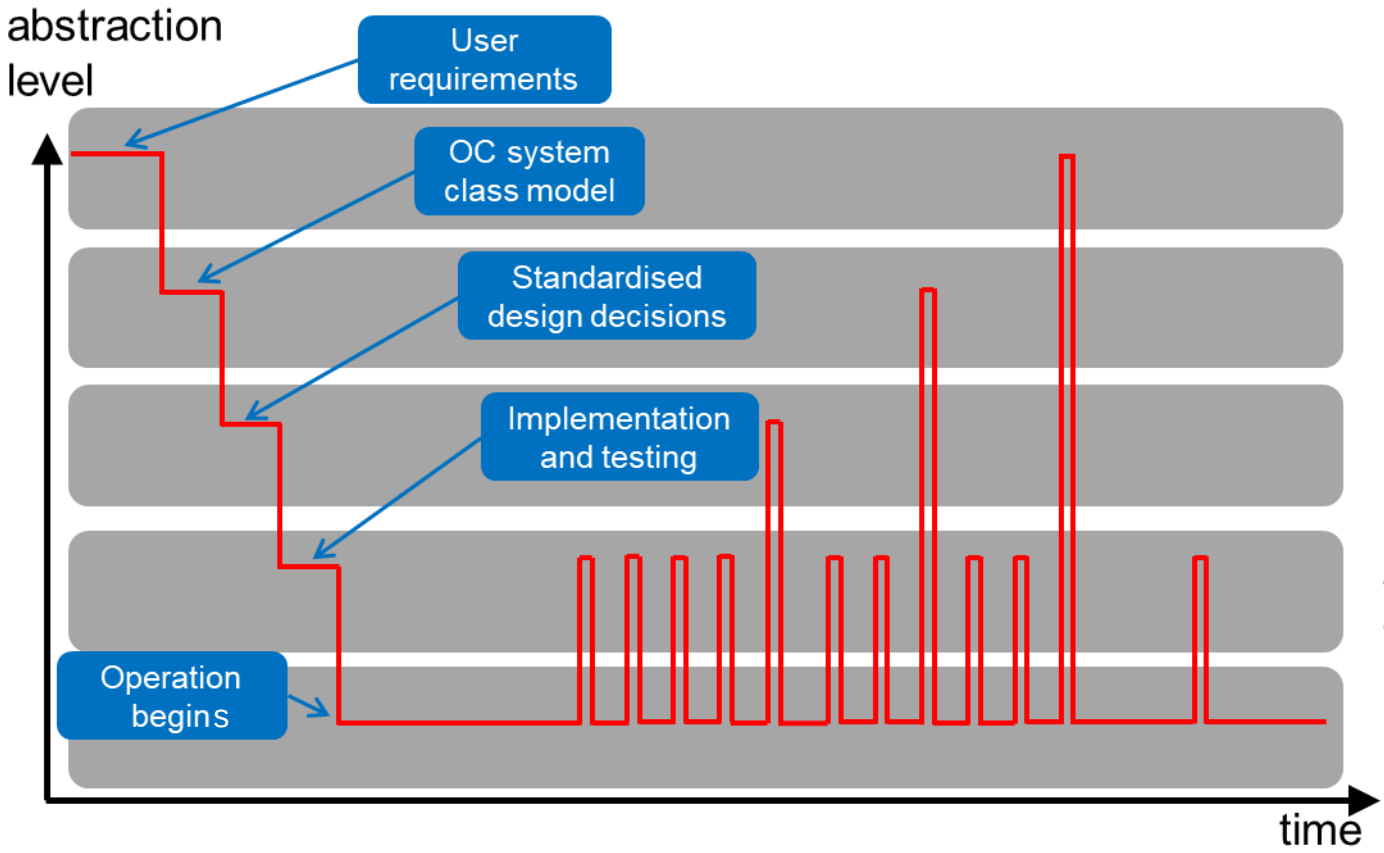
Christian-Albrechts-Universität zu Kiel



## From traditional to intelligent systems

- Intelligent systems consist of autonomous sub-systems:
  - The sub-systems possess sensors and actuators.
  - Sub-systems interact with each other and the environment.
  - No global (or: system-wide) control necessary.
- The resulting in interconnected intelligent systems have to:
  - organise themselves,
  - be adaptive and flexible, and
  - learn the optimal behaviour – to be able to adapt themselves autonomously to previously unknown situations.
- Most of the decisions are taken based on local knowledge and without user / system-wide influence!

## Goal: Automated Jo-Jo design




5) Requirements level

4) Design level

3) Customisation level

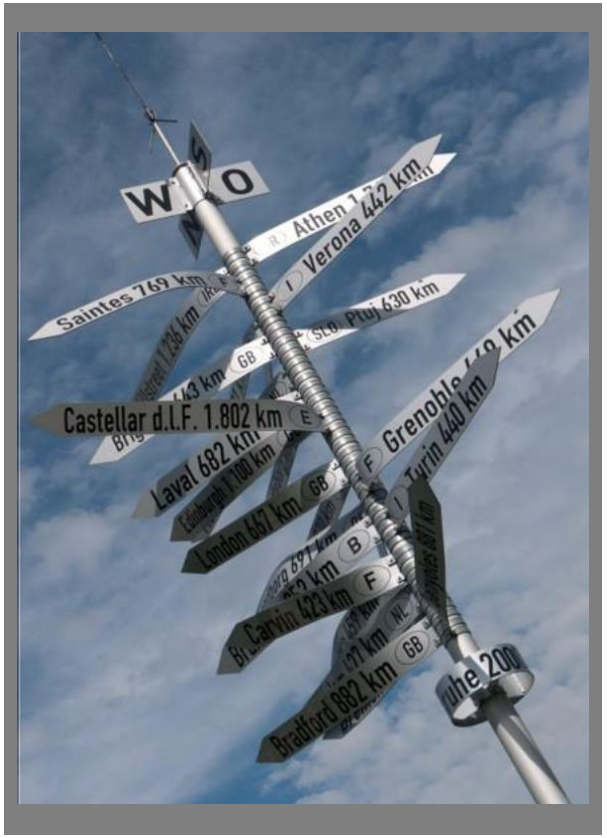
2) Test and configuration level

1) Usage level

# Agenda

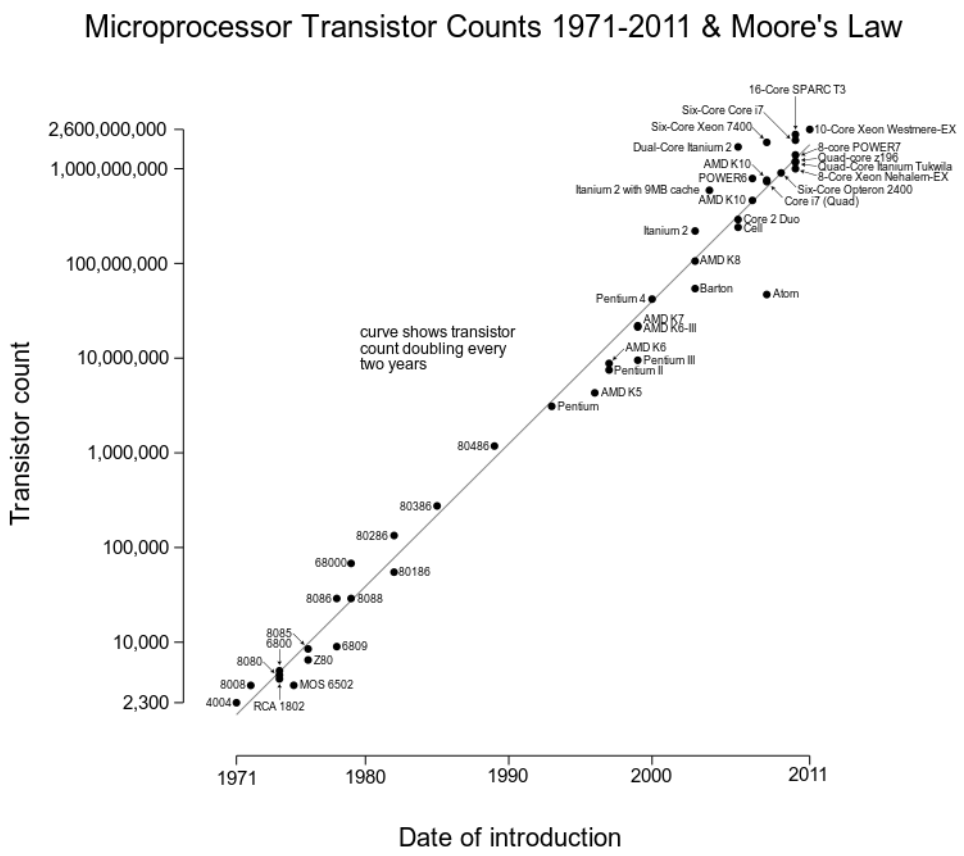


- Motivation
- Examples for the growing complexity
- Organic Computing: Nature as inspiration
- Design of Organic Computing systems
- Example: Organic Traffic Control
- Autonomic Computing
- A reference architecture
- Holonic systems
- Conclusion and references

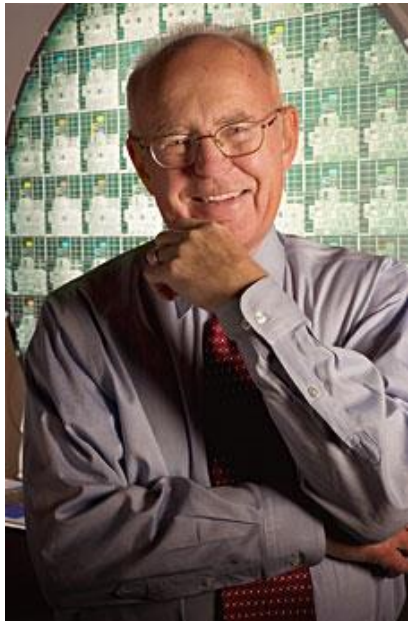


# Example: Moore‘s law



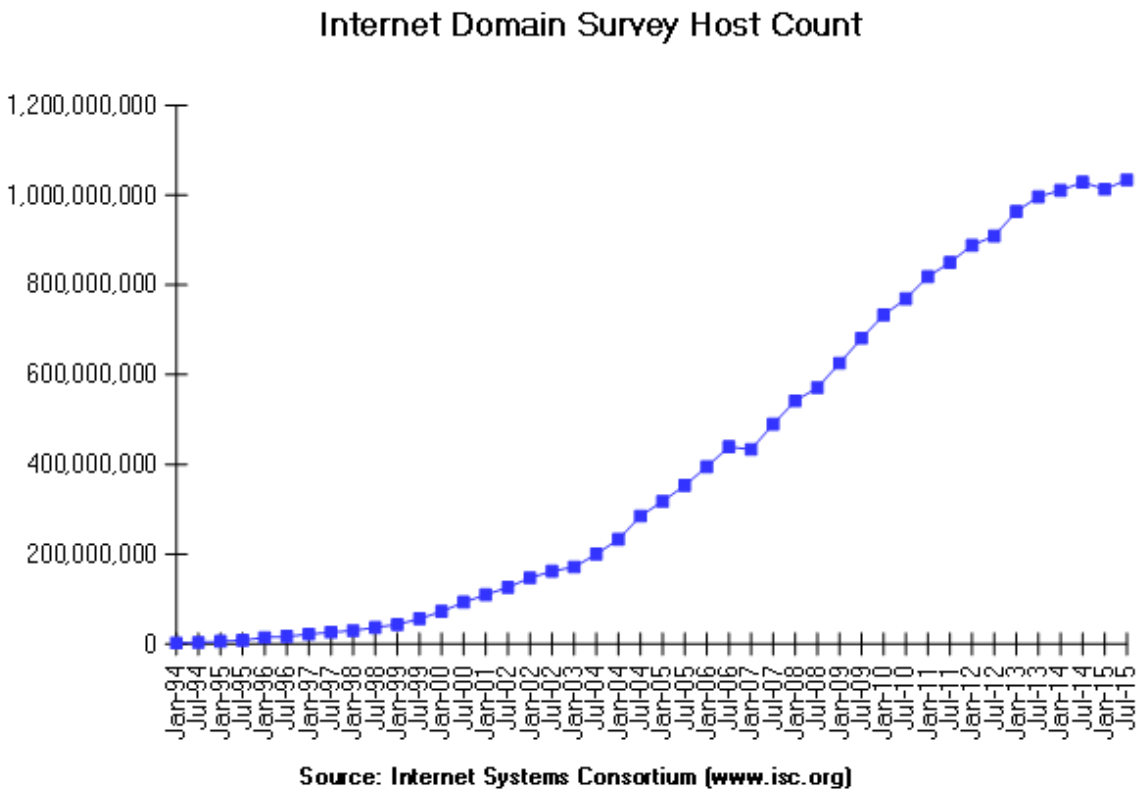



Gordon Moore in 1965:
„The complexity of integrated circuits (IC) with minimal component cost is roughly doubled every two years!“



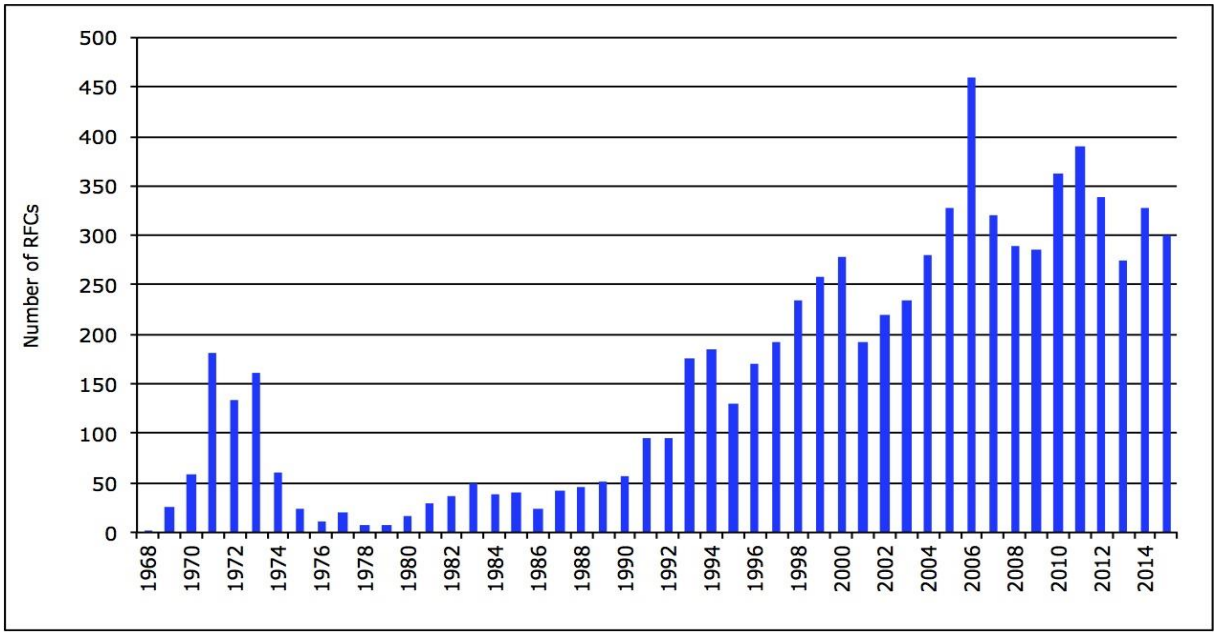
Source: intel.com

Development of the number of hosts reachable via IP address:



# Example: Requests for Comments






- The Requests for Comments (RFC) are a series of technical and organisational documents for the Internet (originally: Arpanet) that has been launched on April 7th, 1969.
- All basic building blocks and standards of Internet technology initially started as RFC, including Email, IP, URL, or calendar formats.

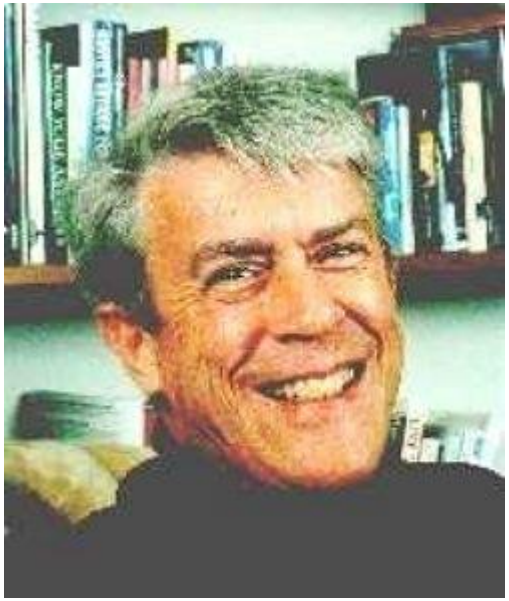
In software engineering:

Glass states that “IT complexity is indirectly related to functionality, in that a 25% increase in functionality increases complexity by 100%”.

- For every 25% increase in the business functionality in service, there is a 100% increase in the complexity of that service.
- For every 25% increase in the number of connections in service, there is a 100% increase in the complexity of that service.

## Robert L. Glass

American software engineer and writer



Source: amazon.com

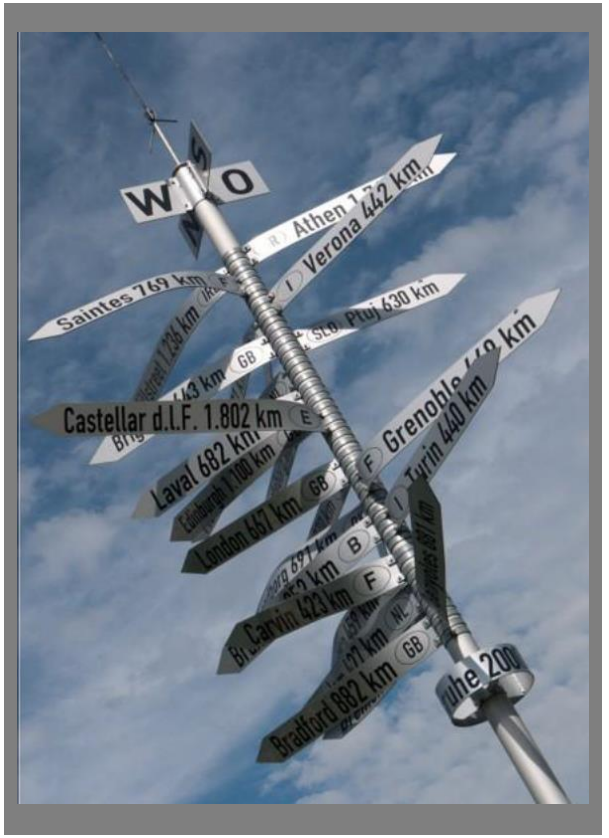
Observation:

- Hard to administrate and to master interconnected systems
- Outages and failures
- Unforeseen influences
- Ranging from non-optimal to even malfunctioning
- Complexity is everywhere!

# Agenda

Christian-Albrechts-Universität zu Kiel

- Motivation
- Examples for the growing complexity
- Organic Computing: Nature as inspiration
- Design of Organic Computing systems
- Example: Organic Traffic Control
- Autonomic Computing
- A reference architecture
- Holonic systems
- Conclusion and references



# *Mastering complexity: Nature as inspiration*

Christian-Albrechts-Universität zu Kiel

## Complexity

- Complexity is a common phenomenon in nature!
- A variety of well-adapted solutions can be observed.
- Common theme: reduction of complexity by collaboration of autonomous entities.


## Natural systems …

- … are typically highly complex systems themselves.
- … have evolved over billions of years.
- … show self-* properties.
- … are changeable and flexible, robust against disturbances, resilient, optimised.

## How is complexity mastered in nature?

- System consists of potentially large collections of individuals.
- The individuals act autonomously without central control.
- Each individual is characterised by self-* properties: self-learning, self-adaptation, self-protection, and so on.
- Decisions are taken by autonomous entities based on local knowledge.
- Entities interact and cooperate, resulting in a macro-level behaviour of the entire system.
- Some system properties appear as an emergent effect.
- Each individual is self-motivated, i.e. it follows its own goals (these goals do not necessarily have to be in line with the entire system).

# Nature as inspiration (2)

Nature as inspiration
- Not: imitation of natural systems
- But: transfer of basic mechanisms

Nature as inspiration

- Mechanisms such as stigmergy or emergence to organise the system → Cooperation and collaboration
- Separation of concerns among autonomous entities → Self-organisation
- Each autonomous entity:
  - Has self-awareness and is situated in an environment → Modelling of conditions; Time series processing / analysis
  - Finds approximately good solutions / behaviour even in unknown conditions → Transfer of knowledge; Anomaly / novelty detection
  - Continuously improves its behaviour by learning and (to a certain degree) reflection → Learning from feedback; System evolution

Organic Computing

"What is an Organic Computing System?"

A computer system that:

- is nature-inspired,
- consists of several cooperating autonomous subsystems that
- achieve a certain performance even
  - in time-variant environments and
  - in disturbed situations,
- which is self-adapting and
- improves its behaviour through experiences.

A paradigm shift in system engineering


- Goal:
  - Move traditional design-time decisions to runtime.
  - From engineers into the responsibility of systems themselves.
- Build collections of smaller systems instead of monolithic structures.
- Allow for autonomous decisions.
- Base these decisions on local knowledge without having access to a global world model.
- Let the distributed entities interact and cooperate.
- Establish a feedback mechanism for each entity: learn from past actions.

## A brief definition of Organic Computing

- Organic systems consist of autonomous sub-systems:
  - These sub-systems possess sensors and actuators.
  - Sub-systems interact with each other and the environment.
  - No global (or: system-wide) control necessary.
- The resulting interconnected OC systems have to:
  - organise themselves,
  - be adaptive and flexible, and
  - learn the optimal behaviour – to be able to adapt themselves autonomously to previously unknown situations (more robust!).
- Most of the decisions are taken based on local knowledge and without user / system-wide influence.

**OC means to move design-time decisions to runtime!**

## Intelligent Systems and Organic Computing

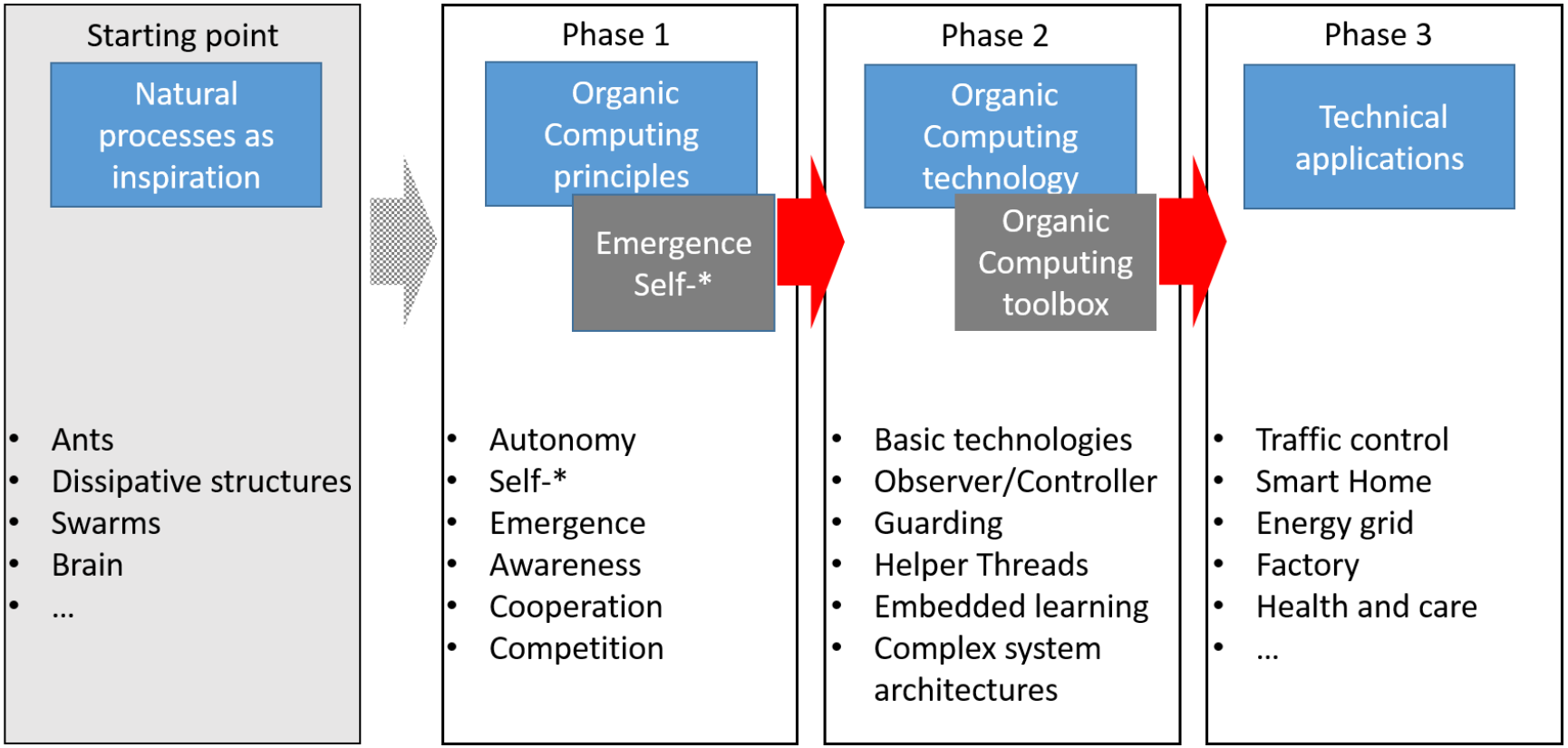
- OC is used as a synonym for “engineering intelligent systems“ in the context of this lecture
- There are several initiatives with the same / a similar motivation or scope, e.g.:
  - Autonomic Computing
  - Proactive Computing
  - Complex Adaptive Systems
  - Self-aware Systems
  - (Multi-Agent Systems)
  - …
- They all have in common that an individual system perceives, analyses, and acts upon gathered information of the environment by making use of machine learning techniques.

# A brief history

Christian-Albrechts-Universität zu Kiel

- 1999: von der Malsburg coined the term “Organic Computing” as a combination of neurosciences, molecular biology, and software.
- 2000: Tennenhouse presented his vision of “Proactive Computing“
- 2002: Workshop on future topics in technical computer science
- 2002: Forrester Research study on “Organic IT“
- 2003: Autonomic Computing proposed by IBM
- 2003 and 2004: OC Philosophy Workshops at Kloster Irsee
- 2005: Special Priority Programme 1183 “Organic Computing“ funded by DFG: Schmeck (KIT), Müller-Schloer (Hannover), and Ungerer (Augsburg)
- 2006: First Dagstuhl Seminar on Organic Computing
- 2006: First professorship on Organic Computing in Hannover
- 2009: Research Unit “Trustworthy Organic Computing Systems“ funded by DFG
- 2010: Collaborative Research Centre “Invasive Computing“: Erlangen
- 2012: First full professorship on Organic Computing in Augsburg
- 2012: Establishment of “Special Interest Group“ within the GI

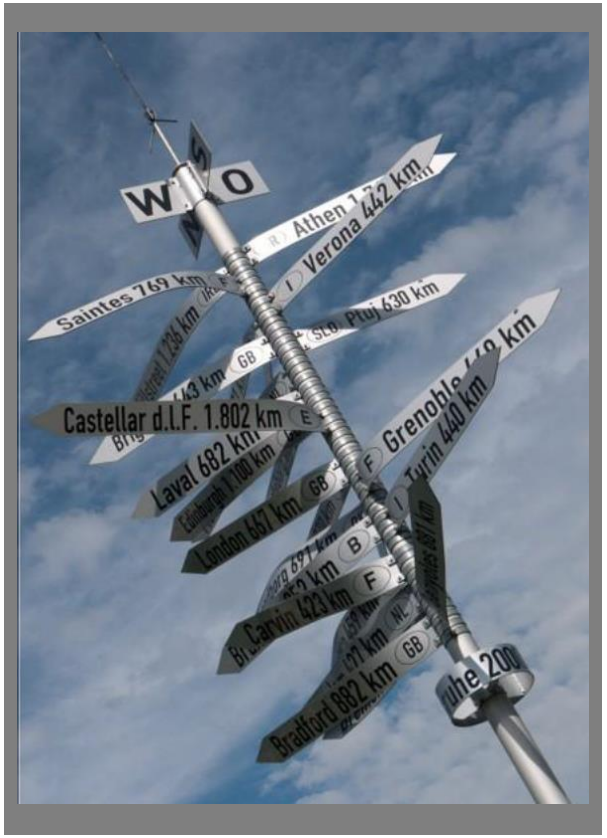
# Well, this has been the initial plan…



# Agenda

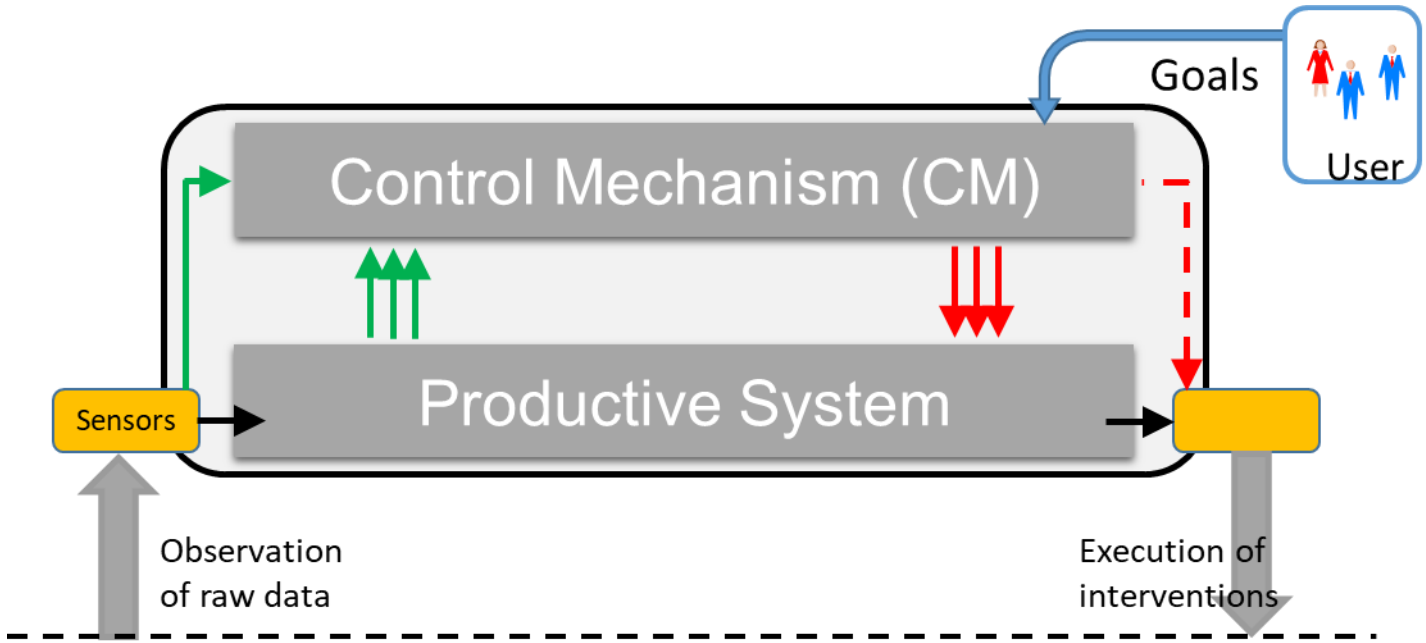


- Motivation
- Examples for the growing complexity
- Organic Computing: Nature as inspiration
- **Design of Organic Computing systems**
- Example: Organic Traffic Control
- Autonomic Computing
- A reference architecture
- Holonic systems
- Conclusion and references



# General concept of an intelligent system






Green: flow of observed data
Red: flow of control interventions

## System boundaries

- Everything not directly related to the system itself is modelled as being part of the external environment.
- Environment is accessed through sensors and manipulated by actuators.

## Take the human operator “out of the loop“

- Productive system is only influenced by the internal control mechanism.
- Theoretical possibility for the user to intervene directly with the output signal.
- User provides just a goal (or a set of goals) defining good or bad behaviour.
- CM figures out what to do in which situation without direct human intervention.
- Classic approach: the user specifies directly what the system has to do.

# General concept of an intelligent system (3)

Christian-Albrechts-Universität zu Kiel

## Requirements

- Access to sensor readings covering all important aspects.
- Access to internal status variables of the productive system.
- Utility function specifying good/bad behaviour.
- Access to control interfaces of the productive system.
- Control interface has to be directly related to performance.

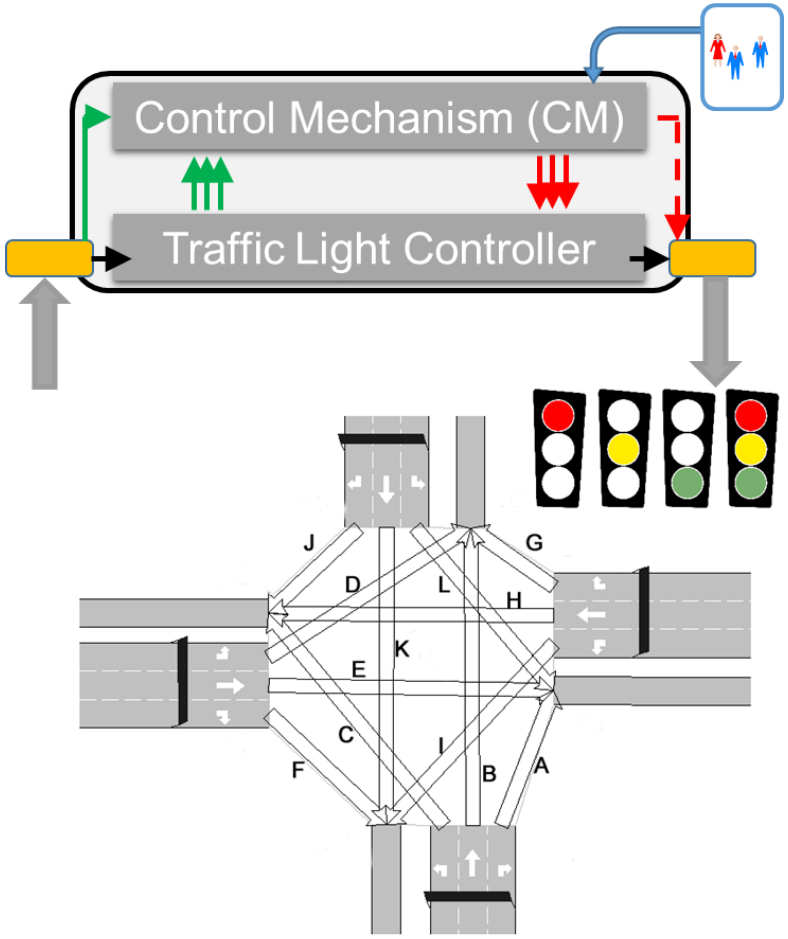
## Actions

- Manipulation of parameters to guide the behaviour of a productive system.
- Activation or deactivation of components and functionality.
- Exchange of algorithms or techniques in use.
- Selection of interaction partners.
- Modification of observation model (i.e. selection, resolution, and frequency of sensor information)

# General concept of an intelligent system (4)

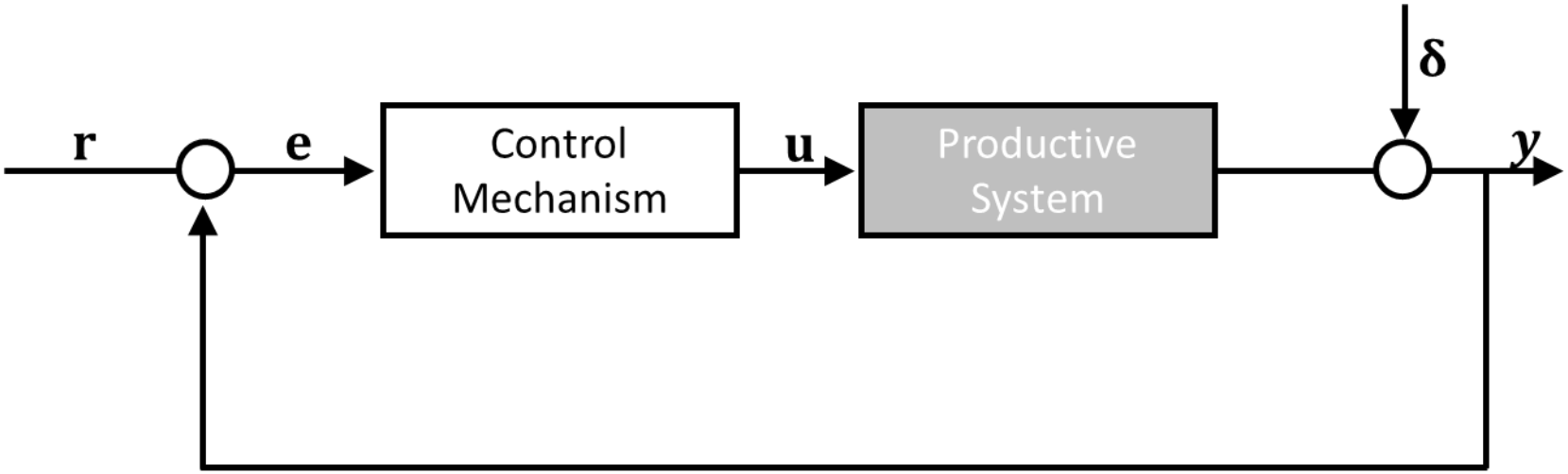
Example: Traffic control

- Productive system: traffic light controller
- Sensor readings: induction loop readings (occupied or not)
- Effectors: traffic signals (green/yellow/red)
- Input CM: vector with traffic flow per hour for each turning
- Output CM: green durations for each traffic light



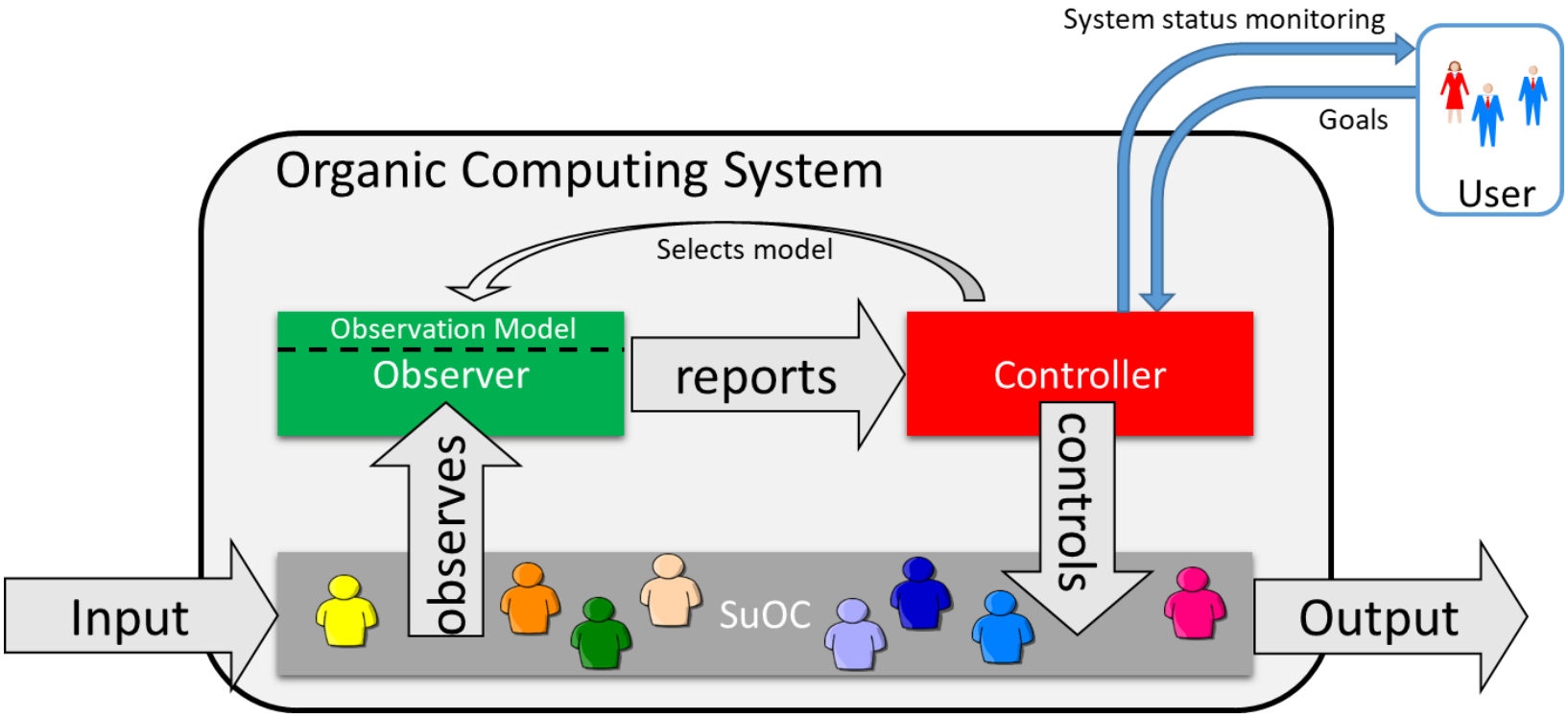
Control loop

- Actions have an impact on performance
- Performance is measured from the environment
- Environment is modified by actions
  → Feedback loop (see control theory)



# Generic Observer/Controller pattern





# Generic Observer/Controller pattern (2)



Three major components:

- System under Observation and Control (SuOC):
  - Productive part of the system
  - Functional even if higher-layered observer/controller components fail
  - Not restricted to individual devices
- Observer:
  - Gathers information
  - Analysis, predicts, detects patterns
  - Builds situation description
  - Based on the observation model
- Controller:
  - Decides about actions
  - Learns from the feedback
  - Modifies observation model

## Basic idea: Learning from feedback

- Rule-based system
- Rule:

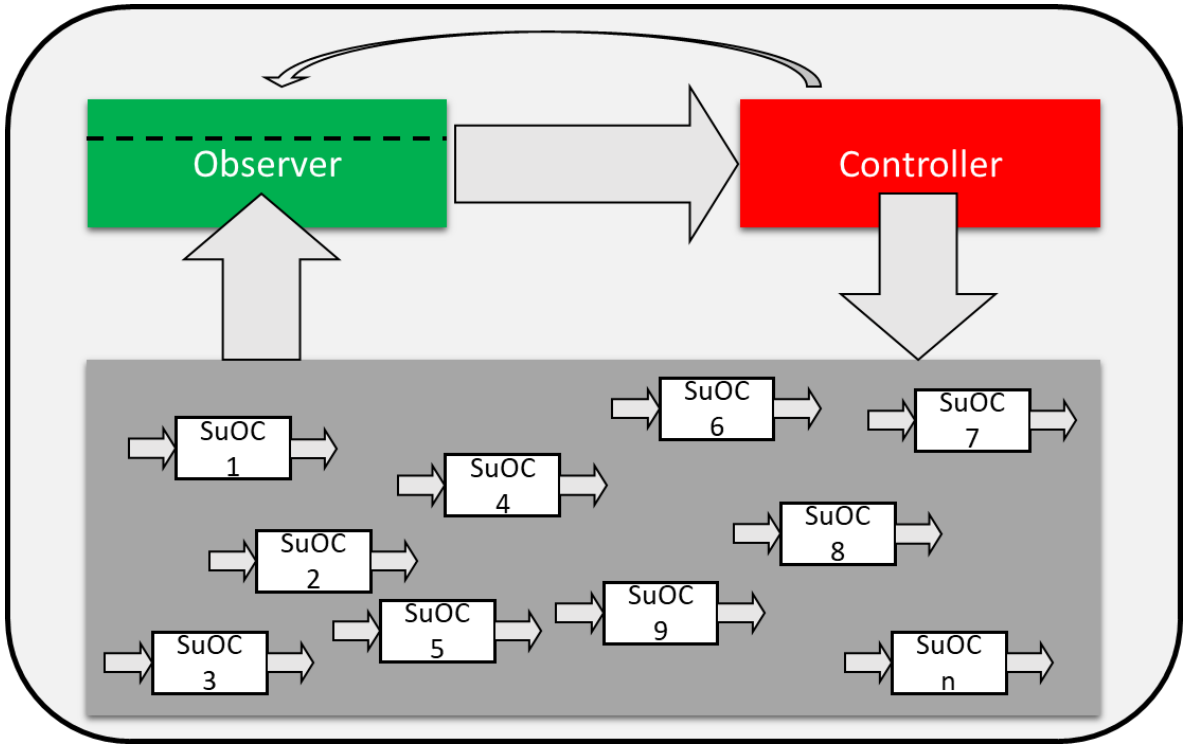
  [Condition] [Action] [Evaluation]
- Rules compete, more than one can fulfil the condition
- Evaluation criteria used to decide which to take, updated in each cycle

We'll discuss feedback-based in learning in detail in the last chapters

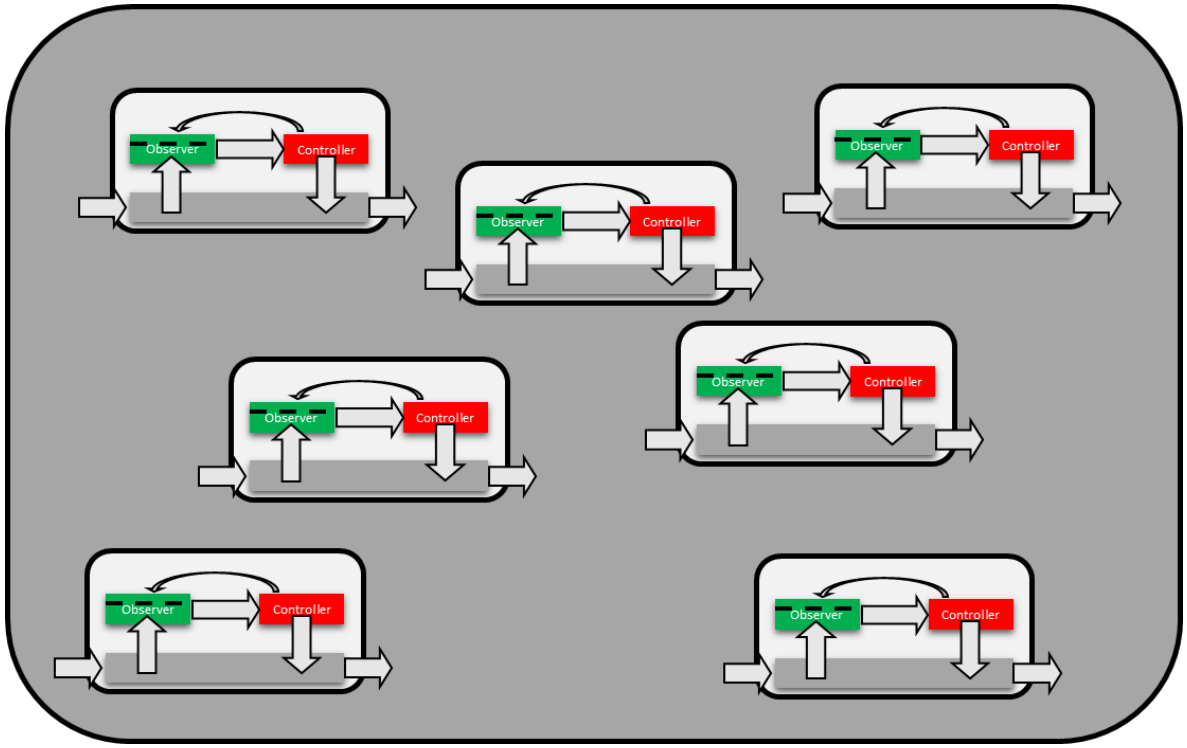
## Traffic example

- Condition: vector with intervals for traffic flows per hour per turning
- Action: Green durations per traffic light
- Evaluation: Averaged delays occurring at the intersection

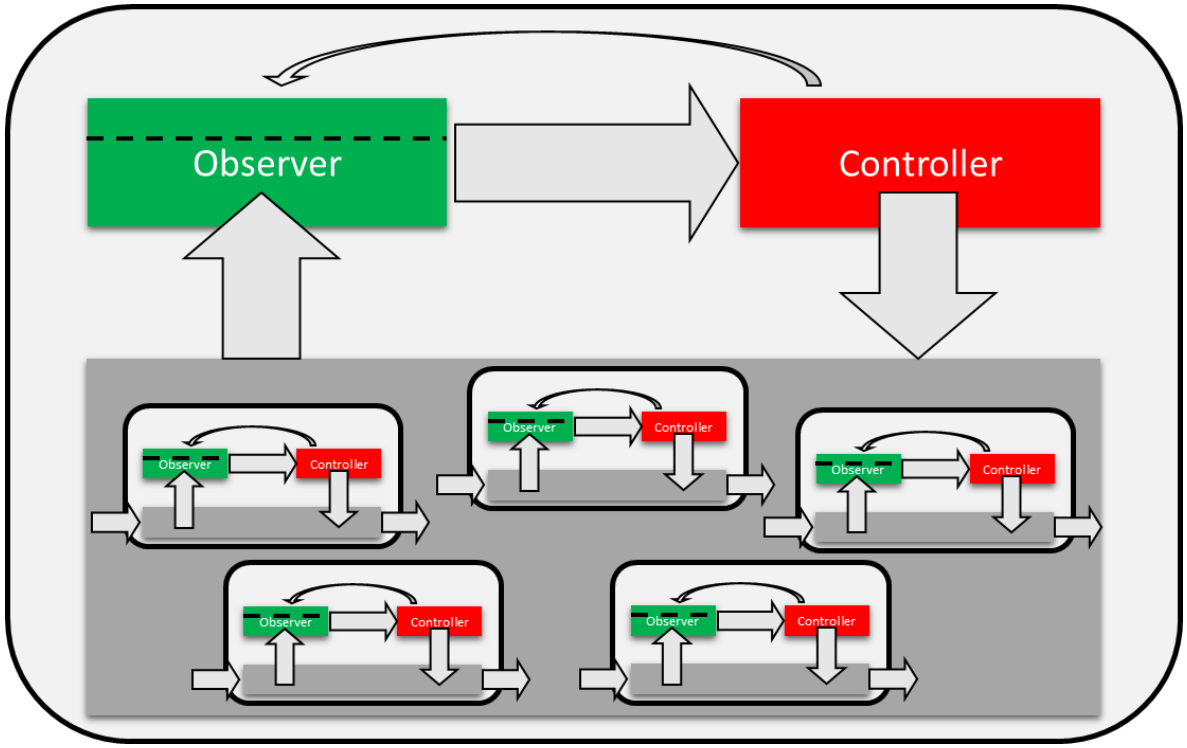
# Variant A: Fully centralised



# Variant B: Fully decentralised

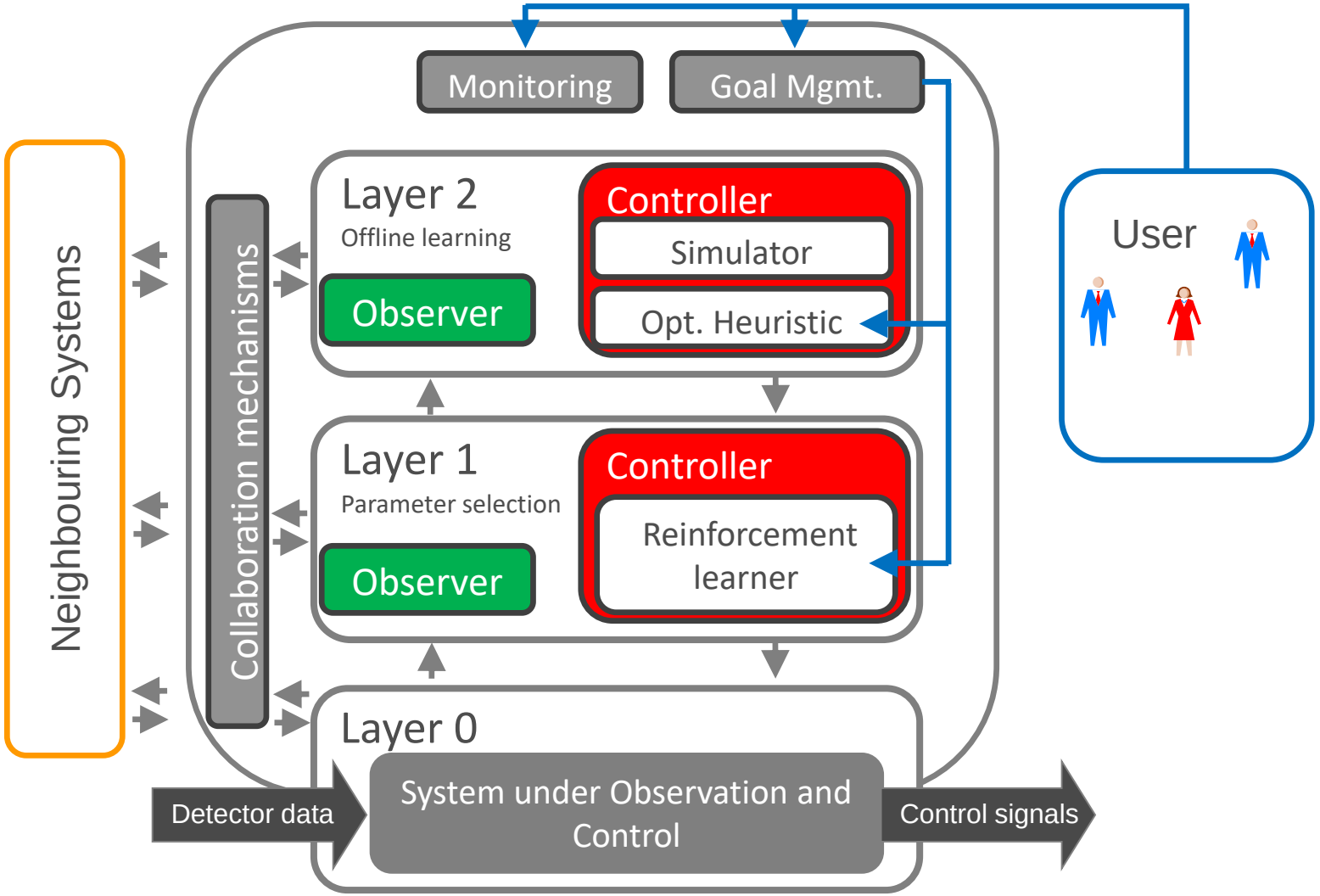


# Variant C: Hybrid



# Multi-level Observer/Controller framework

Christian-Albrechts-Universität zu Kiel



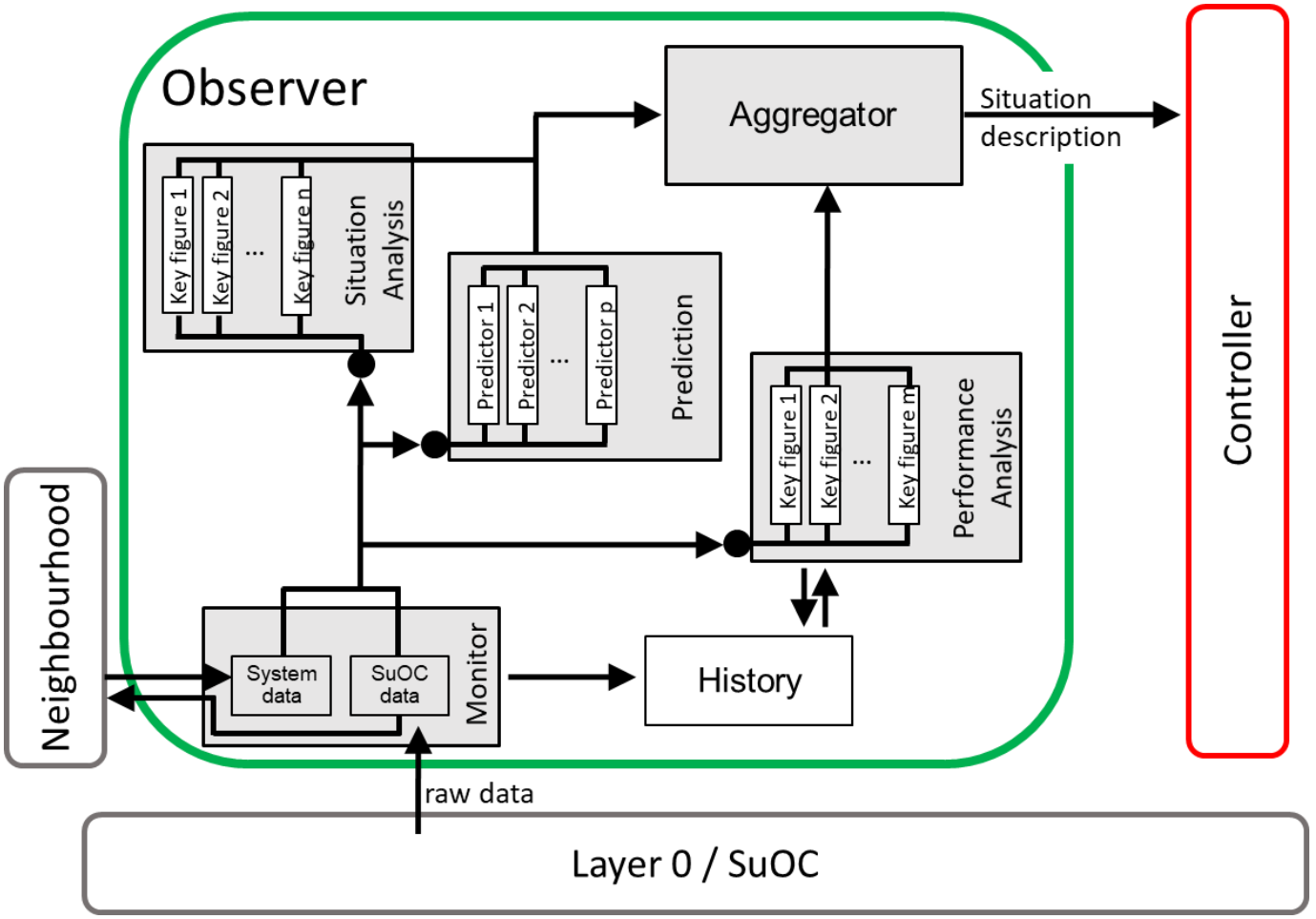
# Multi-level Observer/Controller framework (2)

Christian-Albrechts-Universität zu Kiel

- Layer 0: Productive system
  - Encapsulation of the SuOC
  - Provide access to observation and control interfaces
- Layer 1: Parameter selection and online learning
  - Observation of Layer 0:
    - Monitoring, pre-processing, data analysis, prediction, aggregation
    - Generation of situation description
  - Control:
    - Change parameters, structure, techniques to current conditions.
    - Select from existing rule-set, update [Evaluation] based on feedback.
- Layer 2: Offline learning
  - Observation of Layer 1: Missing or inappropriate knowledge
  - Control: Generate new rule, add to rule-based of Layer 1
- Layer 3: Collaboration
  - User: Self-explanation, goal modification
  - Other systems: Negotiation, communication, trust, …

# Observer at Layer 1



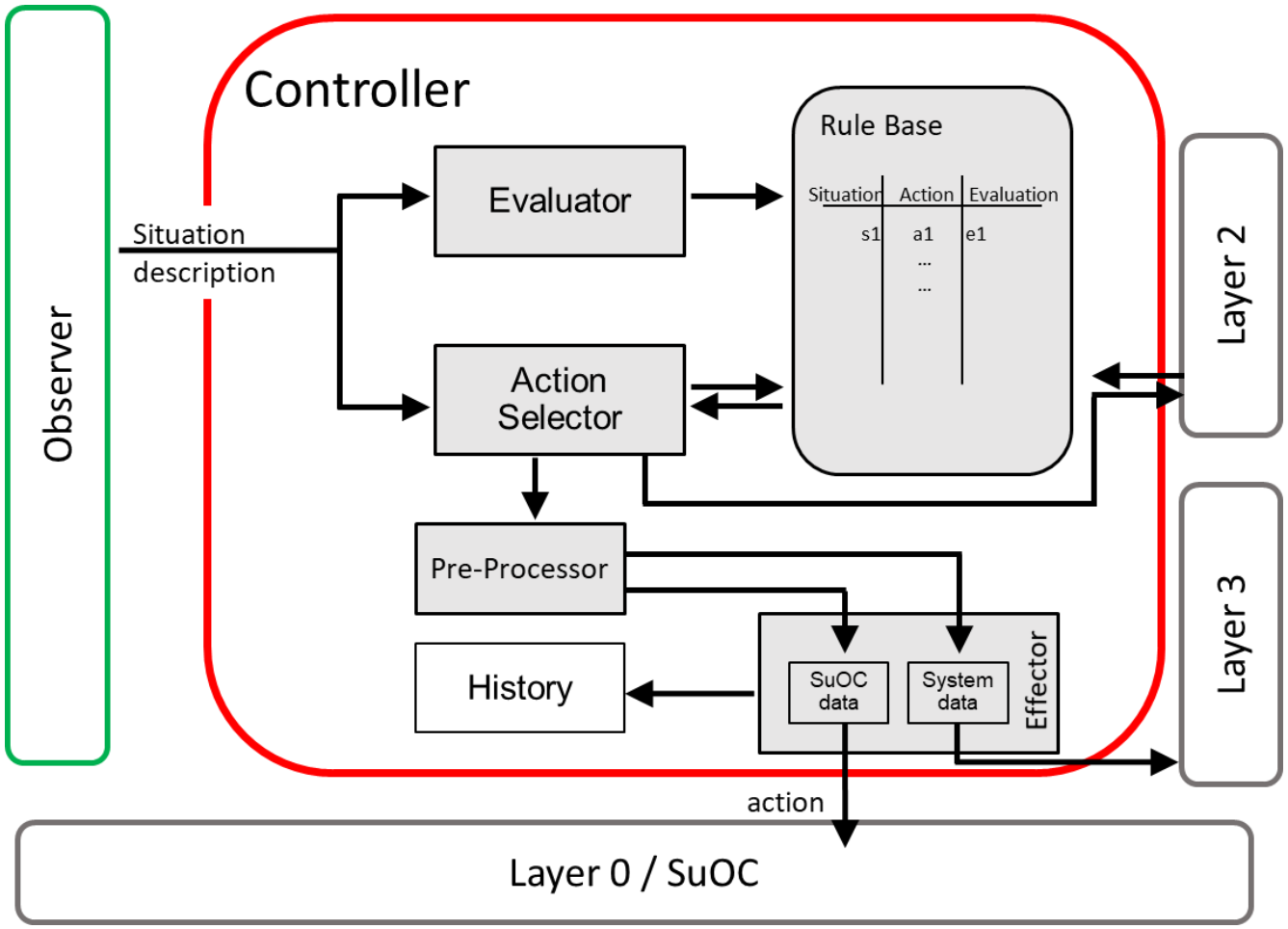


Components:

- Monitor: Gather sensor and status data
- Pre-processing: Estimate missing values,
  delete wrong values, smoothing, etc.
- Prediction: Forecasts for some of the underlying values
- Situation analysis: Find patterns, detect emergence, etc.
- Performance analysis: Extract values related to goals
- Aggregator: Build situation description
- History: List of observed data (for debugging and explanation)

# Controller at Layer 1





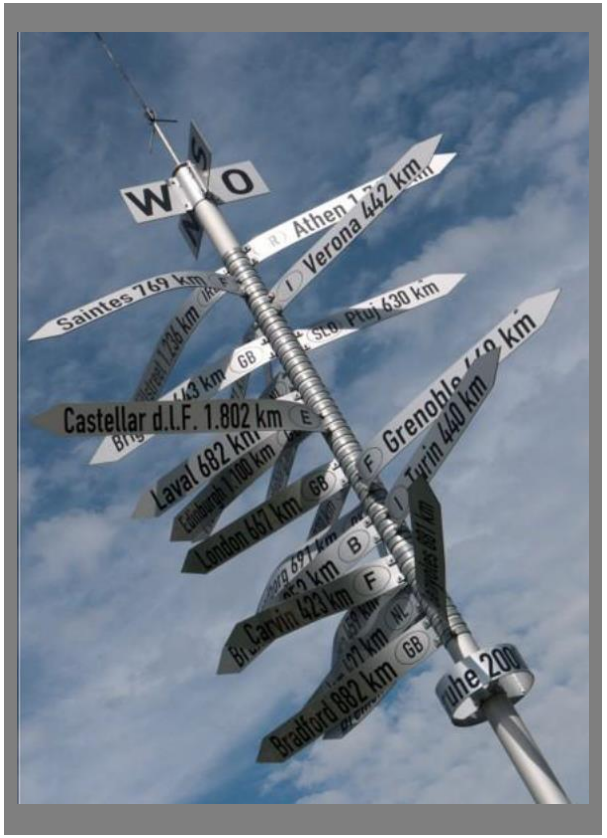
Components:

- Rule base: Set of rules, knowledge of the system
- Action selection: Select rule, the action is encoded
- Effector: Activate action via effectors
- Evaluator: Determine the success of last action(s), update rules
- Action history: List of situation-action mapping that has been performed

# Agenda

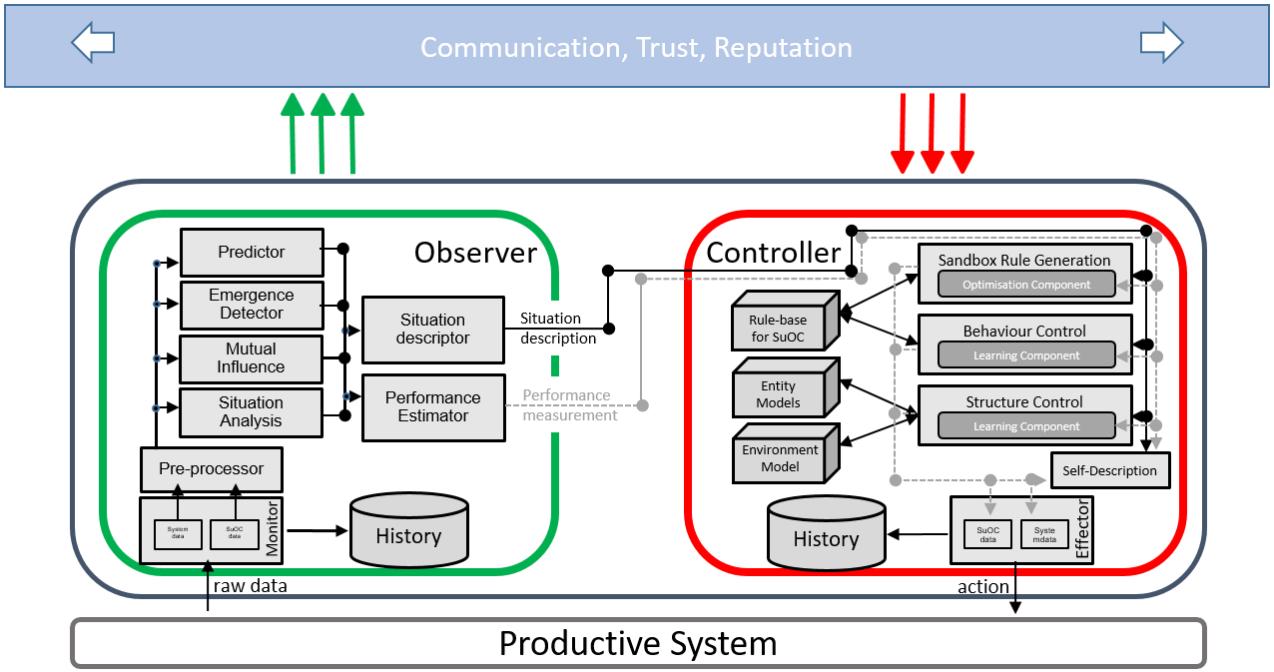
Christian-Albrechts-Universität zu Kiel

- Motivation
- Examples for the growing complexity
- Organic Computing: Nature as inspiration
- Design of Organic Computing systems
- **Example: Organic Traffic Control**
- Autonomic Computing
- A reference architecture
- Holonic systems
- Conclusion and references



# Architectures for intelligent distributed systems

- Establishes feedback loop on top of a productive system
- Realises self-adaptation and learning capabilities
- Self-organisation and cooperation realised by middleware

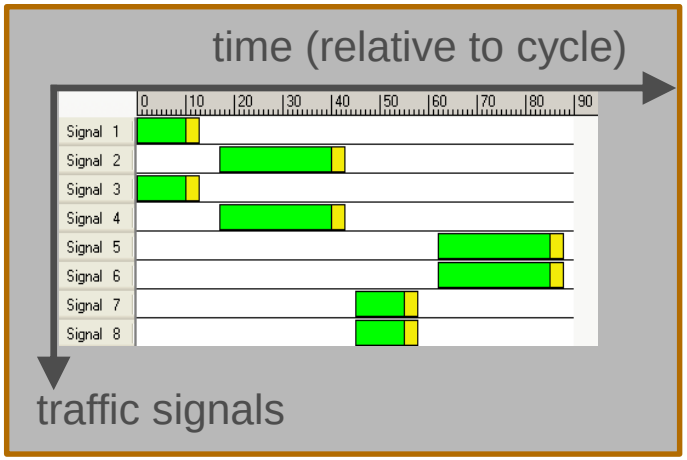


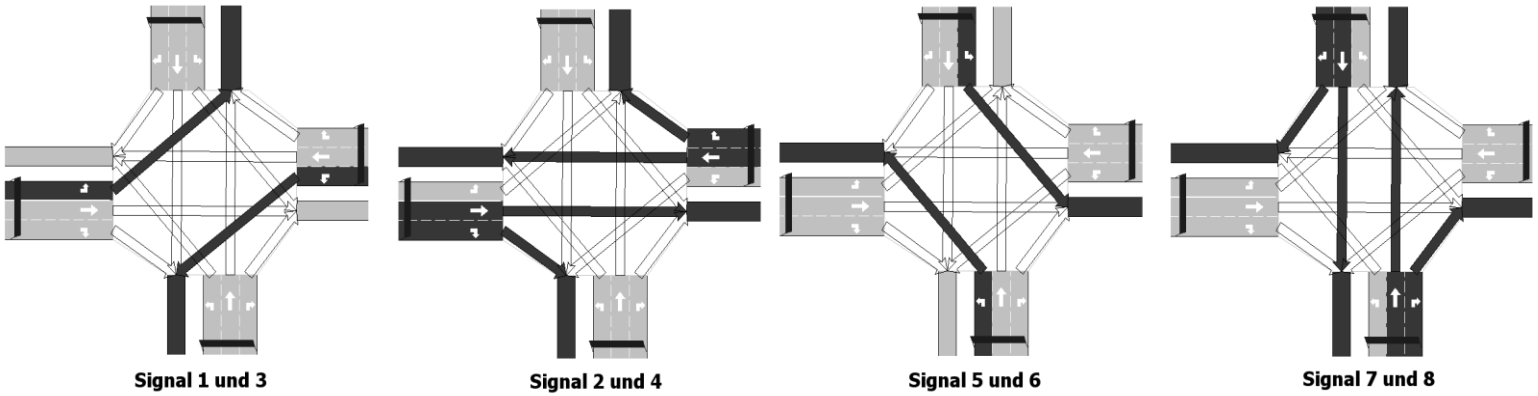

"Incremental Design of Adaptive Systems", Journal of Ambient Intelligence and Smart Environments 2014

Christian-Albrechts-Universität zu Kiel

Organic Traffic Control system

- Adaptation of green durations at the intersection
- Progressive signal systems (“green waves“)
- Decentralised route recommendations
- Anticipatory signalisation and routing



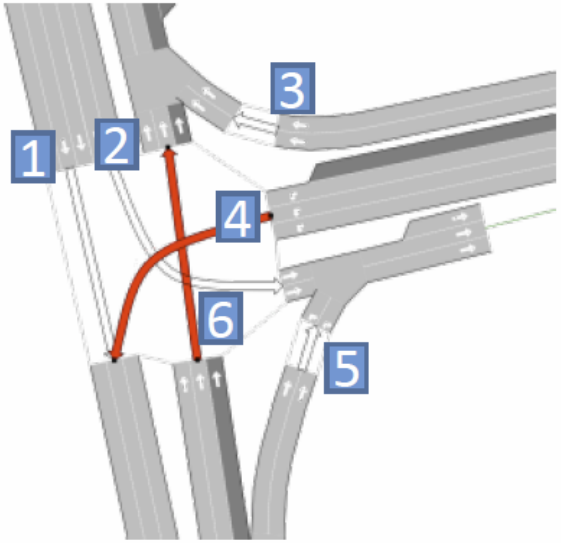

- Goal: Decrease waiting times (alternatives: emissions, stops, etc.)

“An Organic Approach to Resilient Traffic Management”. In Autonomic Road Transportation Systems (2015), pp. 113 - 130.

Christian-Albrechts-Universität zu Kiel

Classifier:

| *Condition* | *Action* | *p* | *ε* | *F* |
|---|---|---|---|---|
| [420,530] [700,800][…] | TLC05 | 26 | 03 | .99 |




Action (TLC):

1, 2 Phase 1: 35 sec

3, 4 Phase 2: 15 sec

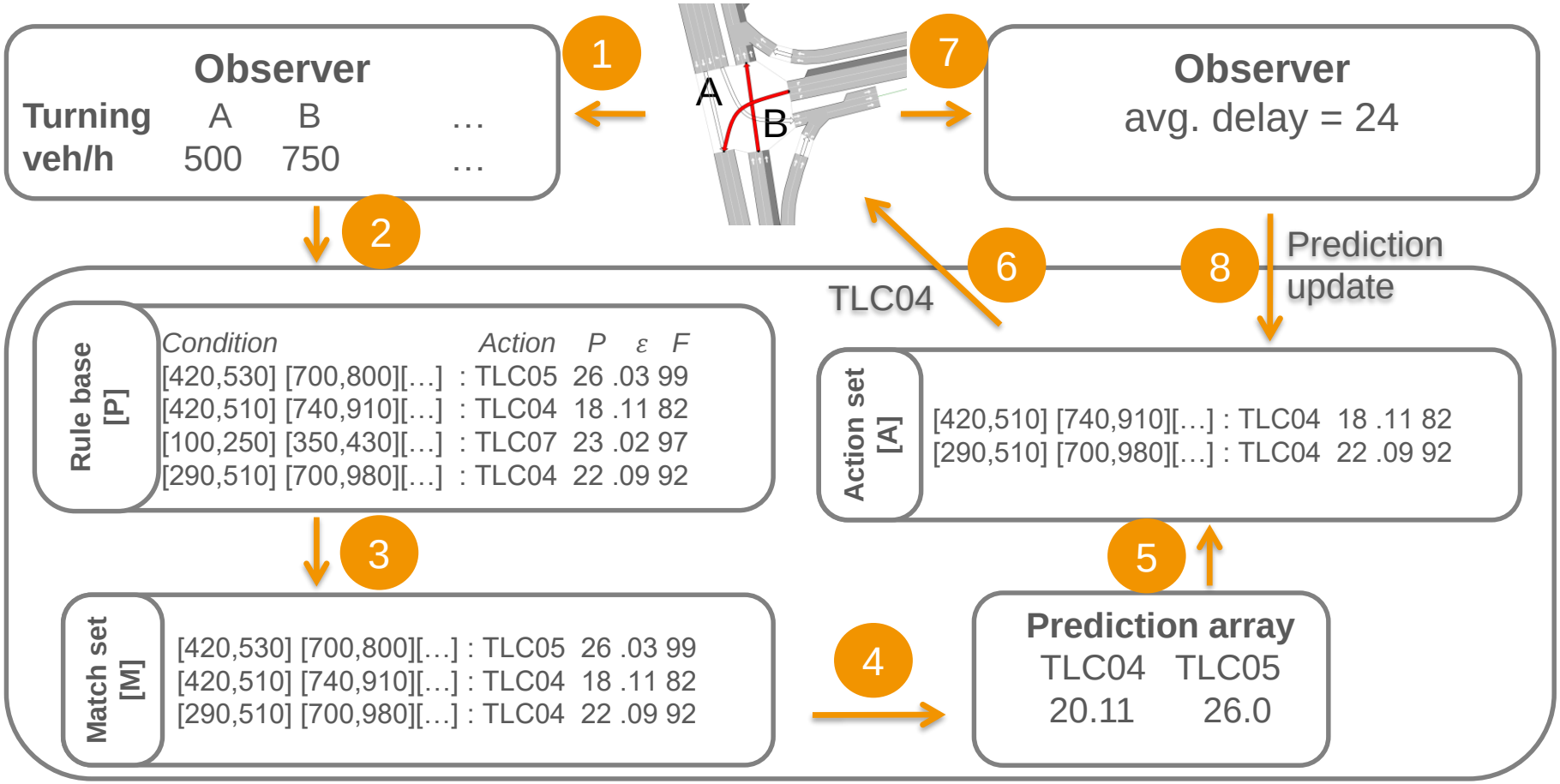
5, 6 Phase 3: 30 sec

TLC = (35, 15, 30)

(Ordering of phases and length of interphases are fixed and not subject to OTC control.)

# Reinforcement-based self-optimisation

Christian-Albrechts-Universität zu Kiel



Update rules:

- Utility: flow-weighted delay at intersection
- Prediction update:

$$p_j = p_j + \beta(P - p_j)$$

- Error update:

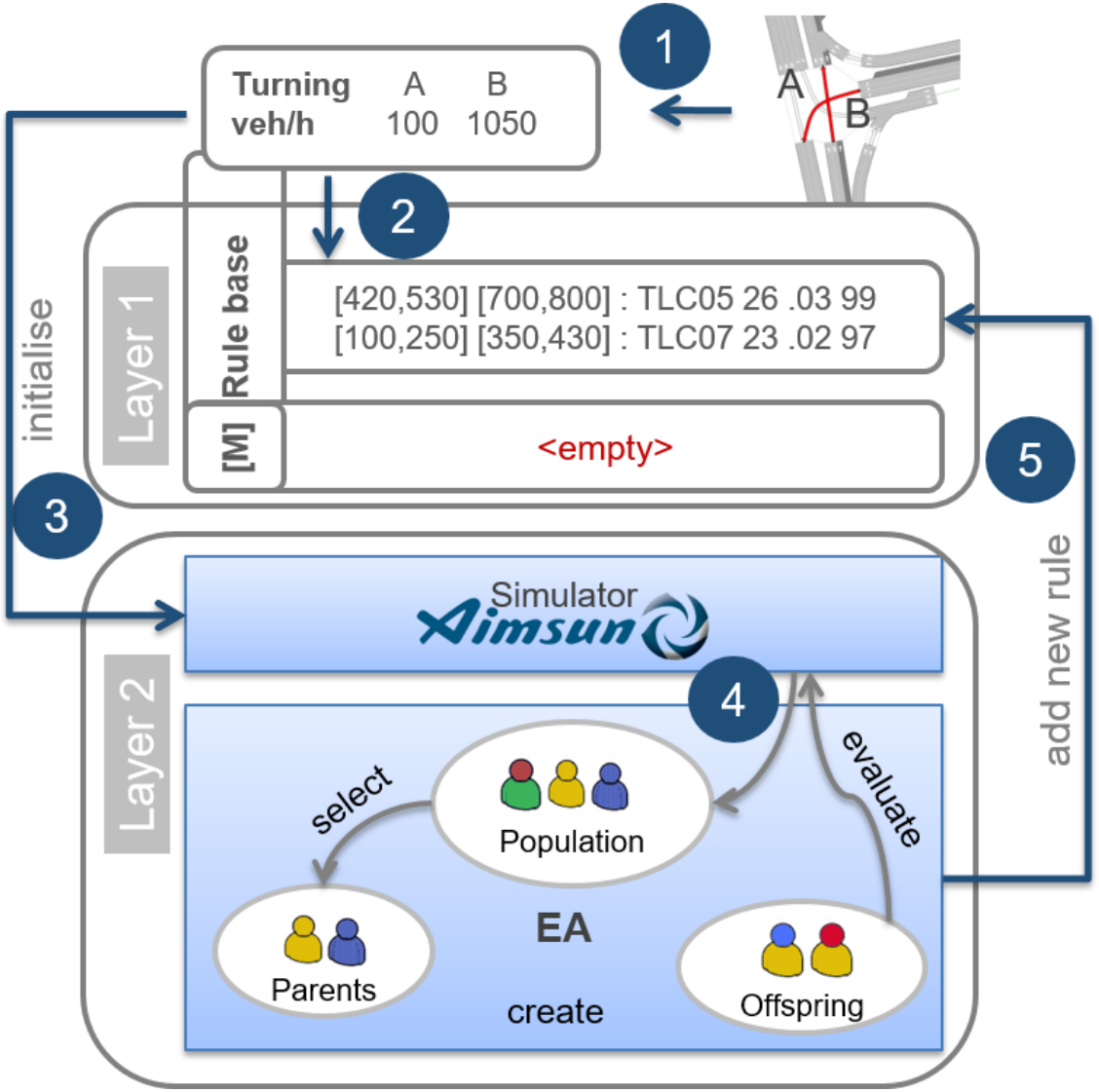
$$\epsilon_j = \epsilon_j + \beta\left(\left|P - p_j\right| - \epsilon_j\right)$$

- Fitness depends on error update

- $\beta$ is the learning rate (typically set to 0.2) and P the reward signal (here: the measured flow-weighted delay)

# Rule generation



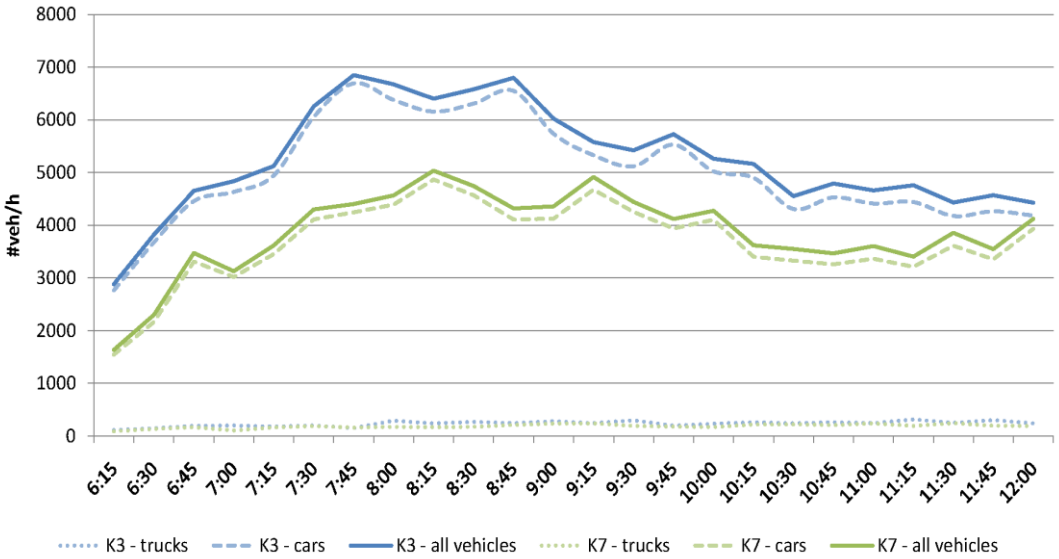


# Evaluation

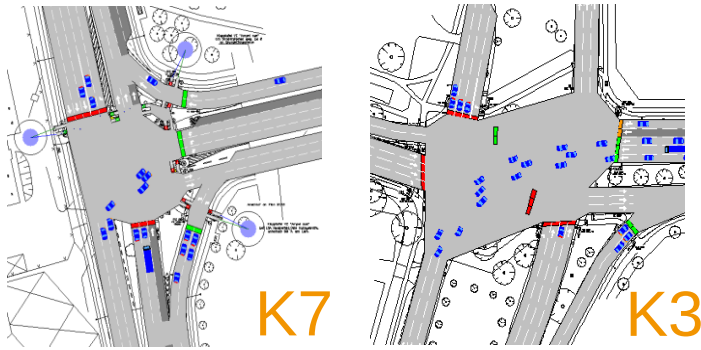


- Network situated in Hamburg
- Traffic demands from census
- Reference: fixed-time control strategy (i.e., as a result of manual optimisation by engineers)
- Normal working day / morning rush hour


Road network at Hamburg



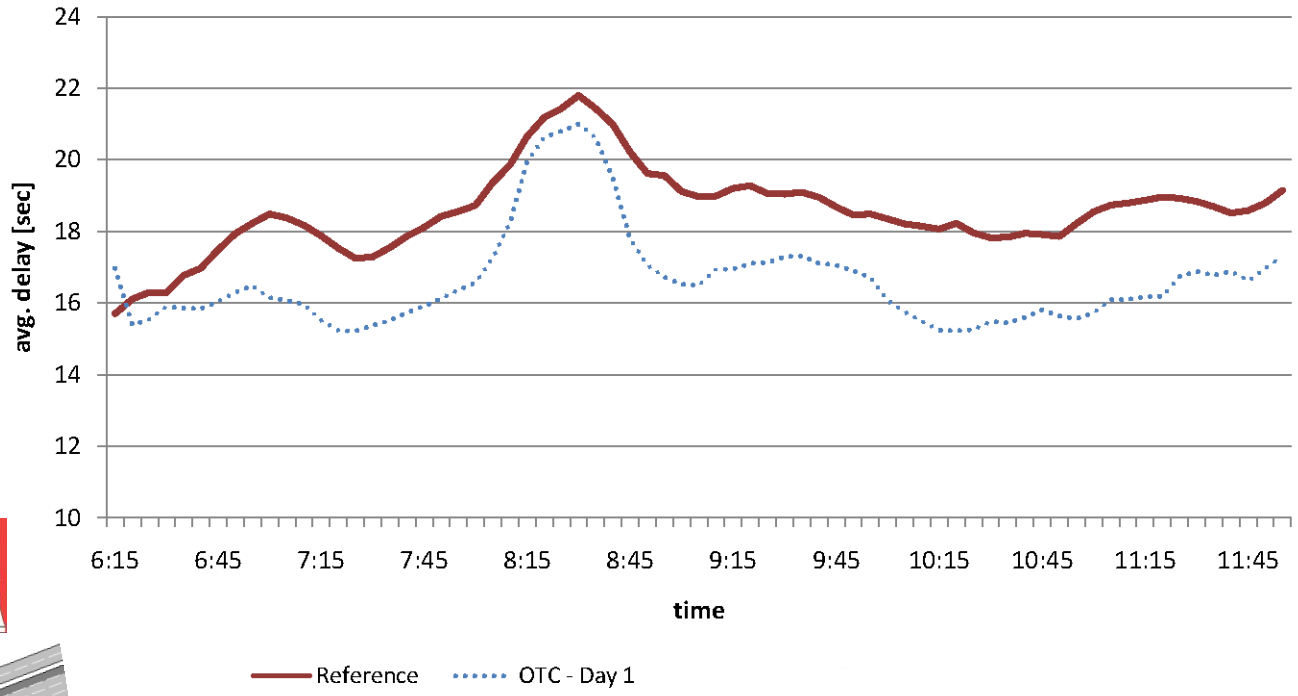

Traffic demands according to the census for intersections K3 and K7

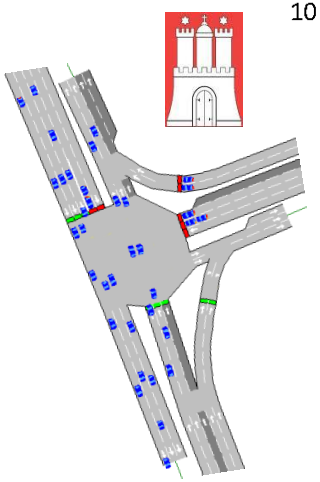



Topology models of intersections K7 und K3

# Benefit of self-adaptation

Christian-Albrechts-Universität zu Kiel



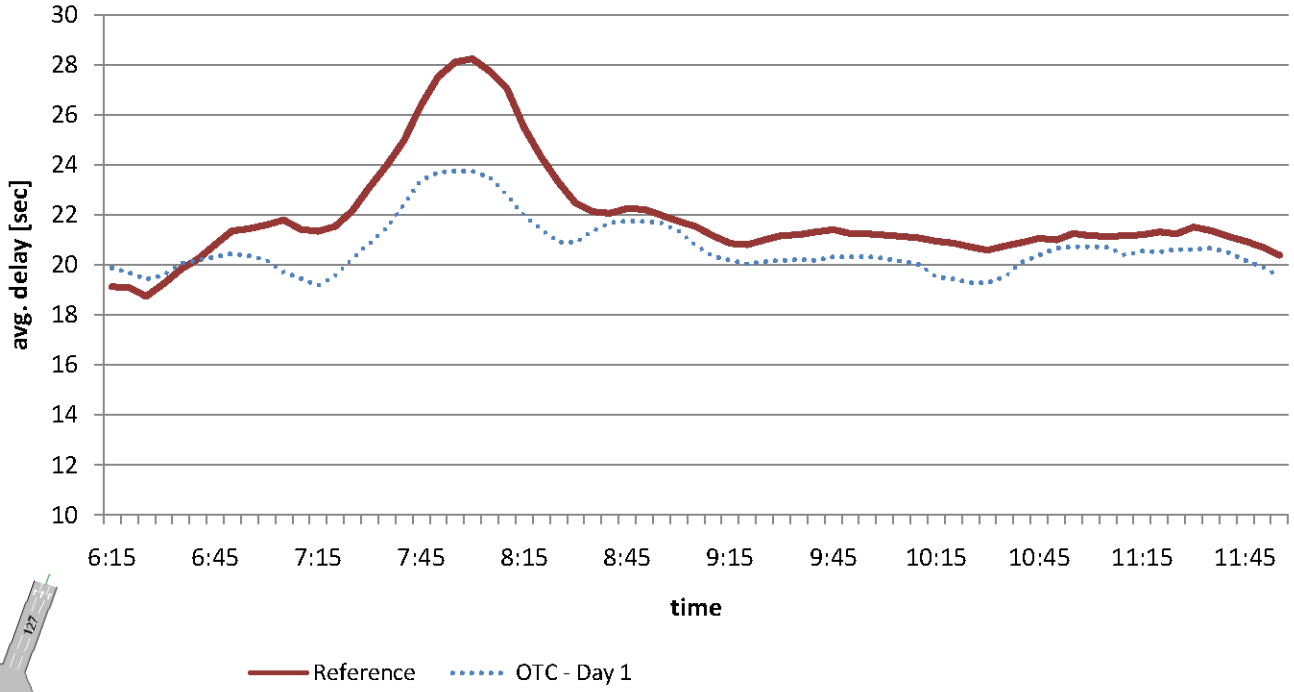



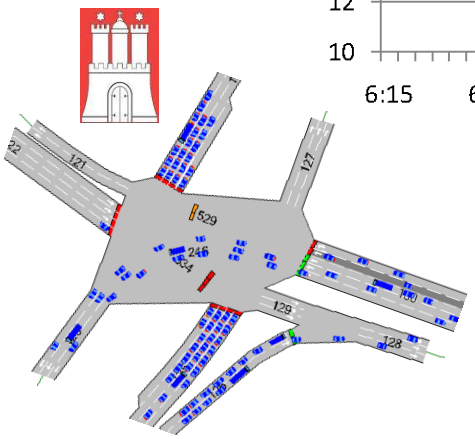
| | Day 1 | Day 2 | Day 3 |
|---|---|---|---|
| Improvement by organic control | 10% | | |

K7 (Kollaustraße / Nedderfeld)

# Benefit of self-adaptation (2)





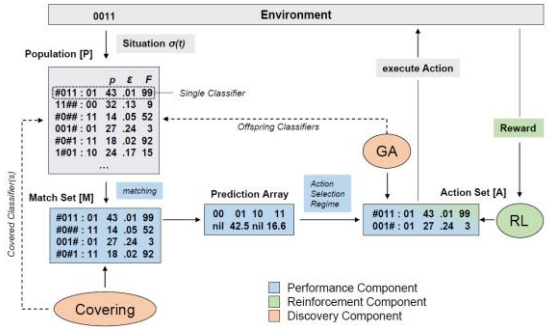




| | Day 1 | Day 2 | Day 3 |
|---|---|---|---|
| Improvement by organic control | 6% | | |

K3 (Alsterkrugchaussee / Deelböge)

# Subsequent research output






Jniversität zu Kiel

## Application: Traffic control

### a) Progressive Signal Systems

- Self-organised coordination
- Establishes “green waves“
- Hierarchical extension

### b) Self-organised route guidance

- Resembles data communication protocols (DVR, LSR)
- Incorporates varying goals
- Hierarchical extension (BGP)

## Machine learning technology

### a) Knowledge aspects

- Assessment of knowledge
- Proactive knowledge generation
- Transfer of knowledge

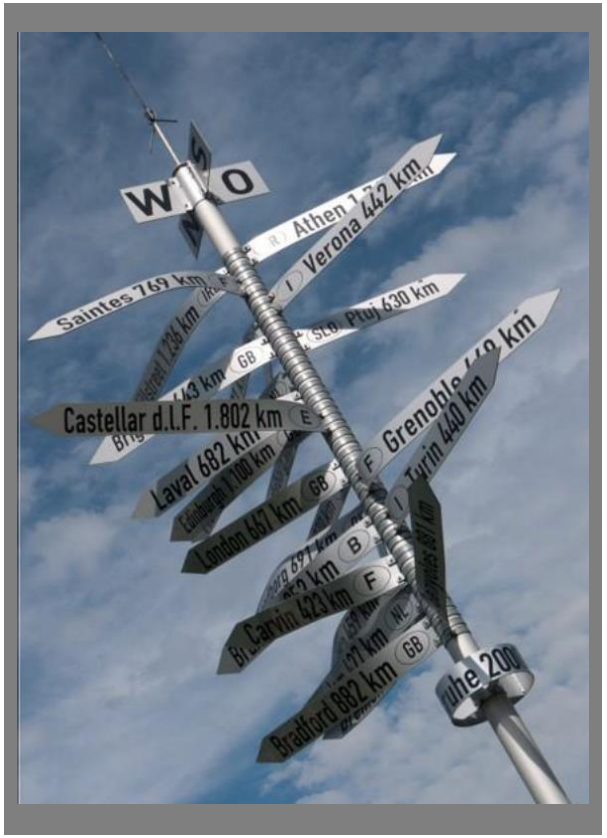
### b) XCS variants

- Efficient covering
- Opportunistic knowledge source integration

# Agenda



- Motivation
- Examples for the growing complexity
- Organic Computing: Nature as inspiration
- Design of Organic Computing systems
- Example: Organic Traffic Control
- Autonomic Computing
- A reference architecture
- Holonic systems
- Conclusion and references



Autonomic and autonomy-oriented computing

- In ‘The Vision of AC‘, IBM warns that the dream of the interconnectivity of computing systems and devices could become the “nightmare of pervasive computing” in which architects are unable to anticipate, design and maintain the complexity of interactions. They state the essence of AC is system self-management, freeing administrators from low-level task management while delivering more optimal system behaviour.
- The focus of AC is set on building an artificial autonomic nervous system e.g. large server farms

COVER FEATURE

# The Vision of Autonomic Computing

**Systems manage themselves according to an administrator's goals. New components integrate as effortlessly as a new cell establishes itself in the human body. These ideas are not science fiction, but elements of the grand challenge to create self-managing computing systems.**

*Jeffrey O. Kephart*
*David M. Chess*
IBM Thomas J. Watson Research Center

In mid-October 2001, IBM released a manifesto observing that the main obstacle to further progress in the IT industry is a looming software complexity crisis.[1] The company cited applications and environments that weigh in at tens of millions of lines of code and require skilled IT professionals to install, configure, tune, and maintain.

The manifesto pointed out that the difficulty of managing today's computing systems goes well beyond the administration of individual software environments. The need to integrate several heterogeneous environments into corporate-wide computing systems, and to extend that beyond company boundaries into the Internet, introduces new levels of complexity. Computing systems' complexity appears to be approaching the limits of human capability, yet the march toward increased interconnectivity and integration rushes ahead unabated.

This march could turn the dream of pervasive computing—trillions of computing devices connected to the Internet—into a nightmare. Programming language innovations have extended the size and complexity of systems that architects can design, but relying solely on further innovations in programming methods will not get us through the present complexity crisis.

As systems become more interconnected and diverse, architects are less able to anticipate and design interactions among components, leaving such issues to be dealt with at runtime. Soon systems will become too massive and complex for even the most skilled system integrators to install, configure, optimize, maintain, and merge. And there will be no way to make timely, decisive responses to the rapid stream of changing and conflicting demands.

## AUTONOMIC OPTION

The only option remaining is *autonomic computing*—computing systems that can manage themselves given high-level objectives from administrators. When IBM's senior vice president of research, Paul Horn, introduced this idea to the National Academy of Engineers at Harvard University in a March 2001 keynote address, he deliberately chose a term with a biological connotation. The autonomic nervous system governs our heart rate and body temperature, thus freeing our conscious brain from the burden of dealing with these and many other low-level, yet vital, functions.

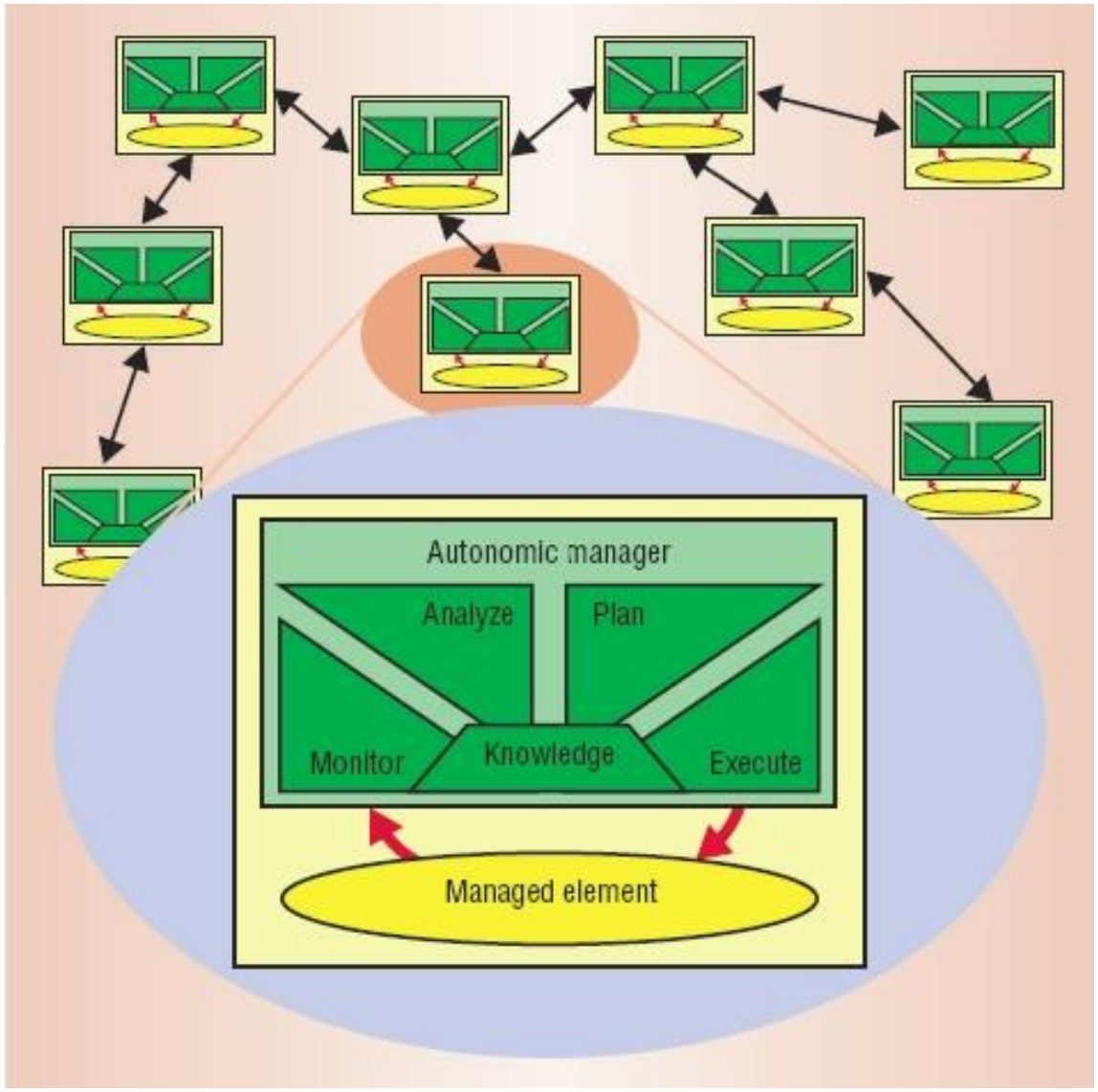
The term autonomic computing is emblematic of a vast and somewhat tangled hierarchy of natural self-governing systems, many of which consist of myriad interacting, self-governing components that in turn comprise large numbers of interacting, autonomous, self-governing components at the next level down. The enormous range in scale, starting with molecular machines within cells and extending to human markets, societies, and the entire world socioeconomy, mirrors that of computing systems, which run from individual devices to the entire Internet. Thus, we believe it will be profitable to seek inspiration in the self-governance of social and economic systems as well as purely biological ones.

Clearly then, autonomic computing is a grand

0018-9162/03/$17.00 © 2003 IEEE Published by the IEEE Computer Society January 2003 

# Autonomic Computing (2)





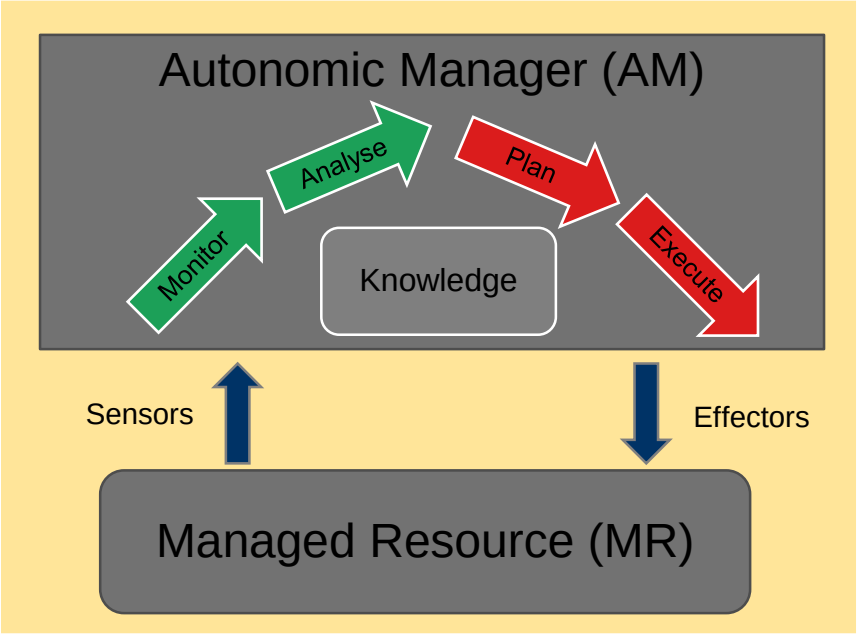
## The vision of Autonomic Computing (AC)

- AC tries to control complexity using principles, such as:
  - Distribution of responsibilities from central entities to local ones.
  - Local goals, decentralised control, (large) populations
  - Self-* properties (for AC: Self-CHOP)

- Self-CHOP
  - Self-**c**onfiguring: automatic configuration of components
  - Self-**h**ealing: automatic discovery, and correction of faults
  - Self-**o**ptimising: automatic monitoring and control of resources to ensure the optimal functioning concerning the defined requirements
  - Self-**p**rotecting: proactive identification and protection from arbitrary attacks

## Designing AC systems

- Basic element: the MAPE (**M**onitor, **A**nalyse, **P**lan, **E**xecute) cycle
- Clear distinction between:
  - Productive system: Managed Resource (MR)
  - Control system: Autonomic Manager (AM)
- The MR describes a productive *software* system that has to fulfil certain tasks.
- The AM is responsible to control the resource. Therefore, it can monitor the MR by its sensors, derive an adaptation strategy using the four name-giving steps, and activate the derived plan through its effectors.

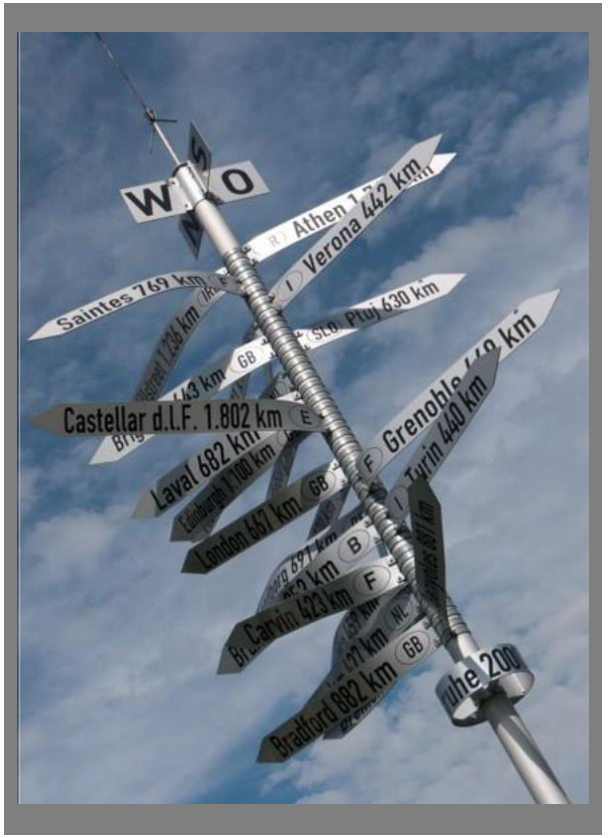


- Ever heard before?
  - Yes, indeed! The approach is similar to Organic Computing.
  - AC as a vision has addressed many central issues already in 2003, but many problems remain unsolved. Especially the aspects of runtime optimisation and all connected problems (like automated learning, degrees of freedom, etc.) have not been investigated sufficiently by a broader community.
  - Compared to OC, the AC initiative has a different focus. Instead of aiming at a new paradigm for system design (both, hardware and software), it deals mainly with large software solutions. In addition, data centre management has been considered as the major challenge, which leaves most of the other application domains and problems unattended.
  - As a summary, AC, which provides predominantly a policy-oriented solution for data centre management, can be seen as addressing just a subset of OC.

# Agenda

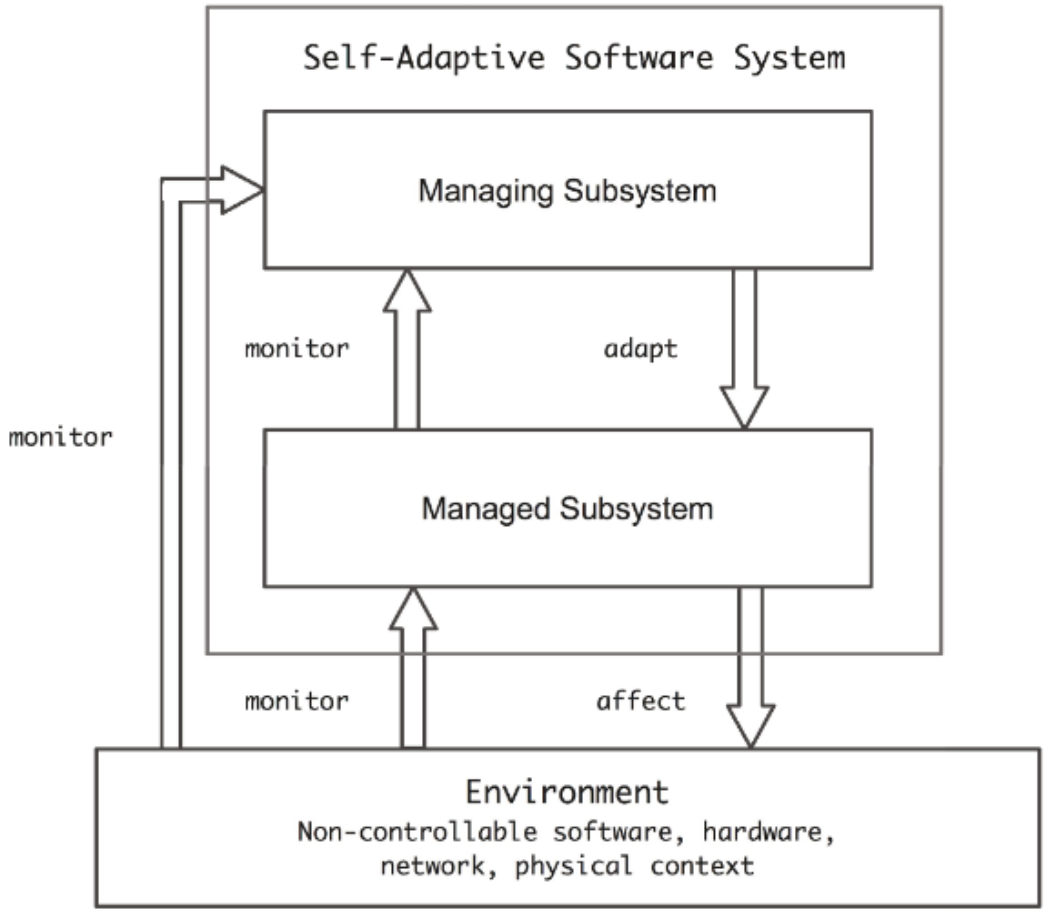


- Motivation
- Examples for the growing complexity
- Organic Computing: Nature as inspiration
- Design of Organic Computing systems
- Example: Organic Traffic Control
- Autonomic Computing
- A reference architecture
- Holonic systems
- Conclusion and references



## Design concept

- Generalise the concepts of OC and AC
- Reference architecture for self-adaptive and self-organising (SASO) systems
- Usage of the AC terminology in most cases



## Terminology

- We use the general terms ‘managed subsystem’ and ‘managing subsystem’ to denote the constituent parts of a self-adaptive software system (SAS).
- The environment refers to the part of the external world with which the self-adaptive system interacts
  - Also: in which the effects of the system will be observed and evaluated
  - It may correspond to both physical and soft-ware entities.
- Example: a robotic system
  - Environment includes physical entities such as obstacles on the robot’s path and other robots.
  - May also contain external cameras and corresponding software drivers.
  - Distinction between environment and SAS is based on the extent of control: The SAS may interface with the mountable camera sensor, but since it does not manage (adapt) its functionality, the camera can be considered to be part of the environment.

## Terminology

- The managed subsystem comprises the application logic that provides the system's domain functionality.
- Robots example: navigation of a robot or transporting loads is performed by the managed subsystem
- Managed subsystem monitors and affects the environment to realise its functionality
- To support adaptations, the managed subsystem has to provide support for monitoring and executing adaptations.

## Terminology

- The managing subsystem control and adapts the managed subsystem.
- It comprises the adaptation logic that deals with one or more concerns.
- Robot example: A robot may be equipped with a managing subsystem that allows the adaption of its navigation strategy based on the changing operating conditions.
- Such condition may be changing task load or reduced bandwidth for communication.
- The managing subsystem monitors the environment and the managed subsystem and adapts the latter when necessary to realise its goals.

## Terminology

- Other layers can be added to the system.
- Idea: higher-level managing subsystems steer the behaviour of underlying subsystems.
- These underlying subsystems can also be managing subsystems.
- Robot example: Robot has the ability to adapt its navigation strategy, but also the way such adaptation decisions are made.
- Decision is based on, e.g., remaining energy level of the battery.
- Here, the subsystem responsible for managing the battery level of the robot must co-ordinate with the subsystem for managing navigation and other robotics tasks, so that the robot does not fail entirely.

Distributed systems differentiate between:

1. centralised data in contrast to distributed, partitioned, and replicated data
2. centralised services in contrast to distributed, partitioned and replicated services
3. centralised algorithms in contrast to decentralised algorithms

Distribution in the context of SAS:

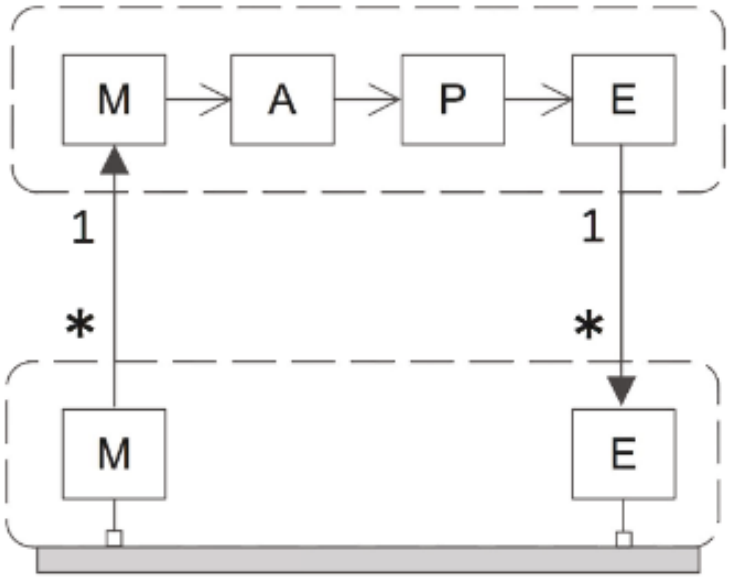
- Refers to the deployment of the software of a SAS to hardware.
- I.e. deployment of the software of both the managed subsystem and the managing subsystem.
- A distributed SAS consists of multiple software components that are deployed on multiple nodes connected via some network.
- Opposite of a distributed SAS: software is deployed on a single node
- The managed and managing subsystems can be deployed on the same or on different nodes.

## Decentralisation

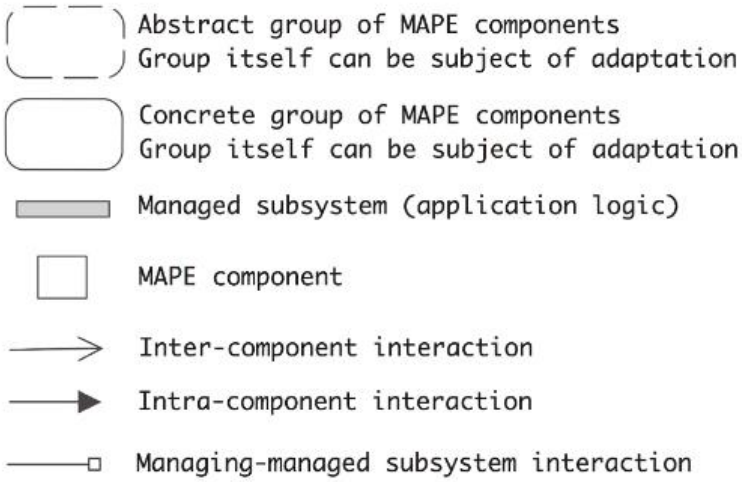
- Refers how to control decisions in a SAS are coordinated among different components, independent of how those control components are physically distributed.
- Consider decentralisation at the level of the four activities of self-adaption:
  - monitoring,
  - analysing,
  - planning, and
  - execution
- Implies a type of control in which multiple components responsible for one of the activities of self-adaption perform their functionality locally, but coordinated with peers.
- Typical examples for such decentralised coordination:
  - Monitoring components coordinate with other monitoring components to collect the knowledge required for subsequent analysis
  - Analysis components coordinate to decide whether the conditions for a particular adaptation hold
  - Multiple planning components coordinate to plan an adaptation
  - Multiple execution components coordinate to execute an adaptation (synchronise their adaptation actions).

# Example of a MAPE pattern



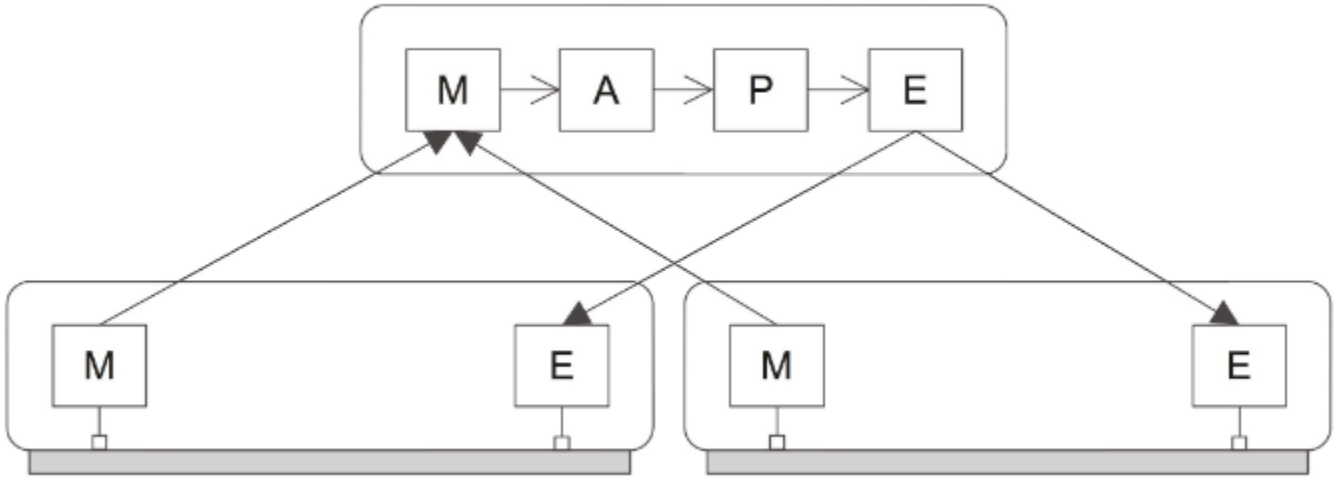



Legend:



# Example of a MAPE pattern (2)






A particular instance of the previous MAPE pattern

- Two distributed managed subsystems are controlled by one MAPE cycle
- Access via M and E interfaces (corresponds to sensors and actuators)

# Observer/Controller vs MAPE pattern

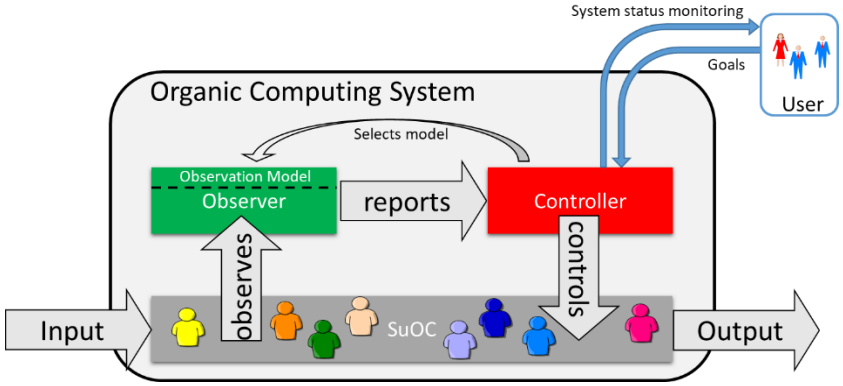


## Comparing O/C and MAPE

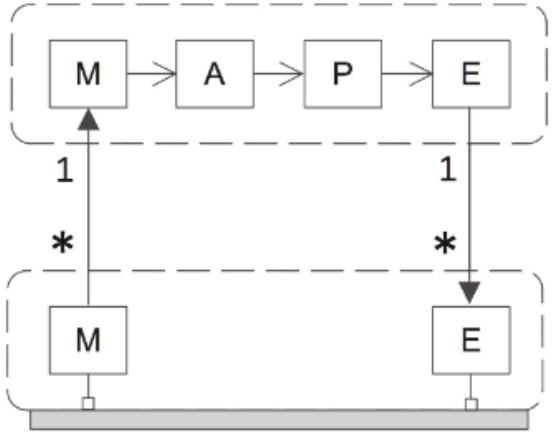
- Monitoring is done by the observer
- Analysis is done in both, observer and controller
- Planning is done in the controller, mostly in higher layers
- Execution is done in the controller

## Further differences (for MLOC)

- MAPE neglects the techniques
- MAPE does not incorporate learning mechanisms
- Details on processing the time-series data is missing

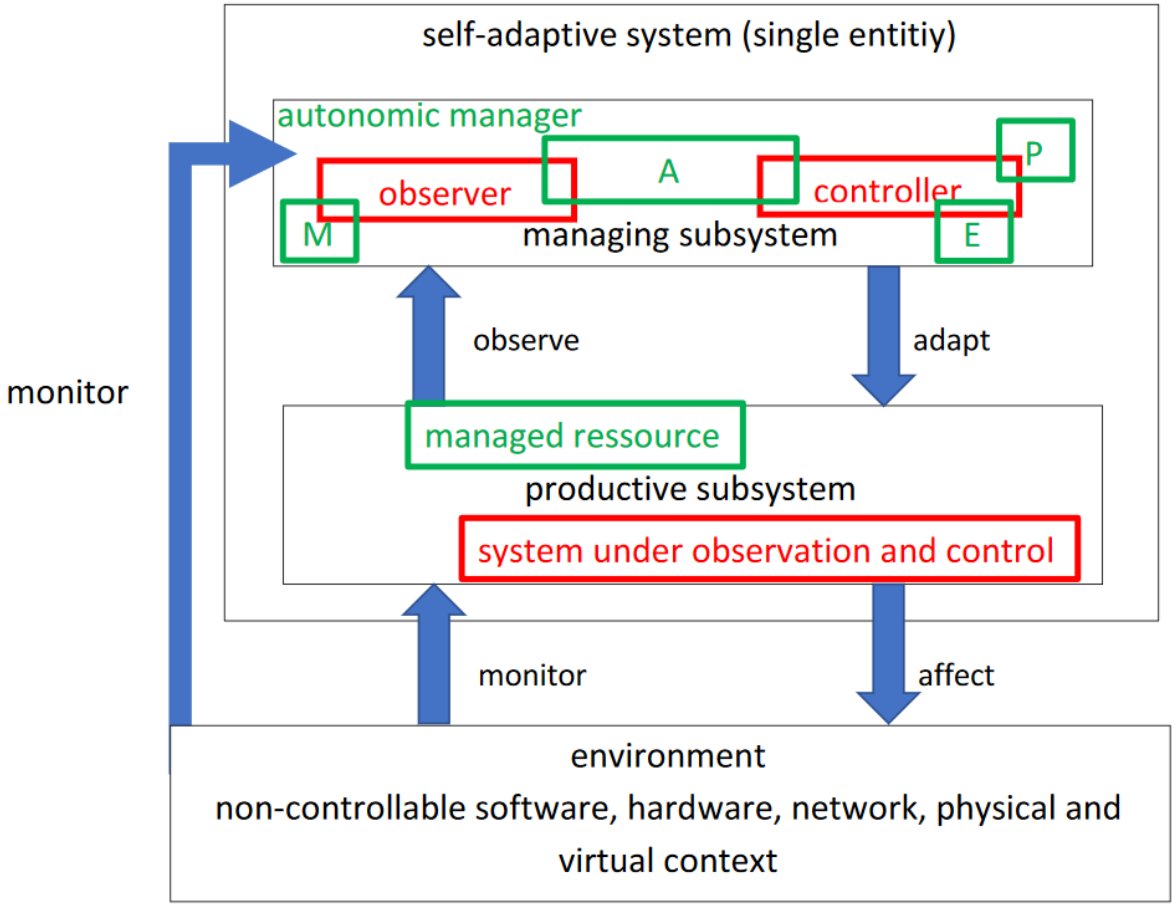



vs.



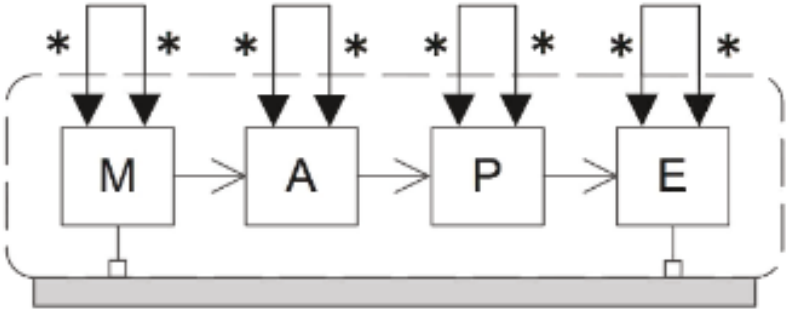
# Observer/Controller vs MAPE pattern (2)



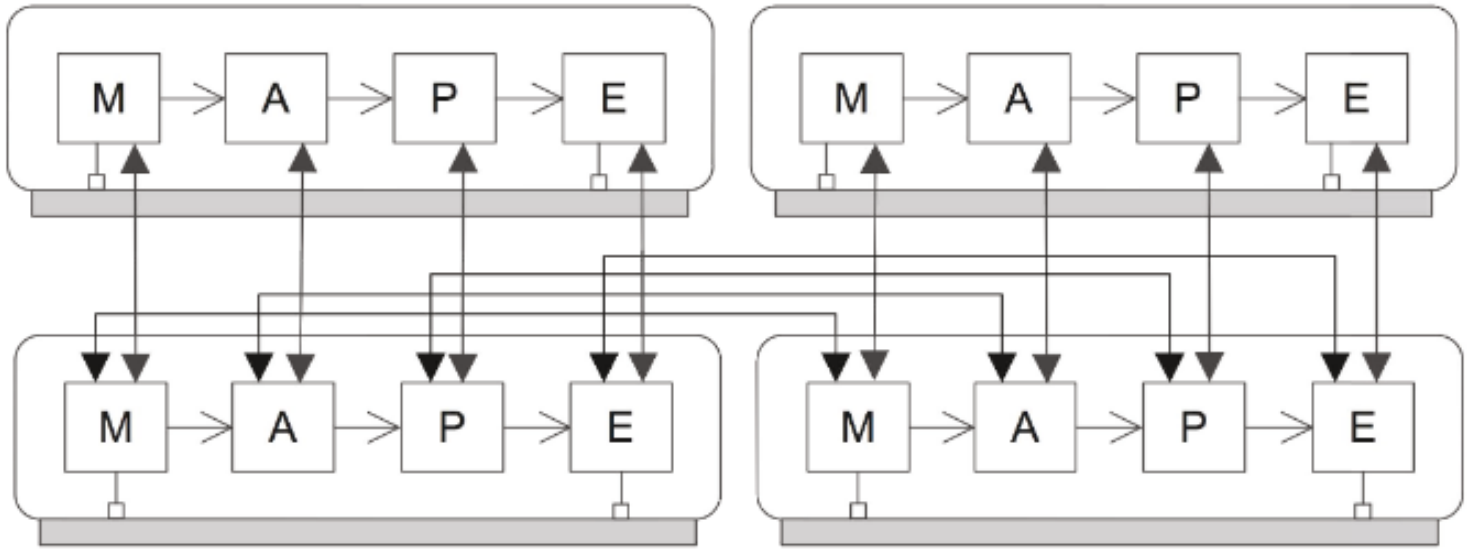


## Coordination of SAS

- Goal: Joint operation of distributed SAS in shared environments
- Coordination refers to all four steps



Example: A possible instance of the coordination pattern



## Decentralised control: Advantages

- Potential of good scalability concerning communication and computation.
- Depends on the coupling degree among peer components, and the number of other peers each MAPE component has to explicitly interact with.
- If adaptations can be performed based on local interactions between MAPE components: the communication overhead is limited to interactions with local peers.
- Computational burden is spread over the nodes.
- May also contribute to improving robustness as there is no single point of failure.
- Especially required if no single entity has the knowledge or authority to coordinate adaptations across a set of managed subsystems.

## Decentralised control: Drawbacks

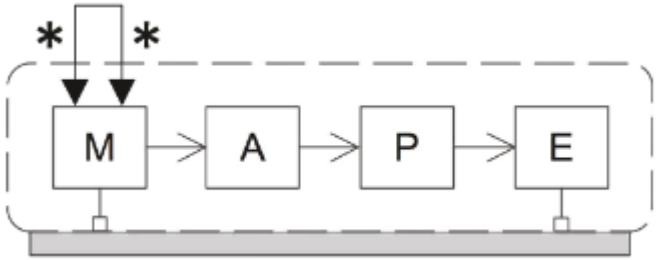
- When coordination is required between MAPE components of many nodes, scalability may be compromised.
- The cost for reaching consensus about suitable adaptation actions may be high (in terms of communication and/or timing).
- Decentralised control may cause problems with ensuring consistency of adaptations.
- It may lead to sub-optimal adaptation decisions and actions, from the overall system viewpoint.

Challenge for SAS

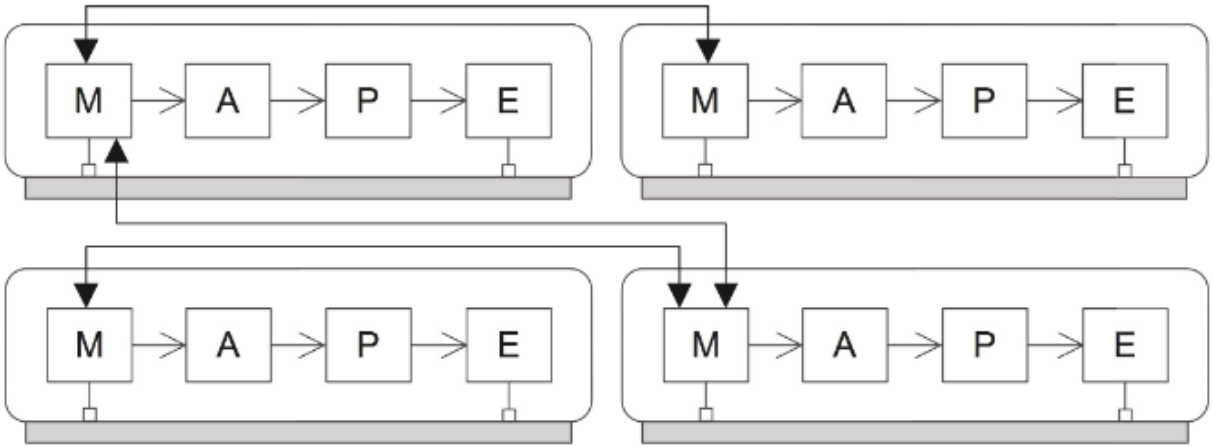
- A software system consisting of a (potentially large) set of loosely connected components requires support for adaptation to maintain a particular concern or quality attribute.
- The components of the system are deployed on different nodes.
- Each part of the system can adapt locally but requires information about the state of other nodes in the system because a local adaptation may impact these other nodes (e.g., on some quality attribute of those operations).
- However, apart from information sharing, nodes do not need to coordinate other adaptation activities.
- Example: Sensor network for environmental monitoring
  - Certain nodes may be equipped with sensors to detect a fire.
  - Node detects a fire: Raise alarm signal that can be spread effectively through the network using a smart gossip algorithm.
  - Node receiving the signal: Activate local adaptation to anticipate disaster

The pattern for information sharing

- Focus: Monitoring




- Particular instance of the pattern:



## Consequences of information sharing

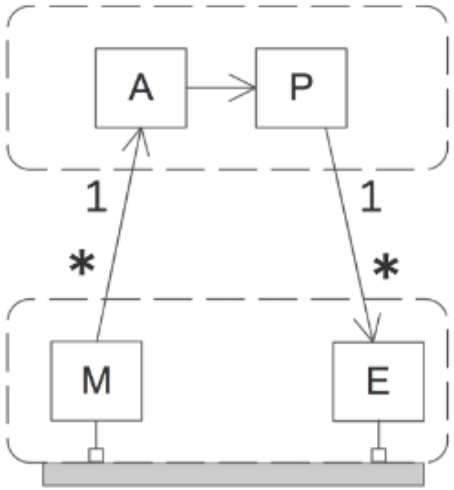
- Scalability perspective: Information sharing may produce potentially higher benefits than coordinated control.
- Less stringent interaction requirements (i.e., limited to ‘M’ components only) may result in solutions that scale even better with respect to the communication.
- This requires that the traffic between ‘M’ components is limited in scope and volume.
- Another potential benefit: Since the ‘P’, ‘A’, and ‘E’ components can act locally without the need for coordination, this may lead to more timely decisions and execution of adaptations.
- Disadvantage: The reduced coordination may increase locally optimal objectives but at the cost of globally optimal ones.
- Worst case: Local decisions may conflict with one another, resulting in perpetual adaptation of the system, thus wasting resources and having an adverse effect on the system’s availability and stability.

## Master/Slave Pattern: Challenge

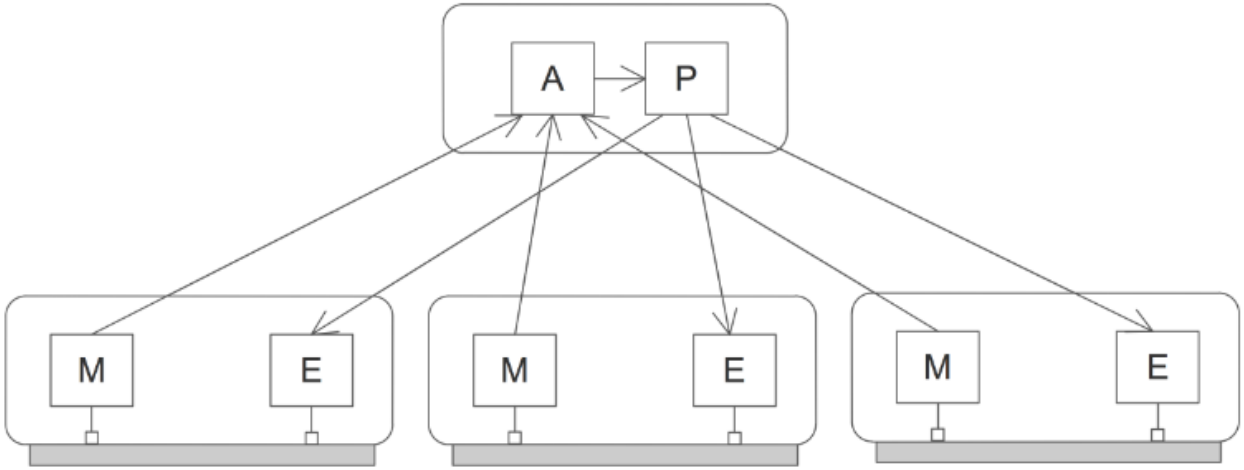
- Idea: There is a need to adapt a distributed software system for some concern.
- Monitoring and adaptations of the software are done locally at each node.
  → E.g. due to the high cost of transferring monitored data or the specificity of local adaptations
- In turn: There is a need to provide global guarantees, predictability, and consistency about the state of the distributed system and its adaptations.
- Example
  - A central controller in automated logistic systems (with cranes, conveyor belts, etc.) relies on locally collected knowledge.
  - May also rely on their environment (which may include complex processing performed by the ‘M’ components).
  - Actions: Trigger some of the machines to change their work mode (which may involve complex manipulations of the machine software performed by ‘E’ components).

The pattern for Master/Slave

- Focus: Separation of concerns




- Particular instance of the pattern:



## Consequences from the master/slave pattern

- It is a suitable solution for application scenarios in which slave control components need to process monitored information to derive the required data allowing centralised decision making, and execute local adaptation (probably based on higher-level adaptation instructions).
- Advantages:
  - Centralising the ‘A’ and ‘P’ components facilitates the implementation of efficient algorithms for analysis and planning
  - Aims at achieving global objectives and guarantees.
- Drawbacks:
  - Sending the collected information to the master component and distributing the adaptation actions may impose a significant communication overhead.
  - The solution may be problematic in case of large-scale distributed systems where the master may become a bottleneck.
  - The master component represents a single point of failure.

## Regional planning patterns for SAS: Challenges

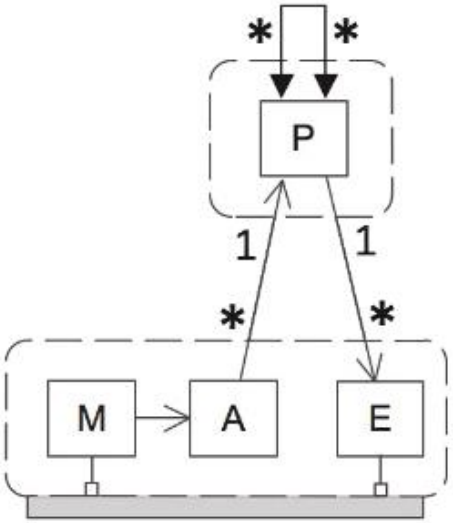
- Different loosely coupled parts of the software (regions) of a complex integrated software system have to realise local adaptations (within a region).
- They also have to perform adaptations that cross the boundaries of the different parts (between regions).
- Typical scenario: Federated cloud infrastructure where adaptations within regions may aim to optimise resource allocation, while the objective of adaptations between regions may be a delegation of certain loads under particular conditions (that owners of regions may not want to expose).
- Alternative scenario: A supply chain management system where partners in the chain have certain local adaptation objectives, while adaptations between partners or system-wide adaptations may aim to achieve some global utility objective.

# Regional planning pattern for SAS (2)

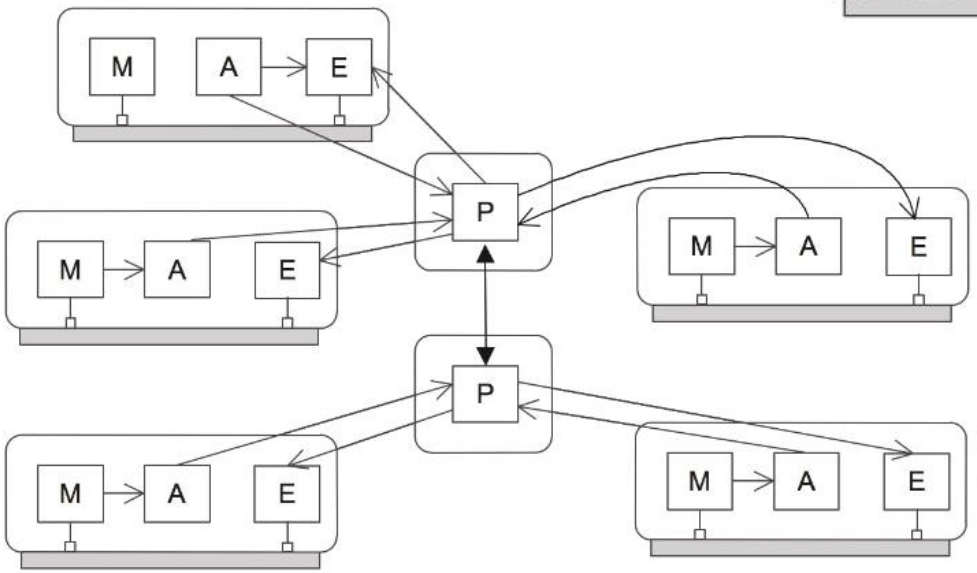


The pattern for regional planning

- Focus: Joint planning decisions




- Particular instance of the pattern:



## Consequences for regional planning

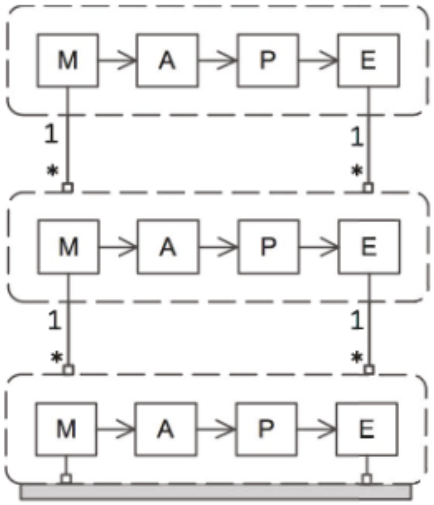
- It enables a layered separation of concerns among different MAPE loops within a single ownership domain.
- Several MAPE loops delegate the planning function.
- For systems that cross the boundaries of ownership domains, regional planner enables a further (flat) separation of concerns for the planning function, where each planner is responsible for the planning of adaptations in its region.
- Local analysis of monitored data may reduce the amount of data and frequency of interactions with the planner.
- Drawbacks:
  - Lack of efficient adaptations
  - Aggregating the results of local analysis and coordinating the planning of adaptations may incur considerable overhead.
  - The pattern may require very detailed planning of the execution of adaptations as it does not support runtime coordination between the ‘E’ components.

## Hierarchical system organisation: Challenge

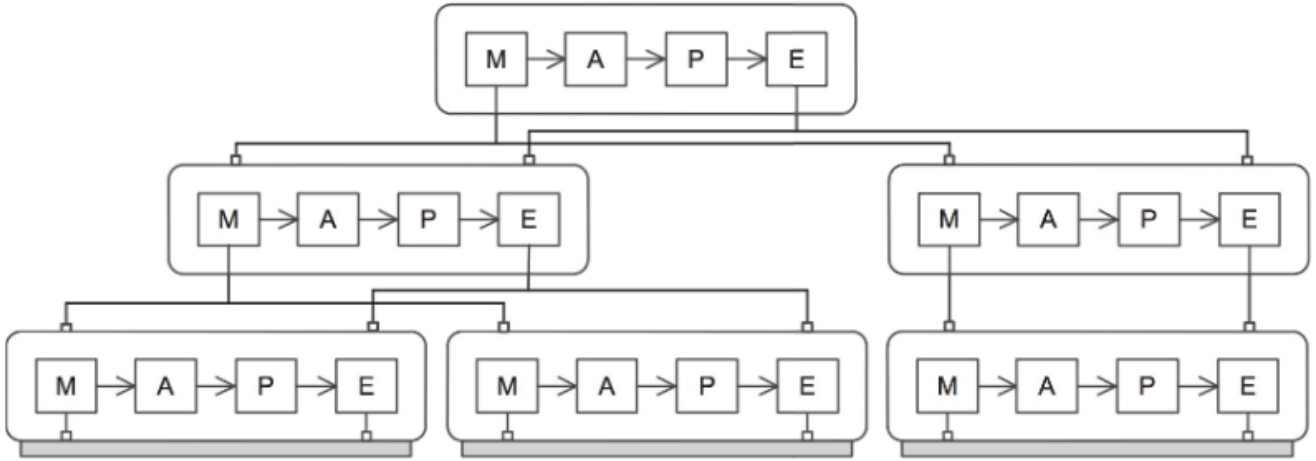
- The control architecture for a complex distributed system may itself become a complex system that needs to be adapted.
- Often: Consider multiple control loops within the same application.
- Loops can be heterogeneous, i.e. they work at different time scales and manage a different kind of resources, and resources with different localities.
- However: Control loops need to coordinate actions to avoid conflicts and to provide certain guarantees about adaptations.
- Examples of such systems are:
  a) Within a single data centre, higher-level control loops are responsible for achieving power consumption or workload goals, whereas local control loops manage workflow distribution between localised subsets of the nodes
  b) Adaptation in pervasive computing environments could be organised into controllers that manage adaptation of human tasks as a user's goals change (in the order of minutes) and controllers that manage particular instances of these tasks to provide fault tolerance (in the order of seconds).

The pattern for hierarchical organisation

- Focus: Abstraction levels

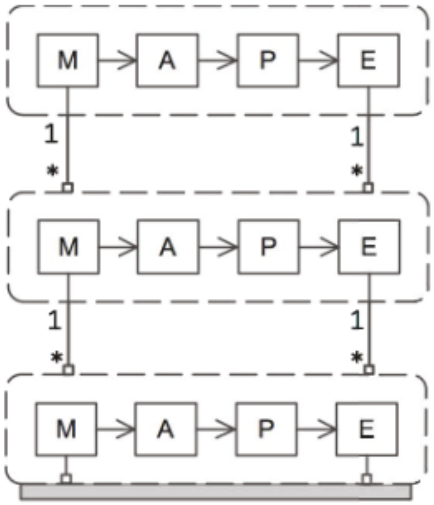



- Particular instance of the pattern:

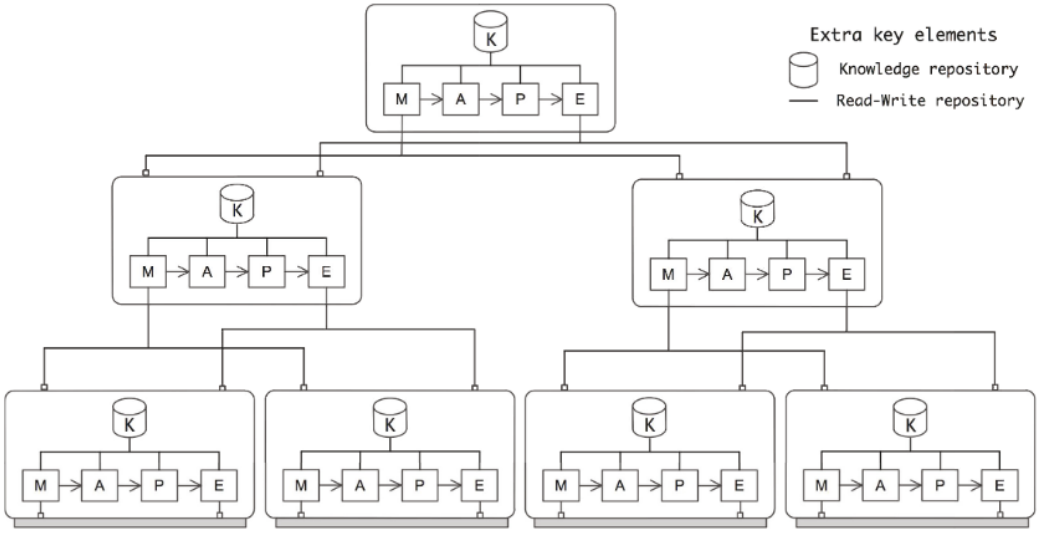


The pattern for hierarchical organisation

- Here: Variant with explicit knowledge models




- Particular instance of the pattern:



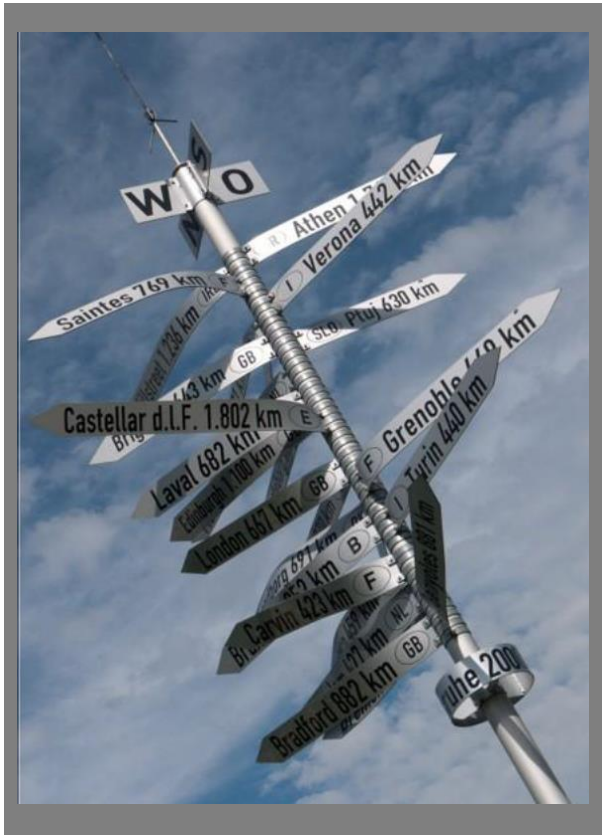
## Consequences of hierarchical organisation

- The pattern structures adaptation logic to reduce complexity.
- It allows bottom layer control loops to focus on concrete adaptation objectives while higher-level control loops have a broader focus.
- Potential trade-offs:
  - The hierarchical decomposition of the adaptation concerns and the allocation of these concerns to different control loops might be difficult to achieve, in particular when goals interfere with one another.
  - All subsystems at the same layer need to have similar tasks, and they have to act at the same abstraction level.
  - From behaviour-based architectures, we know that the design and management of hierarchies with multiple layers can become very complex.
  - Result: There might be no guarantee that the overall solution meets the specifications.

# Agenda

Christian-Albrechts-Universität zu Kiel

- Motivation
- Examples for the growing complexity
- Organic Computing: Nature as inspiration
- Design of Organic Computing systems
- Example: Organic Traffic Control
- Autonomic Computing
- A reference architecture
- Holonic systems
- Conclusion and references



## Structuring systems of systems

- Systems: grow larger, become more distributed and open, become highly heterogeneous and dynamic
- System engineers and administrators must also deal with non-functional properties:
  - scalability,
  - support for diversity,
  - detection and resolution of conflicting goals,
  - cope with openness and with a certain degree of unpredictable change
- Goal: (re)integrate simpler and smaller-scale self-* systems, sub-systems, and components into larger-scale, increasingly complex systems-of-systems; rather than (re)building them from scratch each time
- This system (re)integration process has to be carried-out dynamically and by the system itself; rather than performed offline and manually

Continuous system self-(re-)integration

- Self-integrate available resources
- Discover resources opportunistically
- Depending on goals
- Whenever a change in internal resources, external context, or goals is detected
- Different OC systems:
  - potentially belonging to different authorities,
  - designed separately,
  - achieve different purposes,
  - are executed in shared environments,
  - interfere with each other in unforeseen ways
- Result: complex challenges
  - contention for resources
  - conflicting actions on shared resources

Mastering such open collections raises challenges:

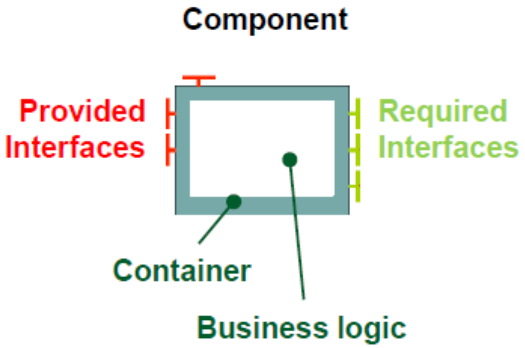
- Scalability: manageable complexity from recycled simplicity.
- Diversity for reusability, adaptability and robustness: able to support the interconnection and co-existence of diverse sub-systems, with specific architectures, control policies, self-* processes, configurations and technologies.
- Interference and conflicts: sub-systems that can achieve their objectives in isolation, may disturb each-other when coupled too tightly.
- Abstracting system descriptions
- Time-tuning self-* processes.

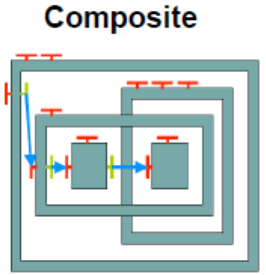
Reference point:

- When everything within and without an intelligent system may change, from its internal resources and integration architecture to its external environment, what is the reference?
- Here: Stakeholders' goal (i.e. the purpose of the system)
- Stakeholder: human developers, owners, users, administrators or other systems.
- Definition: "A goal is an evaluable property that should be achieved or a verifiable statement that can be deemed true (or not), of a state or behaviour of a system under consideration."
- Examples: utility function, user requirements, constraints, etc.

## Divide & conquer

- Component-oriented Software (COS)
  - Encapsulation
  - Limited access through interfaces
  - Containers for non-functional services
  - Composition: “flat” or via encapsulation
- Service-oriented Architectures (SOA)
  - Dynamic service discovery & binding
- Goal-oriented Requirements Engineering (GORE)
  - Translate user goals into technical requirements




## Limitations

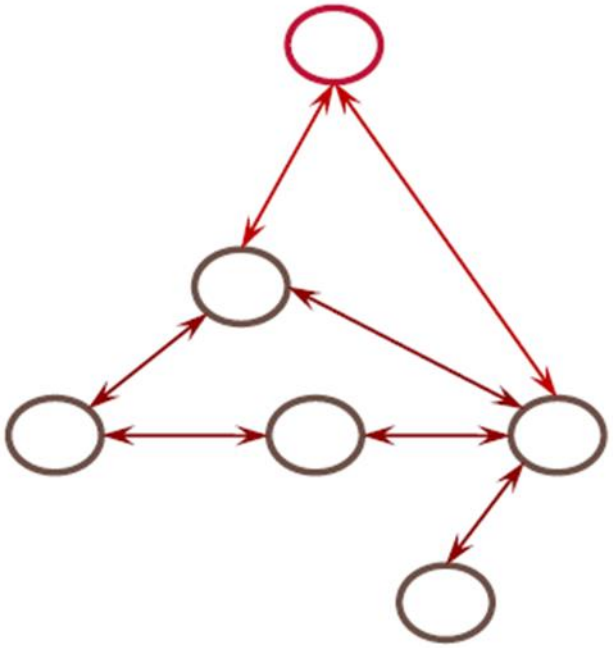
- Wide abstraction gap: from goals to specific service integration
- Brittle: breaks easily when integrating unknown components
- Limited interaction types: reactive request (/reply)

# From MAPE to holonic systems



Support for dynamic change

- Intelligent systems: OC, AC (self-chop), SASO, CAS, etc.
- Control loops for achieving some system property or function
- Coordination of control loops for achieving a global property or function



MAS, AI

- Goal-orientation, autonomy
- Organisations: roles & interrelations
- Dynamic binding of agents to roles
- Dynamic problem solving

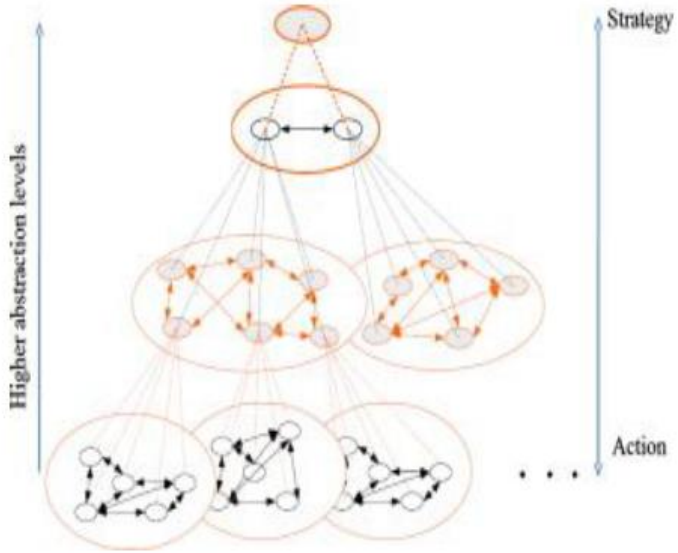
Limitations

- Mostly for achieving a single goal (property or function)
- Difficult to deal with multiple goals, multi-scale goals
  → Need more support for (self-)integration of several self-* systems

## Manage limited rationality

Holonic architectures

- Encapsulated hierarchy: systems made of s
  → For dealing with limited rationality [Simor
- Dual nature: semi-autonomous entities
  → For dealing with individualistic and transi
  behaviour [Koestler67]




Useful properties

- Controlled semi-isolation => limited state-space
  → Optimisation, stability, diversity
- Abstraction, aggregation
  → Progressive divide & conquer (limited rationality)
- Progressive dynamics
  → Avoids or limits chain reactions & oscillations

[Simon62] H. Simon, “The architecture of complexity”. Proc. Am. Philos. Soc. 106(6), 467–482, (1962)
[Koestler67] A. Koestler, “The Ghost in the Machine”, Hutchinson Publisher, London (1967)

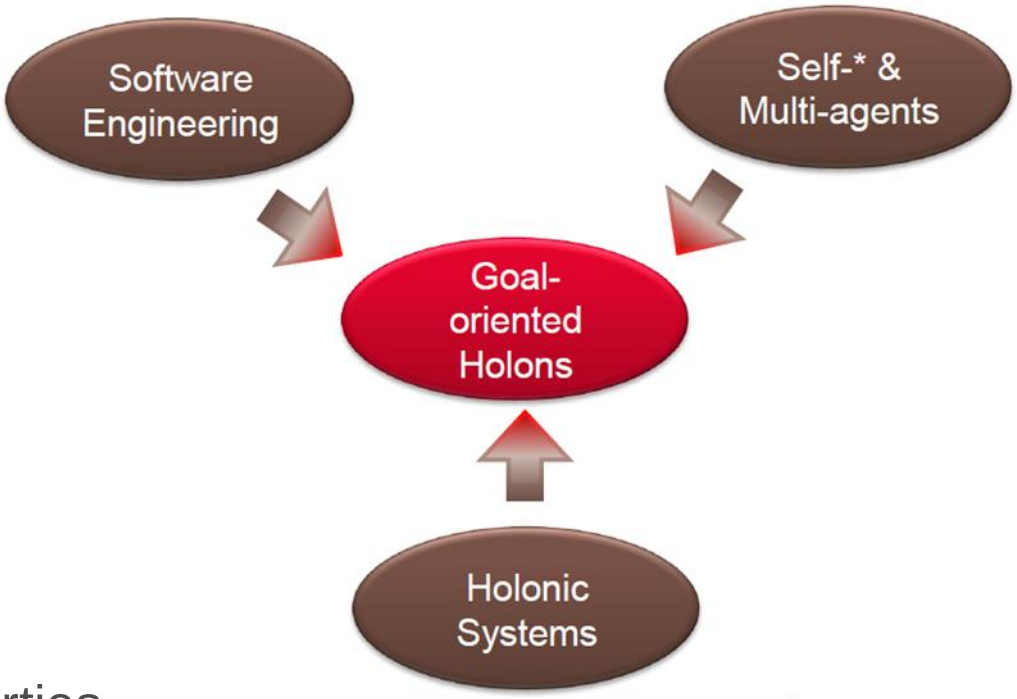
Limitations

- New field
- Little software engineering support
- Only limited experiences
- Application scenarios?

→ Need support for achieving these properties in engineered (self-)integrating systems.

# Goal-oriented holonic architecture



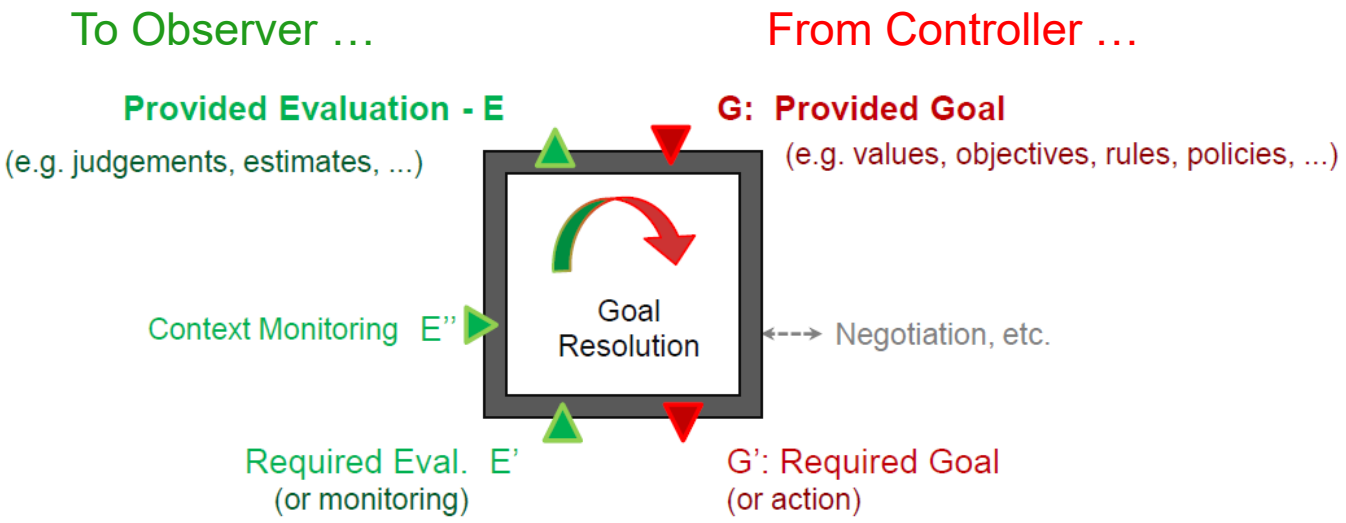



Main features:

- Holonic design and properties
- Self-* functions encapsulated as components / services
- Goal-oriented interactions

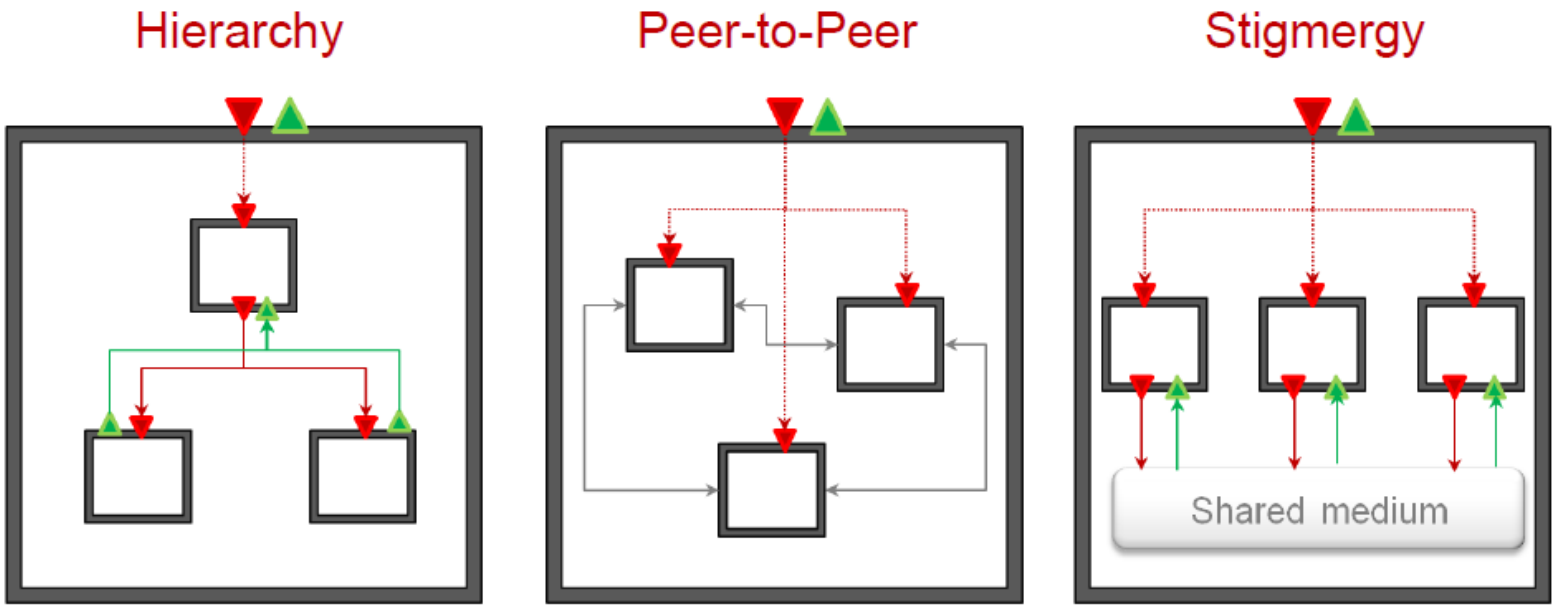
"Classic" Component-oriented SE → Goal-oriented holonic SE

- Interfaces → Goals
- Request / Reply methods → Request / Evaluate goals
- Dynamic service binding → Dynamic Goal matching
- Container → Membrane



## Integration of goal-oriented holonics

- Goal matching, request / reply, execution / evaluation
- Integration patterns



## Basics for goal definitions

- Comment elements:
  - $V$: Viability domain and evaluation function → What?
  - $SR$: Resource scope → Where?
  - $ST$: Time scope → When?

- Details are application-specific and abstraction level-specific

- Examples for the smart house domain:
  - $G_{Cmf}$ = (Comfort, home, forever)
  - $G_{Tmp}$ = ($[T_{min} - T_{max}]$, rooms, intervals)
  - $G_{Ph}$ = ($P$, heater, interval)

## Translation & inverse translation

→ Change the type of a goal element

- $V$: Comfort → Temperature → Power
- $SR$: All electric devices → All TVs and heaters

## Splitting & combination

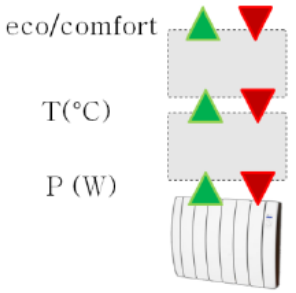
→ Change the values of goal element

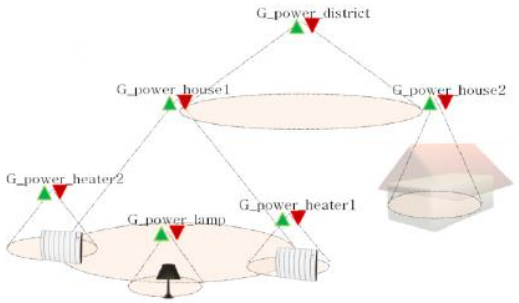
- $SR$: District → Each house → Each device
- $ST$: Forever → Sequential time intervals

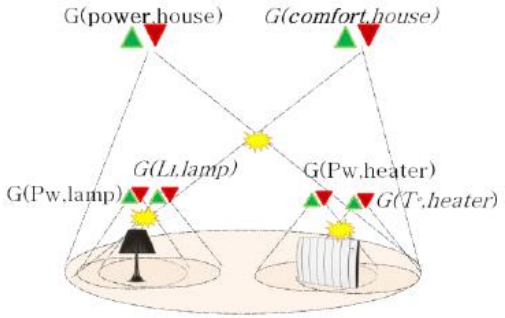
The composition can lead to conflicts!

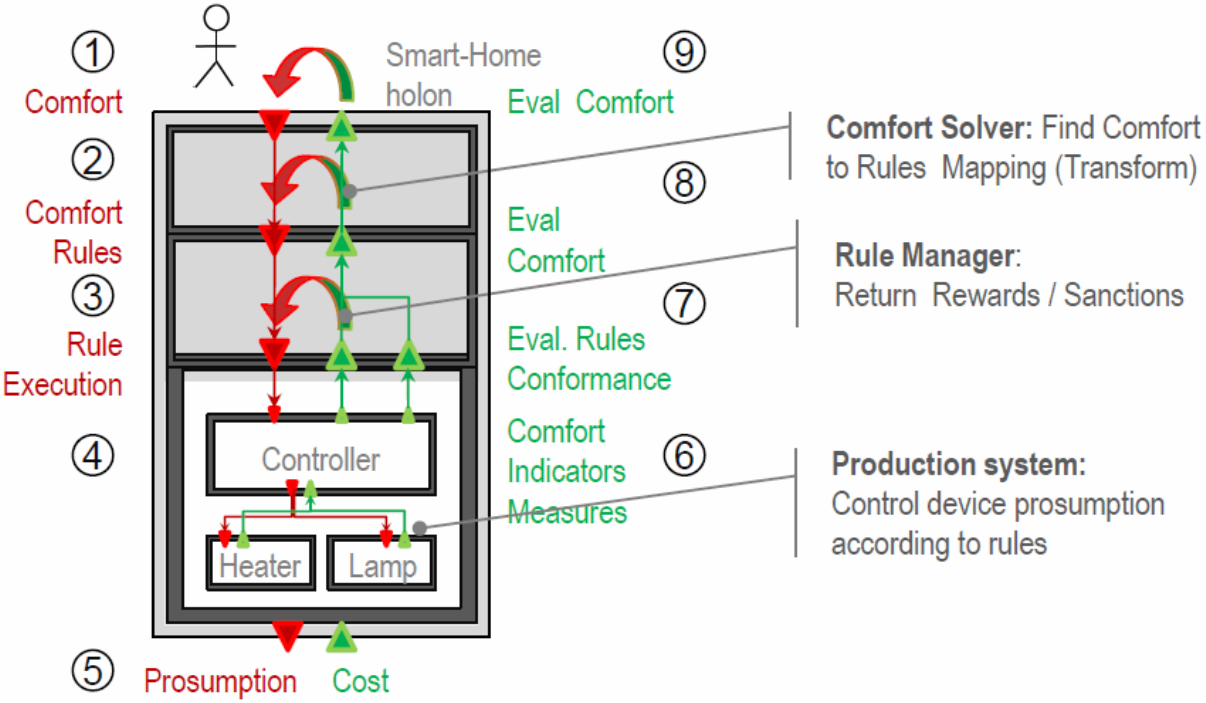
→ When goals with incompatible viability domains have scopes that intersect

- Minimise power consumption vs. maximise comfort
  → Intersection over devices such as heaters and lamps



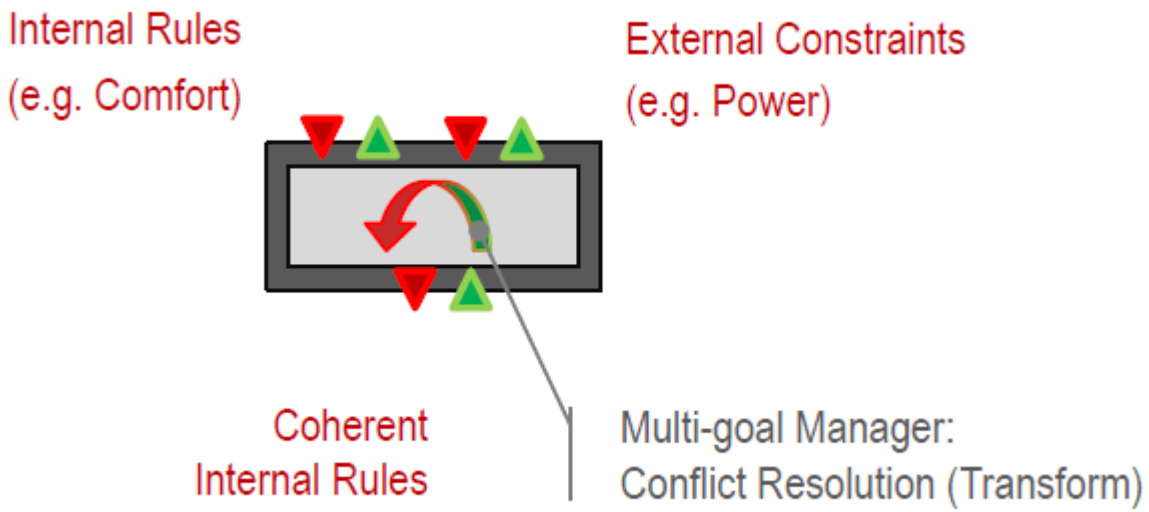



The typical issue addressed:

- Multi-layer translation from Goals to Rules to Rule Enforcement

# Example 2: Multi-goal conflict within a smart home



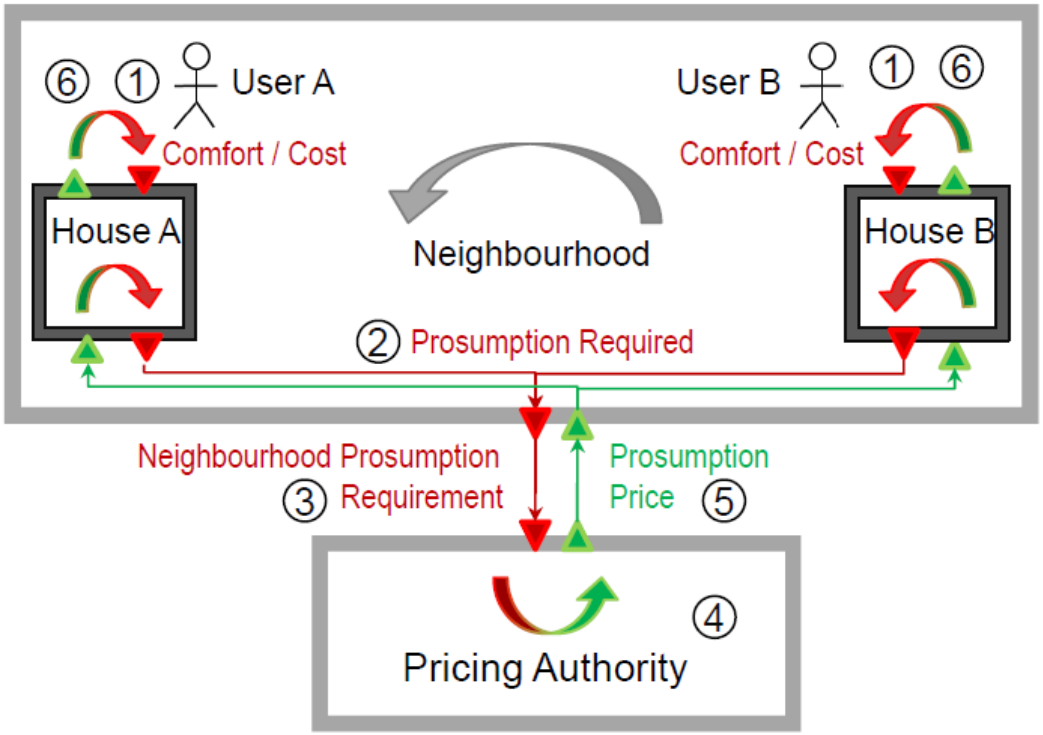



The typical issue addressed:

- Goal conflict resolution (application-specific implementation)

# Example 3: Market-oriented energy distribution



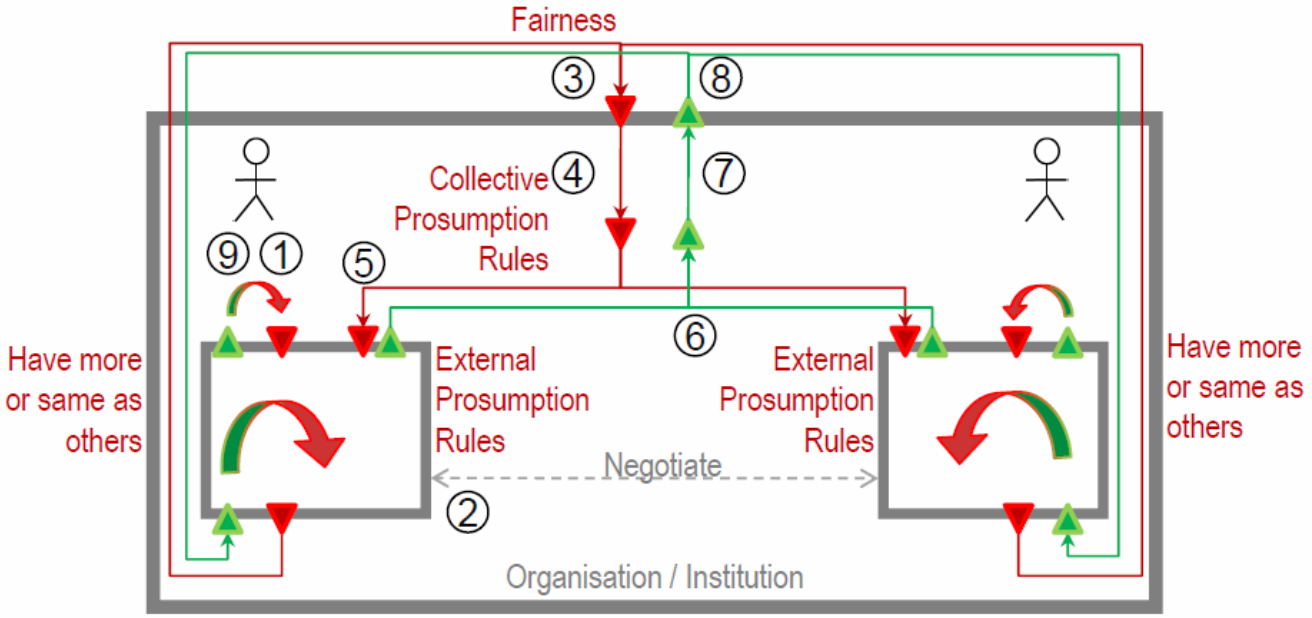



The typical issue addressed:

- Top-down facilitation of bottom-up coordination

# Example 4: Self-governing energy commons






The typical issue addressed:

- Bottom-up goal definition and top-down goal enforcement

# Holonics: Summary



A generic architectural model based on goal-oriented holons

- Goal formalisation → viability and scope
- Goal operations → translation and splitting
- Other interactions → negotiation, coordination, etc.
- Design patterns for conflict resolution →hierarchy, stigmergy, P2P
  => holonic integration
- Specific properties: controlled semi-isolation, abstraction and progressive dynamics

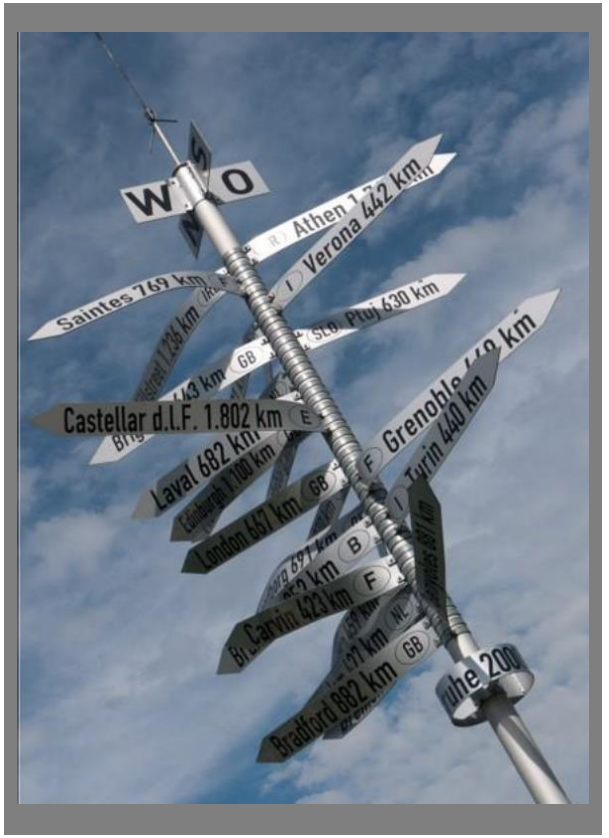
Intended use

- Understand, analyse and define the problem domain
- Design high-level system architecture, or structure
- Communicate, explain and discuss high-level solutions with peers

# Agenda



- Motivation
- Examples for the growing complexity
- Organic Computing: Nature as inspiration
- Design of Organic Computing systems
- Example: Organic Traffic Control
- Autonomic Computing
- A reference architecture
- Holonic systems
- Conclusion and references



## Summary of the chapter

- Standard system design is too limited for current challenges
- Growing complexity can be observed in various examples
- Organic Computing uses nature as inspiration to master complexity
- Design of Organic Computing system uses the Observer/Controller pattern
- Traffic control illustrates the process of an intelligent system
- The ideas of Autonomic Computing are similar to those of Organic Computing
- We developed a reference architecture based on the terminology of AC
- Holonic systems are an extension for system organisation apart from standard hierarchical composition

# References



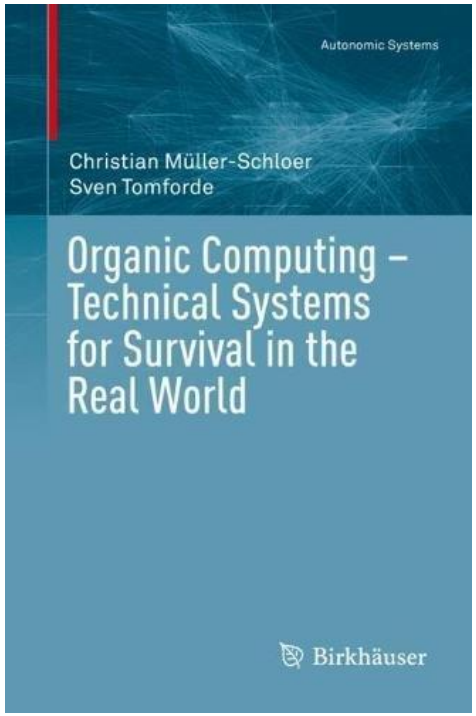
[GS+15] C. Gruhl, B. Sick, A. Wacker, S. Tomforde, J. Hähner: “A Building Block for Awareness in Technical Systems: Online Novelty Detection and Reaction With an Application in Intrusion Detection”. In Proceedings of the IEEE 7th International Conference on Awareness Science and Technology (iCAST 2015), pp. 194 - 200.

[KT+17] J. Kantert, S. Tomforde, R. Scharrer, S. Weber, S. Edenhofer, C. Müller-Schloer: “Identification and Classification of Agent Behaviour at Runtime in Open, Trust-based Organic Computing Systems”. In: Elsevier Journal of System Architecture, vol 75, 2017, pp. 68 -78.

[RT+15] S. Rudolph, S. Tomforde, B. Sick, J. Hähner “A Mutual Influence Detection Algorithm for Systems with Local Performance Measurement”. In: Proceedings of the 9th IEEE International Conference on Self-adapting and Self-organising Systems (SASO15), pp. 144 - 150.

[STH16] M. Sommer, S. Tomforde, J. Hähner: “An Organic Approach to Resilient Traffic Management”. In: Kotsialos, Apostolos; Kluegl, Franziska; McCluskey, Lee; Müller, Jörg; Rana, Omer; and Schumann, Rene (eds.): Autonomic Road Transportation Systems. Birkhäuser Verlag 2015, pp. 113 - 130.

[TM14] S. Tomforde, C. Müller-Schloer: ”Incremental Design of Adaptive Systems”. In: Journal of Ambient Intelligence and Smart Environments 6 (2014), pp. 179 - 198.



- Any questions…?