Inhalt

[1. Introduction 3](#_Toc48502828)

[Motivation 3](#_Toc48502829)

[Beispiele für wachsende Komplexität 3](#_Toc48502830)

[Bekannte Ausfälle mit erheblicher Auswirkung – wegen Komplexität 3](#_Toc48502831)

[Natur als Inspiration 4](#_Toc48502832)

[Wie wird Komplexität in der Natur angegangen? 4](#_Toc48502833)

[Definition Organic Computing 4](#_Toc48502834)

[Aspekte 4](#_Toc48502835)

[Natur als Inspiration 4](#_Toc48502836)

[Systeme 4](#_Toc48502837)

[Terminologie 5](#_Toc48502838)

[OC Systeme bauen 5](#_Toc48502839)

[Paradigmenwechsel 5](#_Toc48502840)

[Grundsätzliche Methoden 5](#_Toc48502841)

[Applikationen 5](#_Toc48502842)

[Related Fields 5](#_Toc48502843)

[Aussicht 5](#_Toc48502844)

[History 5](#_Toc48502845)

[Zusammenfassung 5](#_Toc48502846)

[Fragen 5](#_Toc48502847)

[2. Self-organised Order 6](#_Toc48502848)

[Wassertempel auf Bali 6](#_Toc48502849)

[Selbst-organisierte Ordnung 7](#_Toc48502850)

[Ameisen 7](#_Toc48502851)

[Emergenz 8](#_Toc48502852)

[Beispiele 8](#_Toc48502853)

[Definitionen – Emergenz 9](#_Toc48502854)

[Quantifizierung von Emergenz 10](#_Toc48502855)

[Entropie 10](#_Toc48502856)

[Quantifizierung von Emergenz – ein verfeinerter Ansatz 12](#_Toc48502857)

[Fazit 12](#_Toc48502858)

[3. Quantitatives Organic Computing 13](#_Toc48502859)

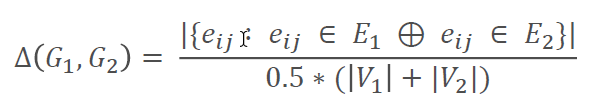
[Motivation 13](#_Toc48502860)

[Autonomie und Selbst-Organisation 13](#_Toc48502861)

[Definitions 13](#_Toc48502862)

[Quantifizierung von Selbst-Organisation 14](#_Toc48502863)

[Selbst-Orga als Prozess 14](#_Toc48502864)

[ 15](#_Toc48502865)

[Der Überlebenszyklus eines Organischen Systems 15](#_Toc48502866)

[Robustheit 15](#_Toc48502867)

[Autonomie 15](#_Toc48502868)

[Schlussfolgerung 15](#_Toc48502869)

# 1. Introduction

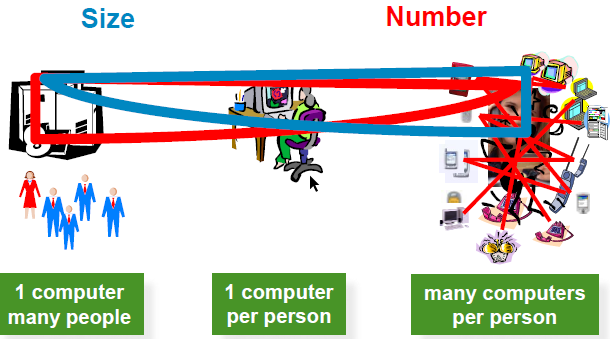
Ziele:

Studenten sollen folgendes können:

* Motivation für OC beschreiben
* Beispiele für steigende Komplexität in Informationssystemen geben
* Erklären wie natürliche Prozesse als Inspiration dienen
* Zusammenfassen welche Aspekte von OC gedeckt werden

## Motivation

Trends: Immer mehr Geräte, immer stärkere Rechner, Fehler durch gegenseitige Beeinflussung



Beobachtung: Hohe Komplexität führt zu Fehlern, verbundene Systeme schwerer zu managen. Unmöglich, alle Situationen vorherzusagen, da dynamische Umgebung und unbekannter Configspace

## Beispiele für wachsende Komplexität

* Moore’s law: Komplexität integrierter Schaltkreise mit minimalen Kosten verdoppelt sich alle 2 Jahre
* Hosts at the internet: `94 0, `07 500 Mio, `15 1 Milliarde durch IP erreichbare Hosts
* Request for Comments
* Glass‘ law: IT-Komplexität hängt indirekt mit Funktionalität zusammen – 25% funktionelles Wachstum erhöht Komplexität um 100%

## Bekannte Ausfälle mit erheblicher Auswirkung – wegen Komplexität

* Skype 2007: Sicherheitskritisches Update von Microsoft hatte Reboot vieler Computer weltweit zu Folge. Ergebnis: Distributed Denial of Service
* Google Mail 2009: Wartungsarbeiten in europäischem Rechenzentrum hatte Verantwortlichkeitsverteilung an umliegende Zentren zufolge. Ergebnis: Kaskadierende Ausfälle von Computern und Rechenzentren.
* EON 2006: Hochspannungsleitung wurde für vorbeifahrendes Schiff abgeschalten, Stromausfälle in ganz Europa

Traditionelle Konzepte, Prozesse und Methoden müssen abgeschafft werden. Die beobachteten Fehler, Ausfälle und der steigende Administrationsaufwand bestätigen das.

## Natur als Inspiration

Komplexität

* … ist ein gewöhnliches Naturphänomen
* Viele gut angepasste Lösungen beobachtbar
* Gemeinsamer Nenner: Komplexitätsminderung durch Zusammenarbeit autonomer Entities

Natürliche Systeme

* … sind selbst hoch komplexe Systeme
* … entwickelten sich über Milliarden von Jahre
* … zeigen self-\* properties
* … sind veränderbar, flexibel, robust gegen Störeinflüsse, belastbar, optimiert

Inspiration ist nicht Nachahmung, sondern Transfer grundlegender Mechanismen

### Wie wird Komplexität in der Natur angegangen?

* System besteht potenziell aus riesigen Sammlungen von Individuen
* Agieren selbst ohne zentrale Kontrolle
* Jeder hat self-\* Eigenschaften: self-learning, self-adaption, self-protection
* Entscheidungen werden basierend auf local knowledge getroffen
* Interagieren und kooperieren miteinander -> Makrolevel verhalten
* Manche Eigenschaft haben Emergenz Effekt
* Jedes Indivuum ist self-motivated und folgt eigenen Zielen

## Definition Organic Computing

* Paradigmenwechsel in Syseng
* Ziele: Designtime- zu Runtimeentscheidung, von Ingenieur- zu Systemverantwortlichkeit
* Viele kleine statt riesiger Strukturen
* Erlaubt autonome Entscheidungen
* Diese basieren auf lokaler Kenntnis, ohne Zugriff auf globale Welt
* Verteilte Entitys interagieren und kooperieren
* Feedback Mechanismus für jede Entity, um zu lernen
* Systeme bestehen aus autonomen Subsystemen: Diese haben Aktuatoren und Sensoren, interagieren miteinander und Umgebung, keine globale Kontrolle nötig
* Die resultierenden OC Systeme müssen: sich organisieren, adaptiv sein und optimales Verhalten lernen

## Aspekte

### Natur als Inspiration

* Selforga und adap meistert Komplexität -> Prinzipien transferieren

### Systeme

* OC ist Systemengineering, definiere das System und seine Orga, kläre die Komplexität

### Terminologie

* Spezifiziere Grundterme wie Selforga, Robustheit, Autonomie

### OC Systeme bauen

* Konzepte für individuelle, kontextbewusste Systeme mit organischen Eigenschaften bauen,
* Makrosicht auf Ökosysteme

### Paradigmenwechsel

* Von Designtime zu Runtime
* Resultierender OC Meta Design Prozess

### Grundsätzliche Methoden

* Spezialisierte Technologie wird für self\*-Properties gebraucht
* Am Wichtigsten: Autonomes lernen und Wissensmodellierung in Laufzeit
* Optimierungstechniken, Interaktionschemata, gemeinsame Einflüsse erkennen

### Applikationen

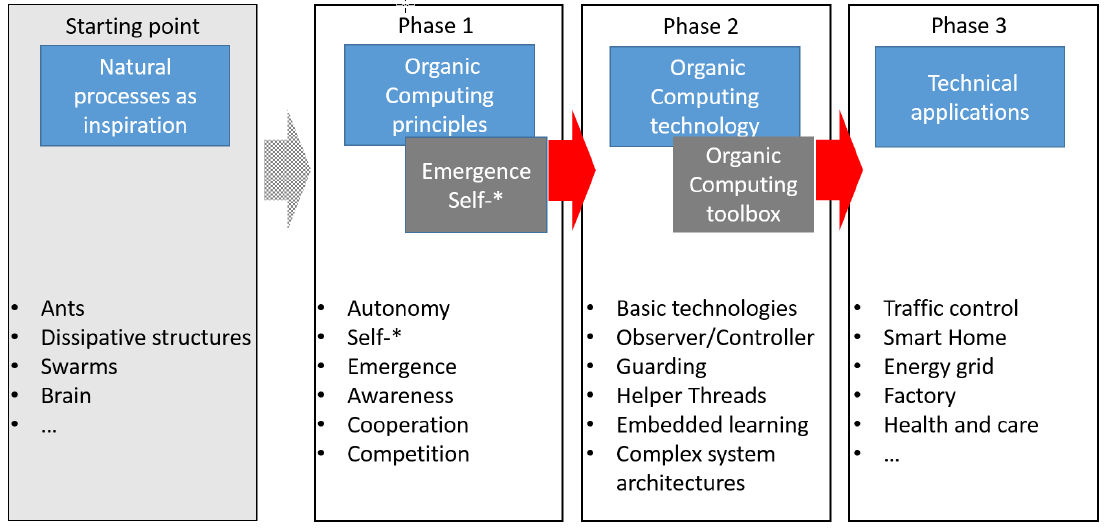
* Technische Systeme erfüllen bestimmten Zweck
* Verkehr, Datenkommunikation, Energienetz, distributed computing, smart cameras

### Related Fields

### Aussicht

* Trend mehr Komplexität
* Jetzige und künftige Herausforderungen für OC Forschung?

## History



## Zusammenfassung

* Trend zur allgegenwärtigen Vernetzung, Komplexität, Outages, Von der Natur lernen, worum geht’s bei OC, wichtige Aspekte

### Fragen

* Was ist die Motivation bei OC?
* Beispiele für steigende Komplexität in Informationssystemen?
* Warum können natürliche Prozesse als Inspiration hinsichtlich Komplexitätsbewältigung dienen?
* Welche Aspekte werden von OC abgedeckt?

# 2. Self-organised Order

## Wassertempel auf Bali

Bewässerungssystem für Reisanbau

* Hauptfaktoren sind Wasserzirkulation und Wechsel zwischen Feucht- und Trockenperioden
* Ziele sind gute PH-Werte, Mikroorganismen, Mineralstoffverteilung, Unkrautvernichtung, Schädlingsbekämpfung, Temperaturstabilisierung
* Problem: Permanente Bewässerung führt zu sehr hohem Wasserbedarf, deshalb muss das Wasser geteilt werden
* Die Farmer wollen ihre Ernte maximieren und müssen hierzu folgende 4 Dinge sequenziell aufeinander abstimmen: Erntephase, Pflanzenart/gruppe, Bewässerung, Trocknen

Wie stellt man optimale Sequenz fest?

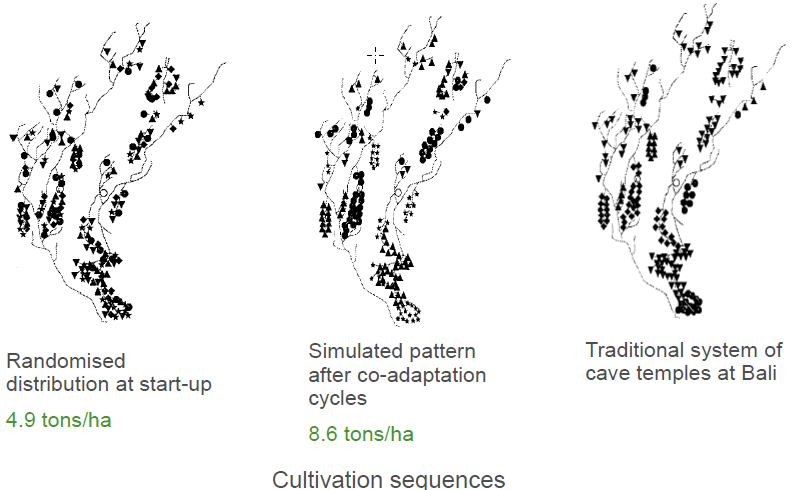
* Trial and error vs. Planning, globale oder lokale Koordination, Gesamtlösung für local brauchbar?

Hypothese:

* Koordinationsalgorithmus, synchron, lokal, wie der beste Nachbar, Co-Adaption

Hypothese verifizieren – Simulation:

* Die Ernte wird als Funktion der Kultivierungssequenz und der Umgebung modelliert.
* Zufällige Initialisierung als Start



Welche Schlüsse kann man ziehen?

* Netzwerk enthält lokale und self-motivated Famer (nodes, agents)
* Kooperation führt zu globalen „optimalen“ Mustern -> Emergenzeffekt
* Bottomup entwickelnde Netzwerke sind adaptiv und reagieren auf Umgebung (zb Regenarmut)
* Erfolg hängt von der Fähigkeit der Nodes lokale Info zu sammeln und darauf zu reagieren ab
* Keine zentrale Autorität nötig: Dezentrales System
* Keine externe Autorität nötig: Selbst-organisierendes System
* Co-adaption verlangt hohe Agentenpopulation
* Agenten entscheiden selbst aber koordinieren sich regelmäßig mit Nachbarn: Semi-Autonom
* System lernt und passt sich in Evolutionszirkeln an
* Zufällige Variationen bei Evolutionsschritten

### Selbst-organisierte Ordnung

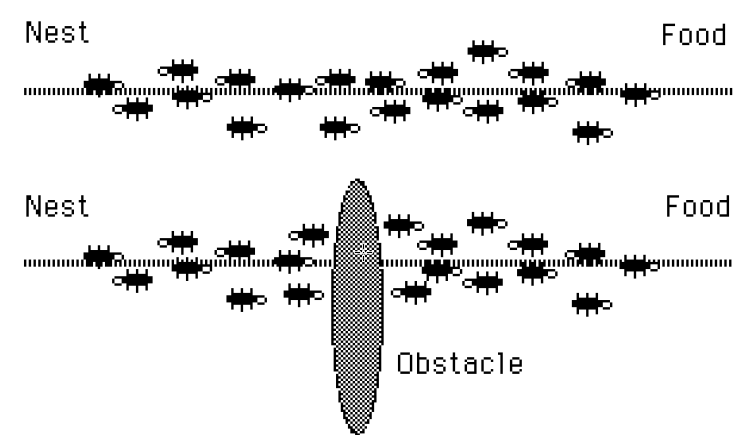
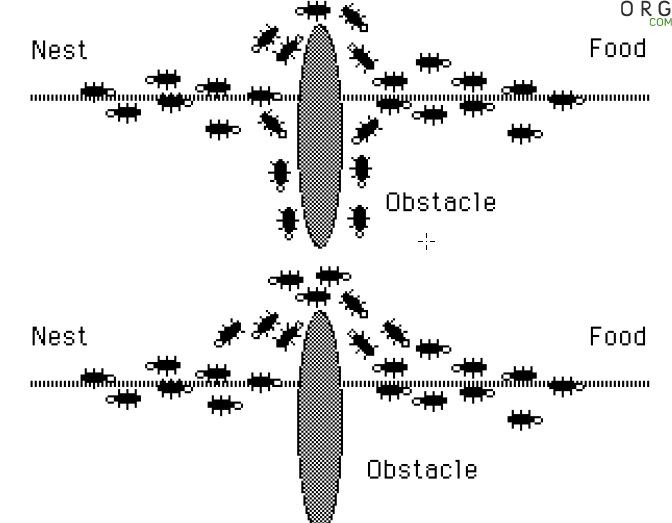
* Strukturiertes System mit globalem Muster entsteht
* Niemand ist dafür verantwortlich, hat globale Sicht oder plant

Mikrolevel: Farmer mit lokaler Sicht, die miteinander interagieren -> Aktionen beeinflussen Makrolevel

Makrolevel: Globale Sicht, Struktur, Muster, „optimale“ Lösung

## Ameisen

* Verteilen Arbeit: Soldaten, Bauarbeiter, Gärtner, Fortplanzung
* Arbeiten zusammen, um Brücken zu bauen
* Bauen Pilze an für Nahrung
* Finden den kürzesten Pfad zum Futter
* Priorisieren anhand Distanz und Erreichbarkeit
* Dynamische Adaption teilnehmender Ameisen, basierend auf Anzahl, gelagertes Futter, nahe Futterquellen und Rivalen

* Ameisen erkunden kurzen und langen Weg, aber finden trotz Kurzsichtigkeit den kurzen
* Folgen dem Weg mit stärkster Pheromonkonzentration, der kürzeste Pfad ist ihnen egal

## Emergenz

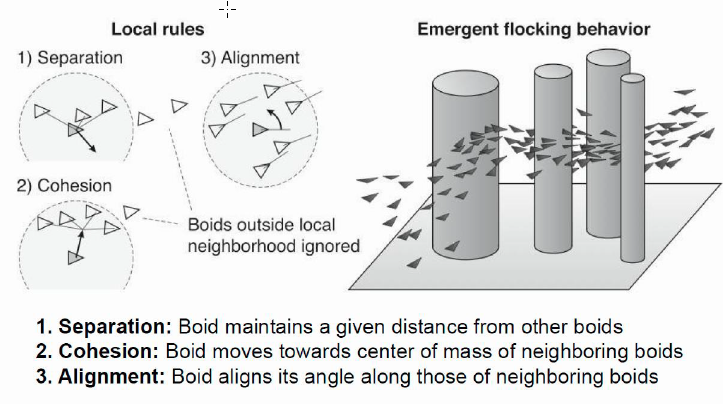
Etwas entsteht einfach – Kürzester Pfad, Schutz, Rollenverteilung, Gärten.

Diese Dinge können nicht anhand einzelner Entitys vorausgesagt werden. Sie entstehen durch Multiple, autonome, homogene, selbstmotivierte und selbstorganisierte Individuen, die miteinander kommunizieren. Keine zentrale Autorität

### Beispiele

**Schwärme**

Fisch- oder Vogelschwärme agieren als Organismus ohne Anführer. Fische wirken als riesiger Fisch, Vögel sparen Ressourcen.



**Termiten**

Bauen Strukturen mit verschiedenen Kanälen und Räumen ohne Einzelintelligenz oder Plan. Emergenz durch lokale und globale Interaktion. Um sicherzugehen, dass sich die Struktur an lokale Gegebenheiten anpasst, ist ein random Verhalten der Individuen von Nöten.

**Menschliches Gehirn**

* Mikro-Level: Neuron, sehr einfaches Verhalten, single Bit
* Makro-Level: Hirn, milliarden Neurone, unendlich anspruchsvolles, kompliziertes Verhalten
* Das Untersuchen einzelner Neuronen kann das Gesamtbild nicht vorhersagen!

**London Millenium Bridge**

Starke und stabile Brücke, die kurz nach Eröffnung wegen starker Schwingung geschlossen werden muss.

* Fußgänger haben sich unbewusst in Phase und Frequenz zueinander und zur Brücke synchronisiert und so für starke Schwingungen gesorgt

**Schweißroboter**

* Mehrere Roboter in Fabrik, je höher Spannung desto besser Schweißnaht
* Roboter schweißen simultan, Spannung fällt und führt zu Defekt
* Spannungsleitung wurde überwacht und Roboter dürfen nur schweißen, wenn Spannung hoch, das machte Problem noch schlimmer da alle synchronisierten
* Random delays eingebaut

**Festplatten**

* Simultane Datenbanksuchen führen zu vibrieren mehrerer Festplatten in einem Serverrig
* Vibrationen synchronisieren sich miteinander und führen zu Störungen

**Verkehrsstau**

* Individuelle Fahrer reagieren auf lokale Kleinigkeiten mit Delay, wirkt sich auf ganzen Verkehr aus

**Gehweg**

* Uni Michigan wollte Gehweg bauen aber wusste nicht wie, deshalb hat sie nur Gras gepflanzt und hat Schüler Wege selbst gehen lassen. Massenverhalten führte dann zu optimalem Pfad

**Sonstige**

* Social Networks, Milgram Experiment (Average number of hops bis Brief target erreicht, 6), Bacon-Number(shortest path Kevin Bacon <-> any actor: 3), Erdös Number (Distance in graph of paper coauthors, 4.7 und 7.3), 6 Hops zwischen Kebapbesitzer und Marlon Brandon.
* Average Number bei solchen Networks ist 6

## Definitionen – Emergenz

* Wie Makroverhalten aus Mikroverhalten wächst
* Emergente Einheiten (Eigenschaften oder Substanzen) wachsen aus fundamentaleren Einheiten aber sind nicht auf sie zurückzuführen
* Emergente Eigenschaft ist nicht nur von Systemelementen beschrieben, sondern von der Interaktion dieser
* Man kann sie nicht aus den einzelteilen zusammenrechnen

**Emergenzquellen:**

* Synchronisation, Kommunikation, nichtlineare Elementinteraktion, Competition über rare Ressourcen(trashing), chaotisches Verhalten (Emergenz als Nautrlösung um mit Chaos umzugehen,

Loops, adaptive Elemente

**Mit Emergenz umgehen:**

* Quellen eindämmen, System limitieren um negative Emergenzeffekte zu verringern
* Stabilität und Robustheit erhöhen
* Mit Simulationen Emergenz frühzeitig erkennen, nicht-linearität verringern
* Ressourcen erhöhen

**Ziel:**

* Positive Effekte nutzen, negative eindämmen

## Quantifizierung von Emergenz

* In OC Systemen können wir keine Emergenz designen, sie entsteht aus Self-Orga Entities
* Wenn das passiert, müssen wir achtsam sein, wegen positiv negativ Effekten
* Wir müssen die Emergenz messen
* Strukturelle Emergenz zeigt sich als Muster in Zeit und Raum oder Muster auf Systemlevel
* Auf Systemlevel zeigt sichs als Interaktion oder große Anzahl gleicher Individuen
* Muster haben Eigenschaften, die in Individuen nicht existent sind
* Emergenz misst Ergebnis von unbekannten Prozessen (Ordnung)
* Ordnung sagt nichts über Selbstorga aus, Entropie misst die Ordnung

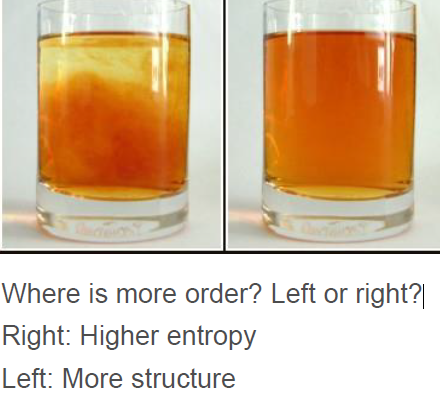
**Ziel:**

* Emergentem System einen hohen Emergenzwert zuweisen
* Emergenz in OC System quantifizieren

**Ansatz:**

* Basis: Verbale Definitionen
* Emergenz = Muster (Symmetrie bricht)
* Muster als Ordnung, Ordnung wird mit Entropy gemessen
* Wir müssen Entropie definieren und zu Emergenz abhängig machen

### Entropie



Ordnung ist subjektiv und hängt von Beobachter ab!

Absicht und Sensorisches Equipment des Beobachters bestimmen die Ansicht.

* System kann abhängig vom Nutzen als geordnet und ungeordnet angesehen werden

Hohe Entropie bedeutet hohe Unordnung

Shannons Entropie ist auch Informationsmaß für zufällige Information in System

**Entropie messen**

* Attribut A wählen mit Values aj, Elemente ei beobachten und Values ai=ei assignen
* Wahrscheinlichkeitsverteilung über Attributvalues aj transformieren
* Für jedes A bekommt man einen Entropiewert, diese aufsummiert ergibt Systementropie

**Definition Versuch 1**

* Emergenz M ist das Abnehmen des Entropy H von Start zu Endzeitpunkt
* Emergenz entsteht aus zunehmender Ordnung
* Prozess, der zu Ordnung führt muss selbst-organisiert sein

Problem: Beobachten der Emergenz bedarf Änderung des Abstraktionslevels, das würde aber zu positiver Entropy führen und verfälscht die Messung.

* Entropiestart und Ende nur vergleichbar wenn selbes Abstraktionslevel

**Definition Versuch 2**

* Emergenz ist Ordnungszunahme aufgrund selbst-organisierender Prozesse zwischen elementen eines Systems
* Hohe Ordnung ist gemessen anhand niedriger Beschreibungskomplexität, also niedriger Informationsgehalt
* Je mehr Struktur ein System hat, desto weniger Info nötig um zu beschreiben
* Eine Entropysteigerung aufgrund eines höheren Abstraktionslevels muss abgezogen werden
* Emergenz M = DifferenzEntropy = Maxentropy – Entropy – differenz der Viewentropy

**Anwendung**

* Emergenz kann für verschiedene Attribute des Systems berechnet werden und dient als früher Indikator emergenter Ordnungsprozesse
* Systememergenz ist nicht selektiv genug, interessanter ist der Emergenz Fingerprint für relevante Attribute
* Welche Attribute sind relevant? Was ist positive und negative Emergenz? Wie können wir Ergebnisse von s.o Prozessen identifizieren?

**Zusammenfassung Prozess:**

1. Entropy für jedes Attribut quantifizieren
2. Emergenz für jedes Attribut Berechnen mit M = HStart – Hend – Hview
3. HView ist mögliche Änderung der Abstraktion wenn man Hstart und End vergleicht, zB Float in int values
4. Systememergenz als Summe über alle Attribute bestimmen und als Fignerprint darstellen
5. Ist Selbstorga verantwortlich?

## Quantifizierung von Emergenz – ein verfeinerter Ansatz

## Fazit

# 3. Quantitatives Organic Computing

## Motivation

Organic Computing bedeutet Konzepte wie Selbstorga in technischen Systemen zu verwenden

* Ergebnisse:
* Designtime Entscheidung wird zu runtime gemoved, System statt Ingenieur trifft Entscheidung
* Hohe Robustheit, weniger Komplexität

Ingenieur muss Systemeigenschaften wählen, Basisinhalte und die benötigten Aspekte quantifizieren.

OC Systeme sind speziell:

* Emergenz möglich
* Selbst managing Prozess, der Objectives oder Utility benötigt
* Haben autonom handelnden Kontrollmechanismus
* System wird nach Störung recovered
* Selbstorga ändert Systemstruktur
* Selbstadaption ändert Systemkonfiguration

## Autonomie und Selbst-Organisation

Sortingstick Example?

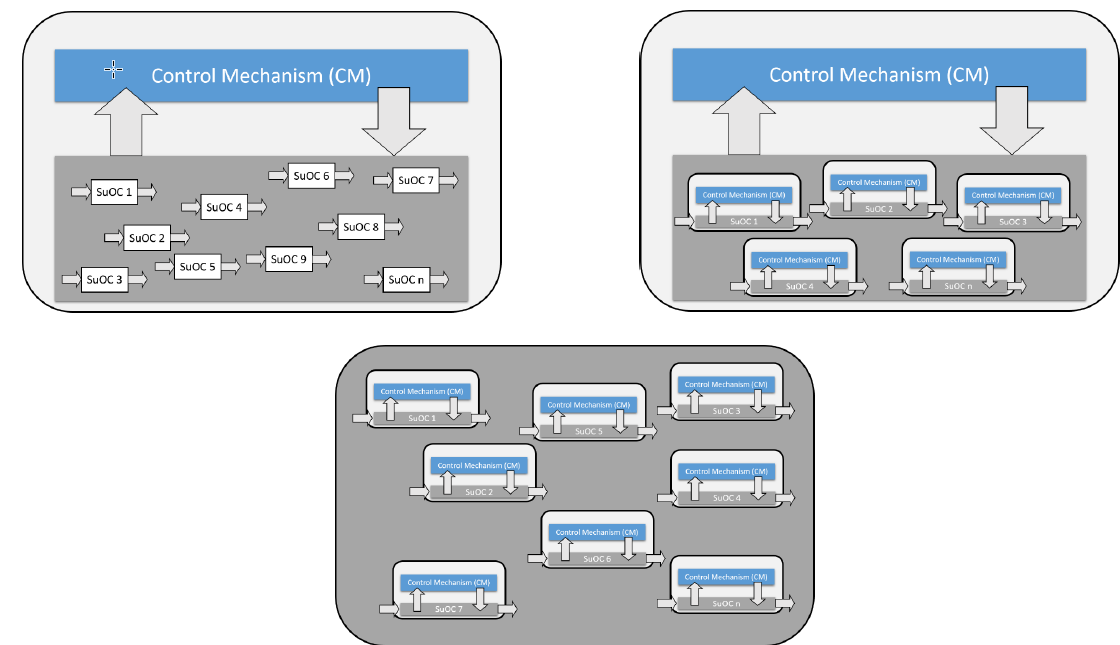
### Definitions

**Autonomie:**

* Ein System ändert die Struktur ohne ausdrückliche externe Kontrolle
* Es muss einen internen Kontrollmechanismus geben!

**Selbst-Organisation:**

* Der interne Kontrollmechanismus ist verteilt!
* Multielementsystem mit wenig bis keiner externen Kontroll zur Restrukturierung (hohe Auto)
* Hat produktive Parts und internal CM, der über Elemente verteilt ist
* CM kann zentralisiert (einer), über alle Elemente oder über hierarchy von CMs verteilt sein

Distribution CM

## Quantifizierung von Selbst-Organisation

Statischer Grad der selbst Organisation

Adaptives System S mit m Elementen, hoher Autonomie und voll oder teilweise verteilte k Controll Mechanisms CM, das zu einem Grad der Selbstorga führt (k:m):

* S ist stark selbst-organisiert, wenn k = m
* S ist selbst-organisiert, wenn k > 1
* S ist schwach selbst-organisiert wenn k = 1

**Term-Unterscheidung**

* Systeme können in Runtime bearbeitet werden (Struktur und Parameter)
* Selbst-Konfiguration -> Parametrisierung
* Selbst-Organisation -> Strukturierung (Komponenten und Verbindungen)
* Selbst-Management -> Beides plus evtl. weitere

**Grad der SelbstOrga**

* Setzt vollen Zugriff auf Designkonzept voraus
* Wird gebraucht, um CM Strukturen zu identifizieren
* Externe Quantifizierung, ohne in Systemlogik und Design einzugreifen braucht anderen Ansatz

### Selbst-Orga als Prozess

* Systemstruktur autonom modifizieren, um bestimmte Funktion zu optimieren
* Nutzt SO Fähigkeit, um auf Events zu reagieren, weniger als permanente Änderung
* SelbstOrga wird beobachtet, wenn System den Organisationsgrad erhöht und Struktur aufbaut, zB bei Systemstart
* SelbstOrga wird beobachtet, wenn System gestört wird und das System seinen vorherigen Grad recovered

**Ansatz**

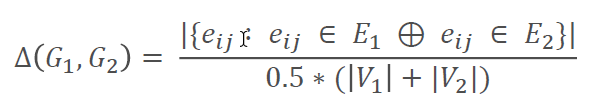
* Innere Werte nicht einsehbar
* Jede einzelne Komponente beobachten
* Beziehung zwischen Komponenten bestimmen Struktur
* Beziehungen werden via Kommunikation gefestigt und geändert
* Beobachte Kommunikation und schließe auf Beziehung
* Graph bauen, mit Nodes und Edges

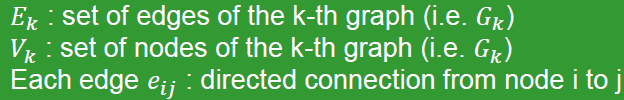
Dieser Graph zeigt uns zwei Aspekte:

* Änderungen: Gab es SO Prozesse zwischen 2 Zeitpunkten?
* Differenz zwischen möglichen und tatsächlichen Beziehungen über Zeit: Schöpft das System das Orga-Potential aus?

**Ansatz**

* Baue Graphen an 2 Timesteps
* Kommunikation muss von außen einsehbar und über Quelle und Ziel verfügen
* Nachrichten sind semantisch unterscheidbar: Strukturbezogene werden zu Edges





Formel repräsentiert selbst-organisation, die im Timeframe zustande kam, allerdings betont sie nur dass IRGENDWAS passiert ist.

* Guter Indikator für SO

Wir möchten aber Potentialsystem mit aktuellem System vergleichen, wie Potential hat das System ausgeschöpft? Potential erfassen durch:

1. Information ist verfügbar wegen Design, Designer spezifiziert mögliche Partner

oder

1. Schätzung bei Runtime

**Schätzmethode**

* Jede Message analysieren und Liste der Interaktionspartner für jede Node führen
* Nach bestimmter Zeit Graph mit dieser Liste builden
* Anstatt wie vorher zwei Zeiten zu vergleichen schauen wir den gesamten Prozess an
* Vergleiche alle Veränderungen (zB Edgenummer) mit dem Referenzgraph um Grad der Selbstorganisation zu bekommen

## Der Überlebenszyklus eines Organischen Systems

Systeme müssen in einer sich ändernden Welt überleben, Verhalten kann als Kreis modelliert werden. Model wird dann für Formalisierung und Metriken benutzt.

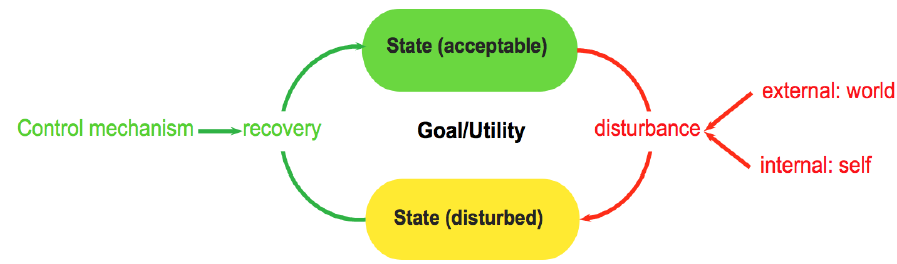
**Initiales Model**

* State Machine, externe Events ändern State
* Automat kodiert eine vorbestimmte Schrittfolge die ausgeführt wird je nach Event.

Es gibt allerdings keine Art, Präferenzen anzugeben, weil Tranisitions deterministisch und States äquivalent sind

**Ein OC Ansatz**

* State Machine aber mit Präferenzen
* Agent hat Ziel zu überleben, manche States sind begehrenswerter
* Interne(zB Agent Malfunction) und externe Events ändern State Z
* State ist neutral und ohne value
* Wir brauchen den Nutzen des States hinsichtlich eines gegebenen Ziels
* Ziele unterscheiden gute und schlechte States, acceptable und nicht
* OC will immer acceptable State
* Störung lässt von acceptable State abweichen, der CM initiiert Recovery Actions um zurückzukommen



**State**

* State ist Vektor Z mit Attributen
* Wir wollen überwachbare Metriken haben, die den State beschreiben(aus System heraus)  
  Beispiel:

Roboter mit GPS, der x und y Koordinaten hat. Ohne GPS könnten wir nur von außen beobachten, somit x y kein Teil des Statevektors

* System ist jederzeit in State z(t)
* Bei n überwachbaren Attributen um S zu beschreiben, ist z(t) ein Vektor in d-Dimensionalem Raum

**Nutzen**

* Statusvektor ist neutral und objektiv gegenüber Wissen von System S
* Er bekommt erst gut oder schlecht wenn man dem State Wert zuweist – via Function n
* Diese mappt State auf echte Zahlen u:

u = n(z)

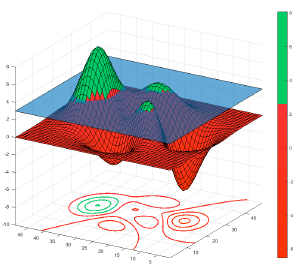
* U repräsentiert Ziele, die CM erreichen will, diese können Maximierung oder Minimierung sein
* U kann aus unterschiedlichen U1, U2 bestehen, basierend auf State Eigenschaften und Kombos, können sich auch wiedersprechen

zB: U ist Anzahl der Falschen Stäbchen beim Ordnen, es wird versucht u=0 zu erreichen

zB: Fahrzeug: ttravel ist Reisezeit, topt ist Optimum, Nutzen wird als U= ttravel-topt definiert

Konflikt könnte weiteres Ziel Spritverbrauch sein

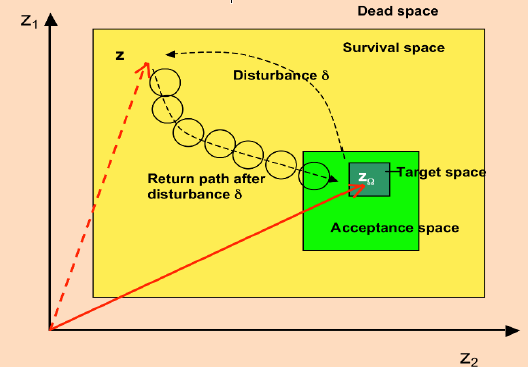
* Utility/Fitness kann die Probleme visualisieren, die Agenten haben

Alle GrünenStati sind akzeptierbar und wollen erreichtwerden

**State Space**

* Ist momentane Utility niedriger als gewollte, muss CM Aktionen anstoßen um System zu optimieren
* Man brauch Schwellwerte, die das Verhalten Kategorisieren

Ziel: States entsprechend Nutzen klassifizieren und verschiedene SS die verschiedene Aktionen vom CM brauchen bereitstellen



T Space: States mit u >= utarget

A Space: U >= UACC

SuS: Von hier aus ist Recovery möglich

DS: Tödlich für System

**Störungen**

* Ändern State zu Disturbed zδ(t) = δ(z(t)). Und Utility uδ= η (zδ(t)).
* Wenn U-D < UAcc ist S nicht mehr in Acceptance Space
* zB Platter Reifen oder blockierte Straße
* Änderung des Evaluationskriterium zB Größe zu Farbe beim Sortieren
* CM muss aktiv sein bis z ist wieder acceptable
* Ein Utilityshift (zB Zieländerung vom User) heißt „flexibility“

OC System ist flexibel wenn der CM das System innerhalb einer bestimmten Zeitperiode mindestens zu einem acc State zurückführt, nachdem ein Utilitychange auftrat

**Recovery**

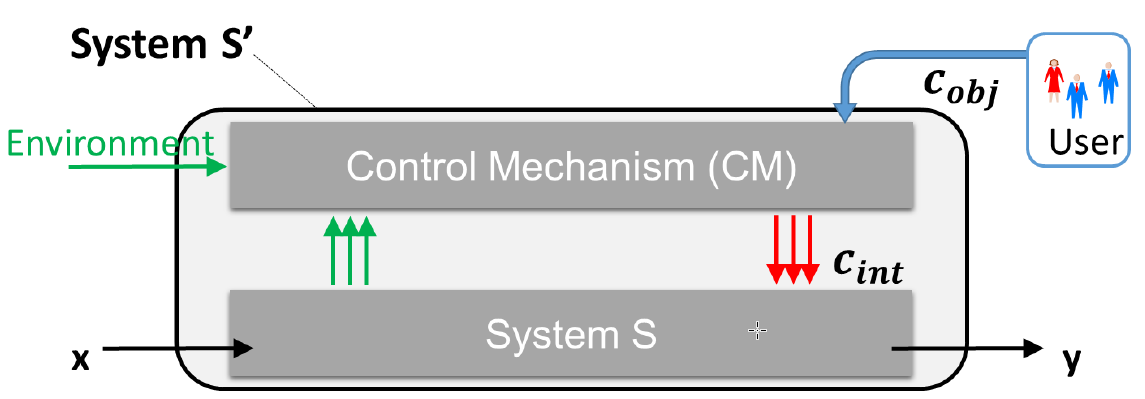
* Utility ändert sich über Zeit, weil State variiert
* CM verantwortlich für Rückführung zu AS
* Utilitydrop ist schnell, Recovery langsam, Charakteristisch für OCSystems

Grund: Prozess ist noisy und wird von Rückfällen unterbrochen, Disturbance kann andauern

* Systemrecovery wird als Bewegung des Statevektors in AS/TS beobachtet (Kreise)

**CM**

* Macht Recovery
* Ändert S Verhalten durch Attributänderung INDIREKT
* System S heißt System Under Observation and Control (SUOC), das ist das produktive System das die Arbeit macht
* Kann Input Signale X erhalten und in OS y transformieren
* ControlSignalint bildet Konfiguration des SUOC
* CM stößt ControlActions an / sendet CSint an S
* S bildet mit CM ein S‘, was wiederum durch höheren CM mit Signalen Cobj kontrolliert wird

****

## Robustheit

Self\*-Eigenschaften um SinglePointOfFailure zu vermeiden und System gegen Störung resistent zu machen

OC Systeme sind nicht per se performanter als normale, aber kehren schneller zu AState nach Störung zurück

* Das wird Robustheit genannt

Robustheit: SystemStatus ändert sich durch internen oder Umweltchange. Bleibt das System acceptable, ist es Robus.

Flexibilität: Evaluation/Akzeptanzkriterium ändert sich, Target und ASpace bewegen sich. Kann System sich anpassen, ist es Flexibel.

**Robustheitsklassen**

* Stark Robust, wenn System nach allen möglichen Disturbances wieder zu Targetspace geht
* Robust, wieder nach AccSpace
* Schwach Robust, wenn nach SurvivalSpace und CM es nach Acc pushen kann

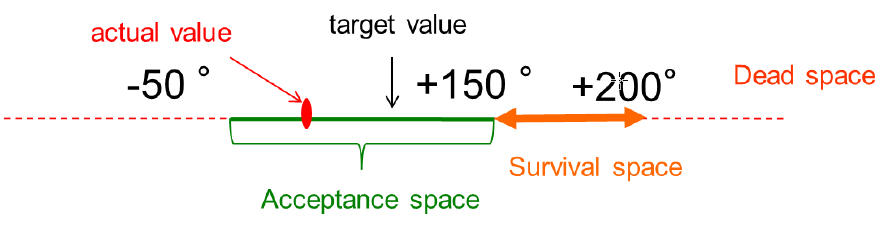
****

Abbildung Schwach robust, mit Ventilator der zw. 150-200 agiert

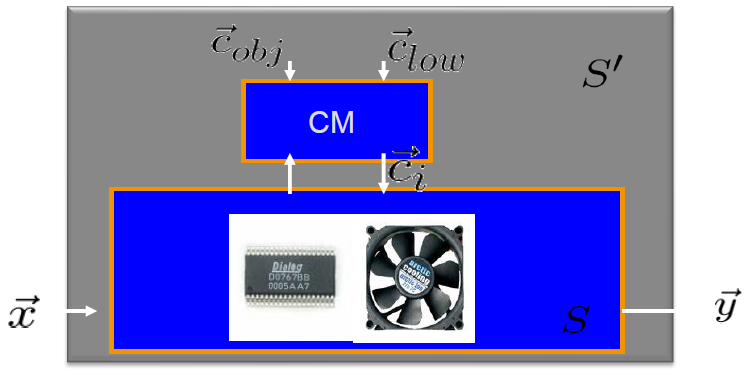


Abbildung 2S ist adaptable, S' adaptiv

### Quantifikation der Robustheit

* Ziel ist, Kontrolle zu schätzen
* Verschiedene CM Designs bezüglich Widerstand vergleichen
* Nutze ‚Area oft he characteristic utility degradation over time‘
* Optimalerweise 0
* Besteht aus 1) Höhe Utility Drop 2) Wiederherstellungsdauer

Der allgemeine Ansatz

* Funktioniert bei extern messbaren Variablen,
* Braucht keine zusätzliche Info
* Unterscheidet zwischen systemeigener (passiver) und hinzugefügter (aktiver) Robustheit
* Kann verschiedene Systeme derselben Probleminstanz vergleichen

Ungestörtes System zeigt Performance.

Es wird mit Utility Measure U bewertet. Bei Störung weicht es von Uacc mit Udiff ab.

Typischerweise haben OC Systeme passive(Phase 1) und aktive(Phase2) Robustheit, zusammen bilden sie die Deviation Phase.

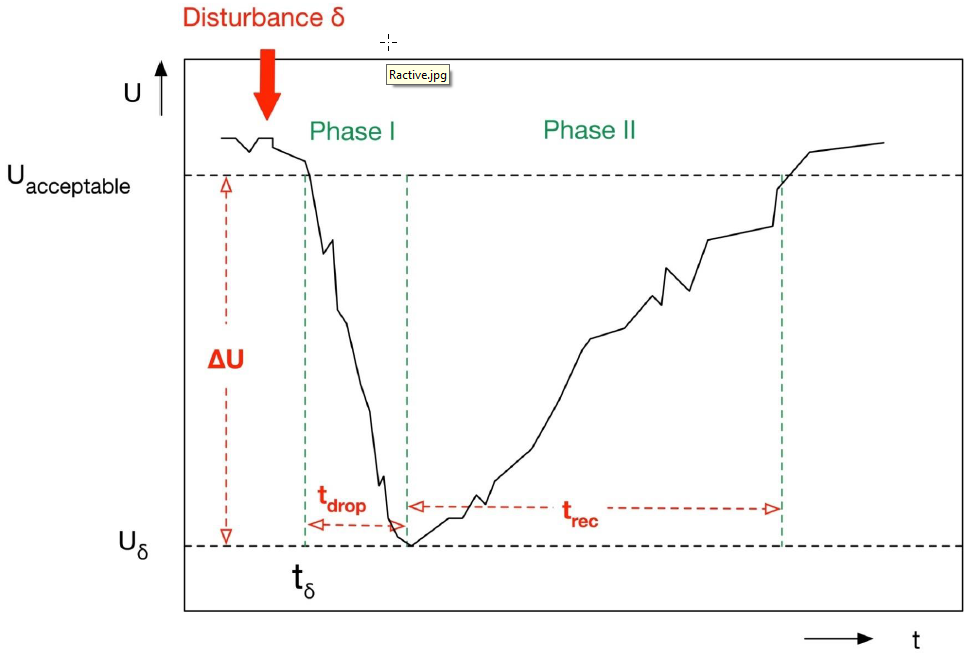
Zum quantifizieren beachten wir:

* Stärke der Störung
* Abfall Udiff der Utility von U acc
* Dauer der abweichung, also recovery time trec

**Robustheitstypen**

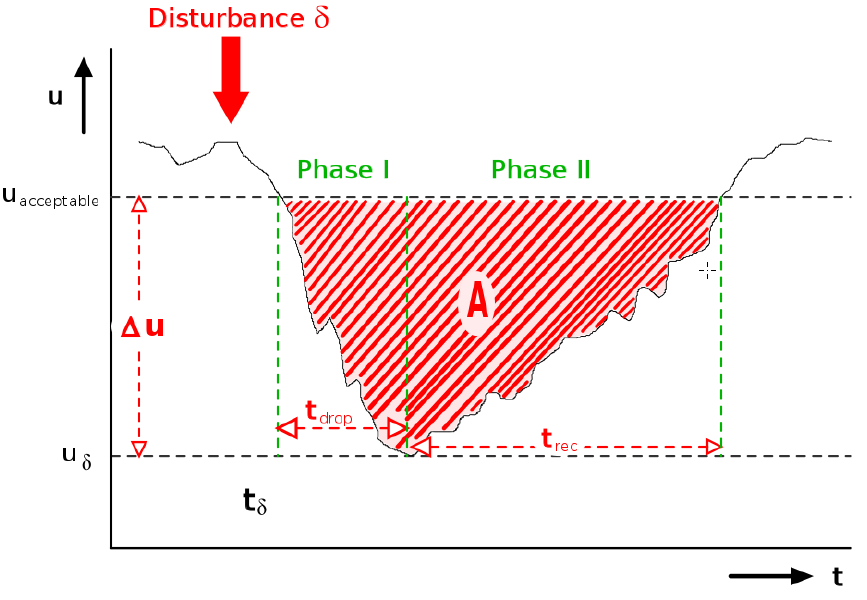
Passiv:

* Reagiert auf Störung durch Deflection Δu= Δx, zB Verbiegen des Fernsehturms bei Wind
* Empflindlichkeit von U gegenüber Störung, Systemstabilität ohne aktiven CM
* σ = du(δ)/dδ
* Ist null, wenn Störung keinen Effekt hat



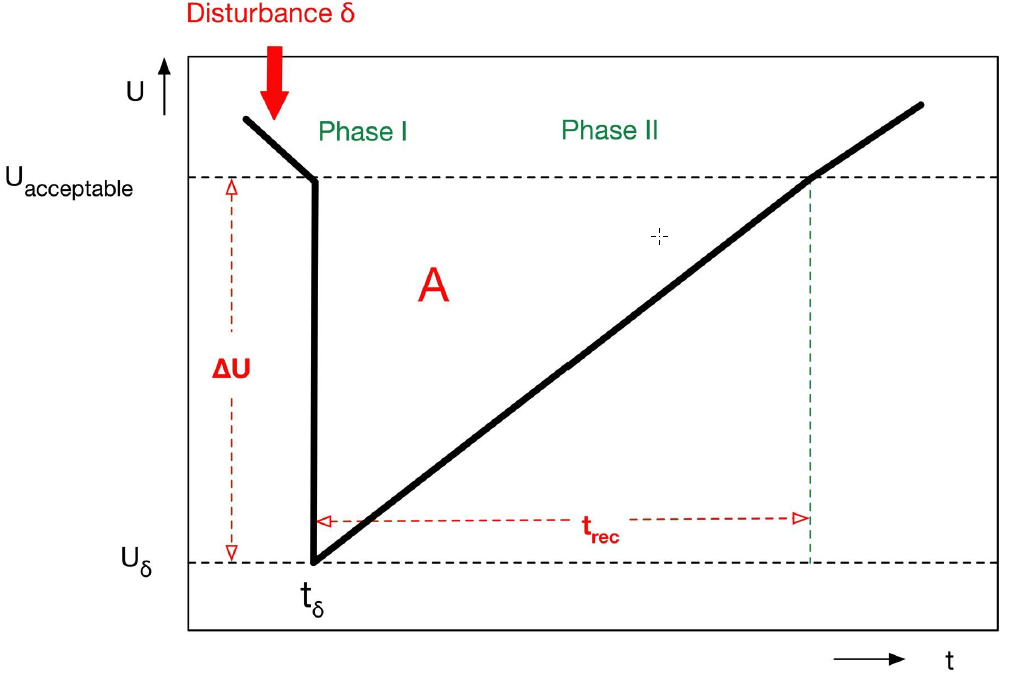
Aktiv:

* Durch Recovery Speed bestimmt
* 
* S Active ist CM Eigenschaft: Udiff/trec
* Ohne CM bleibt System disturbed, mindestens bis D verschwindet
* Recoverytime hängt von Utilitydrop, der Empfindlichkeit und des aktiven Recoveryspeed ab



* Robustheit (δ, ∆u, trec) or (δ, σ, sactive).
* A ist kompletter Einfluss aufs System, sie heißt Utility Degradation DU
* Kann man minimieren durch minimieren von trec + Du

Degradation annähern:



* Für besseres Schätzen und Vereinfachung
* Drop kommt sehr shcnell, deshalb nahe 0
* Utility steigt wieder linear



* Beobachtung: Empfindlichkeit mindern bringt mehr als recovery speed erhöhen
* Grund: Empfindlichkeit beeinflusst DiffU und trec
* Tradeoff zwischen aktiv und passiv möglich

## Autonomie

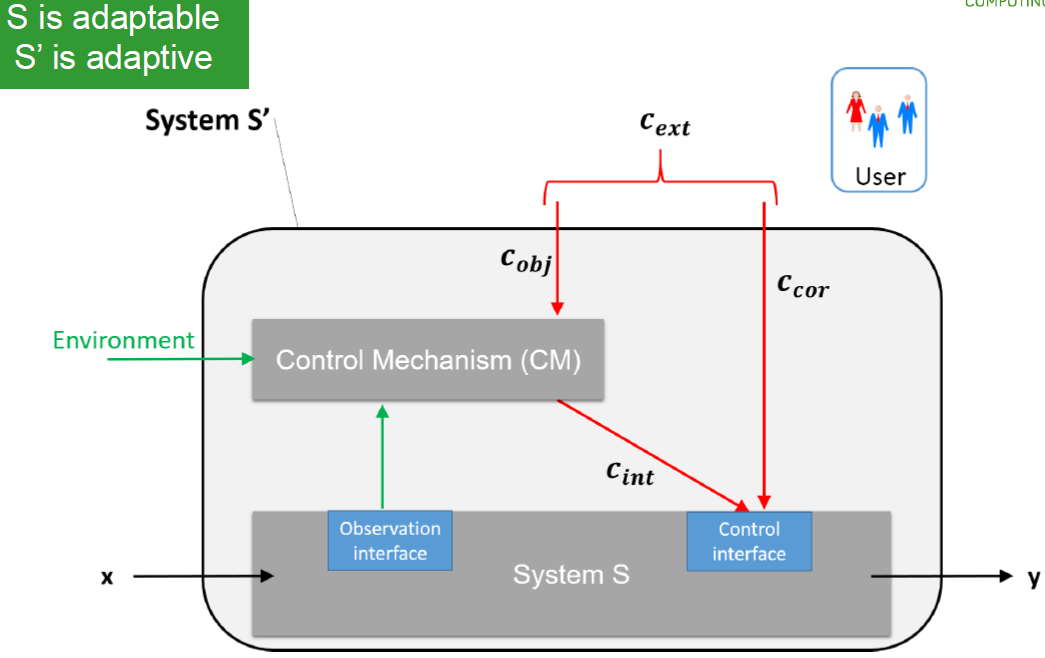
* OCS müssen in ändernder Umgebung überlebe und robust und flexibel sein
* CM, der S und Umgebung überwacht
* CM, der Abweichung vom AkzeptanzS bestimmt
* CM, der Aktionen ergreift um S in Sacc zurückzuholen, der von Cobj(extern; Ziel) bestimmt
* In diesem Sinne überlebende S sind autonom
* Wir wollen stets von außen durch Ziele/Grenzen kontrollieren können

Aber: S mit sowenig Extern wie möglich

* Balance zwischen zu viel (unkontrollierbar) und zu wenig (man muss immer eingreifen) Autonomie
* Semiautonom

### Den CM verfeinern

* S braucht observation und control interfaces für CM
* S stellt bestimmte interne Parameter zur Beobachtung und Veränderung (zb Schwellwerte/Timings) zur Verfügung, es wird adaptable (passive Eigenschaft)
* Aus observierendem und kontrollierendem CM + adapt. Und beobachtbar S wird semiautonomes S‘



Cint : Signale von innerhalb System

Cobj: highlevel Ziele vom User oder höheren Systemen (Einfluss auf S indirekt)

Cext: Externe Quellen, die S direkt beeinflussen

* S nicht autonom, wenn S‘ nur auf cext reagiert

### Variabilität

* Von außen erreichbare Kontrollparameter werden in control vector C zusammengefasst und auf Control Interface angewandt -> eine mögliche Konfig
* Variabilität V = Größe Konfigspace = Größe Vektor C = Bits Vektor C

### Komplexität reduzieren

* Wenn korrekt designed, kommt cobj seltener zum Einsatz als Cint
* Complexity Reduction = Vint – Vobj = #cint - #cobj
* Positive CR: S hat größeren ConfigSpace als S‘: CM erreicht gewünschte CR

### Corrective Control

Perfektes OCS:

* CM kann highlevel ControlS in interne CS umwandeln
* Ziele werden eingehalten
* S strebt nach AccSpace

CC:

* CM braucht externe Kontrolle, falls S nicht AccSpace möglich: Ccorr
* Ccorr als Erweiterung zu Configuration Space von S‘: cobj + ccorr

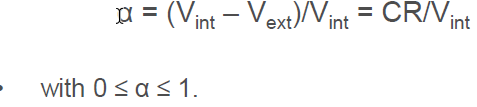
Cext = (Cobj;Ccorr)

* Falls Ccorr: CR = Vint – Vext = #cint - #cext
* Falls viel kontrolliert werden muss wird Vext, negative CR, S‘ schwieriger als S

### Grad der Autonomität

Ziel: Definiere Statischen Grad der A von S‘ als CR, relativ zur internen Variablen Vint

**Statisch**

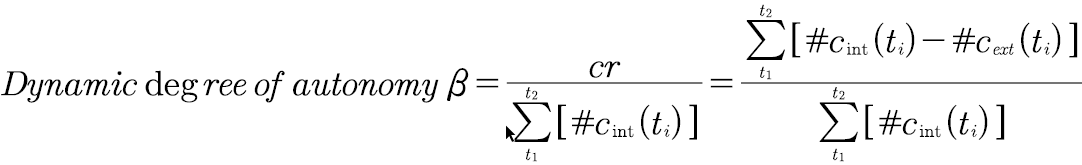
****

A = 0 falls keine CR

a=1 falls Vext = 0 -> S‘ kann nicht von außen kontrolliert werden

**Dynamisch**

* Statisch ist indikator für MÖGLICHE control actions von S und S‘, drückt nicht die tatsächlichen Cas von CM oder extern aus
* Ziel: Kontrollfluss in Zeitspanne messen, der von Control Interface angewandt wird

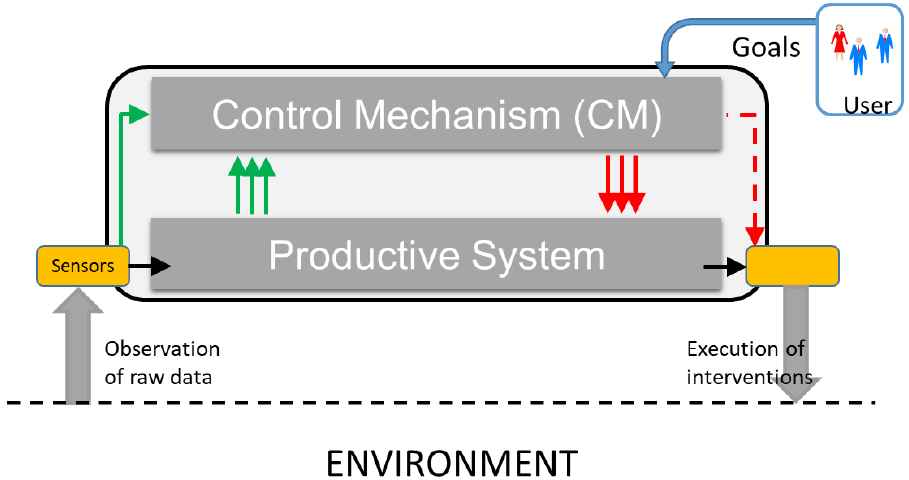


0 heißt Kontrolle von außerhalb, 1 nur innerhalb

# Oberserver Kontroller Architektur

* Generelles Designkonzept für OS
* O/C Arch
* Multilevel O/C framework

## Generelles Designkonzept für OS

Grün Beobachtung, rot Kontrolle

**Systemgrenzen**

* Alles, was nicht direkt zum System gehört ist Environment
* Dies wird mit Sensoren eingesehen und mit Aktuoren beeinflusst

**Mensch außer Kraft setzen**

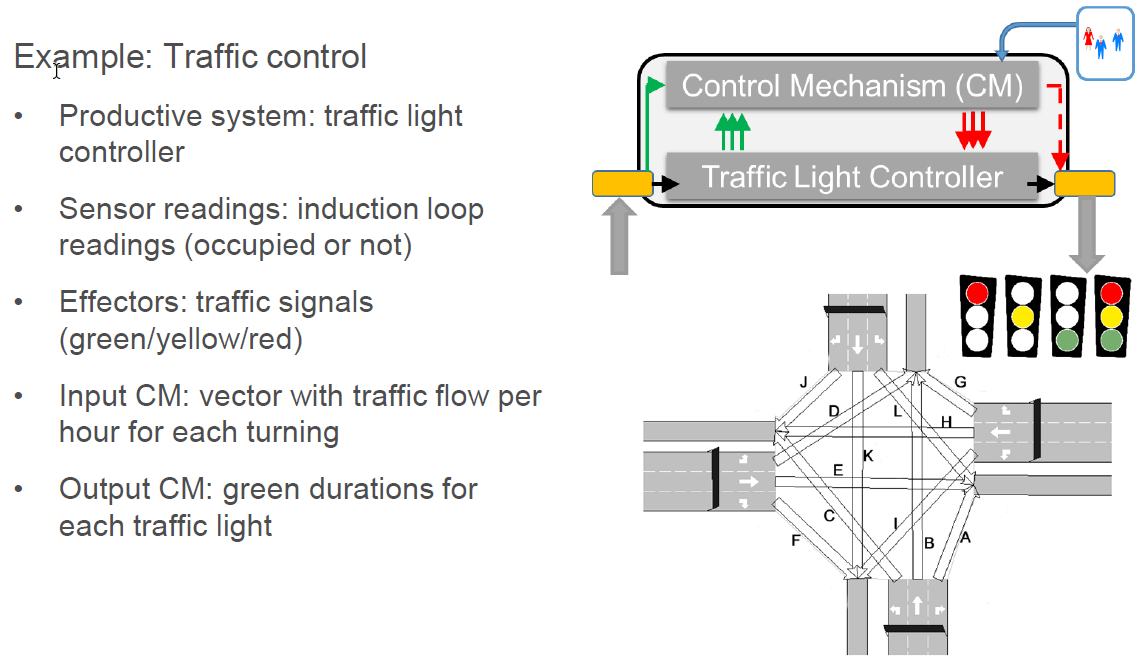
* Prodsystem nur von CM beeinflusst
* Theoretisch kann Nutzer direkt mit Outputsignal agieren
* Nutzer stellt nur Ziel zur Verfügung, das gutes oder schlechtes Verhalten bestimmt
* CM bestimmt was wann zu tun ist

**Anforderungen**

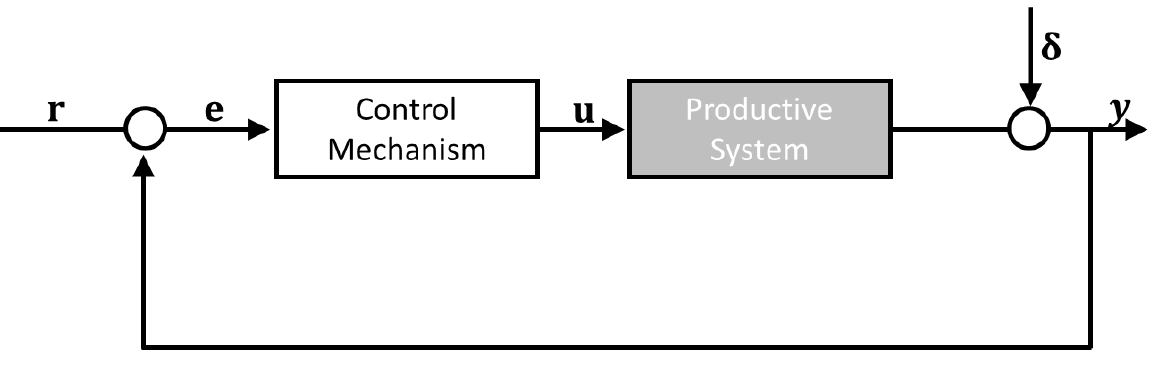
* Zugriff auf Sensorwerte
* Zugriff auf interne Variablen
* Utilityfunktion für gut/schlecht
* Zugriff auf CI von Prodsystem
* CI muss direkt mit Performance korrelieren

**Aktionen**

* Parameteränderung um Systemverhalten zu steuern
* Komponenten und Funktionen de/aktivieren
* Algorithmen/Techniken austauschen
* Interaktionspartner wählen
* Observermodel ändern (sensorinfo, Auflösung usw)

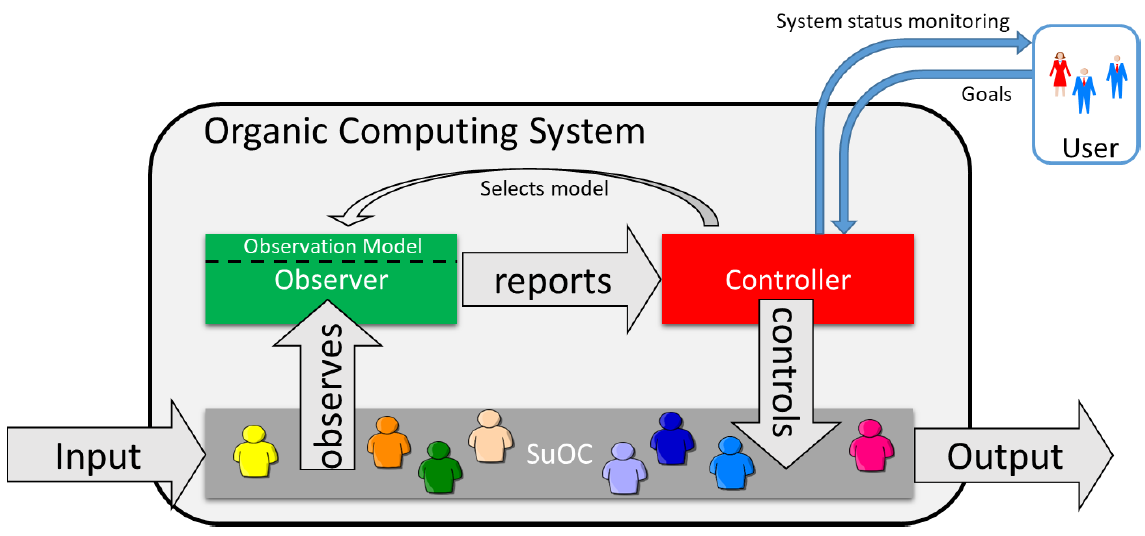
****

**Kontrollschleife**

****

* Aktionen beeinflussen Performance, P wird von Environment gemessen, E wird von Aktionen gesteuert

## Oberserver/Controller Architektur



### Drei Hauptkomponenten

**System Under Observation And Control**

* Produktiver Systemteil
* Funktioniert noch, wenn highlayer O C failt
* Nicht auf einzelne Devices beschränkt

**Observer**

* Sammelt Info, Analysiert, erkennt Muster, schaut voraus
* Beschreibt Situation, basiert auf Observation Model

**Controller**

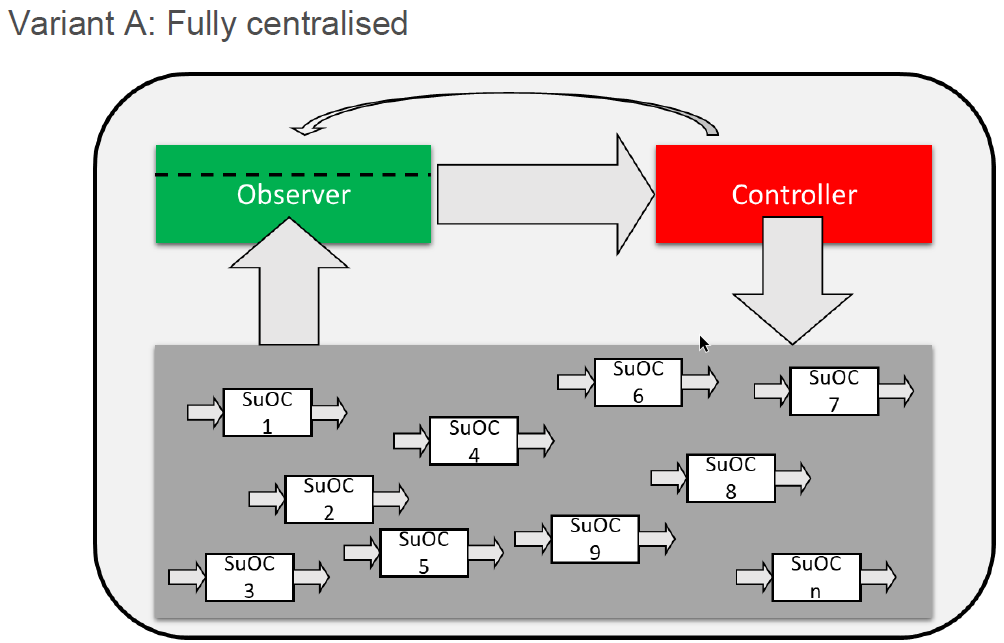
* Entscheidet Aktionen, lernt von Feedback, ändert Observation Model

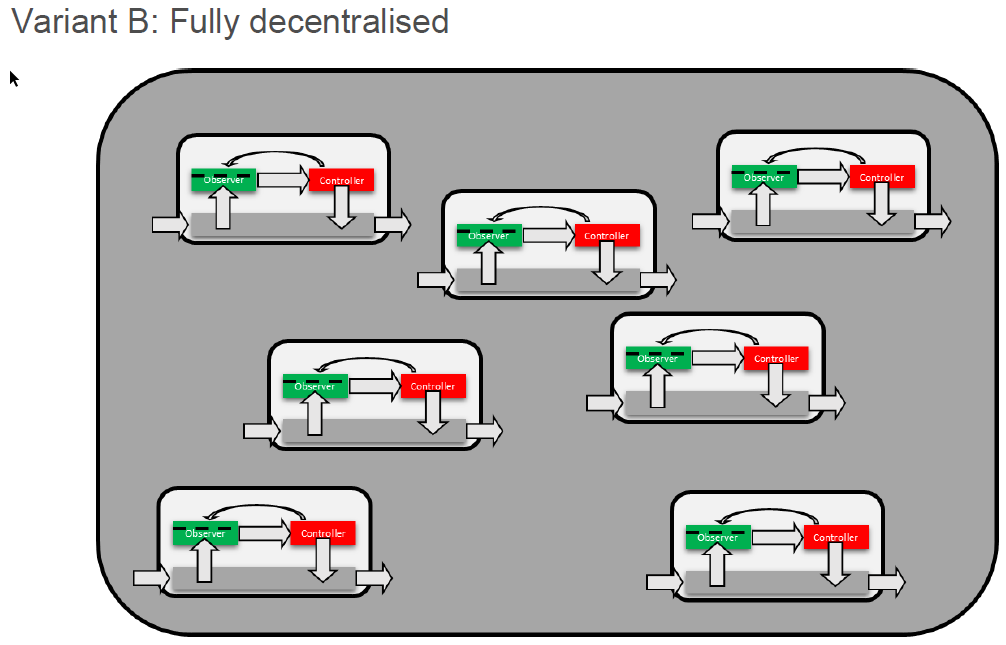
Von Feedback lernen:

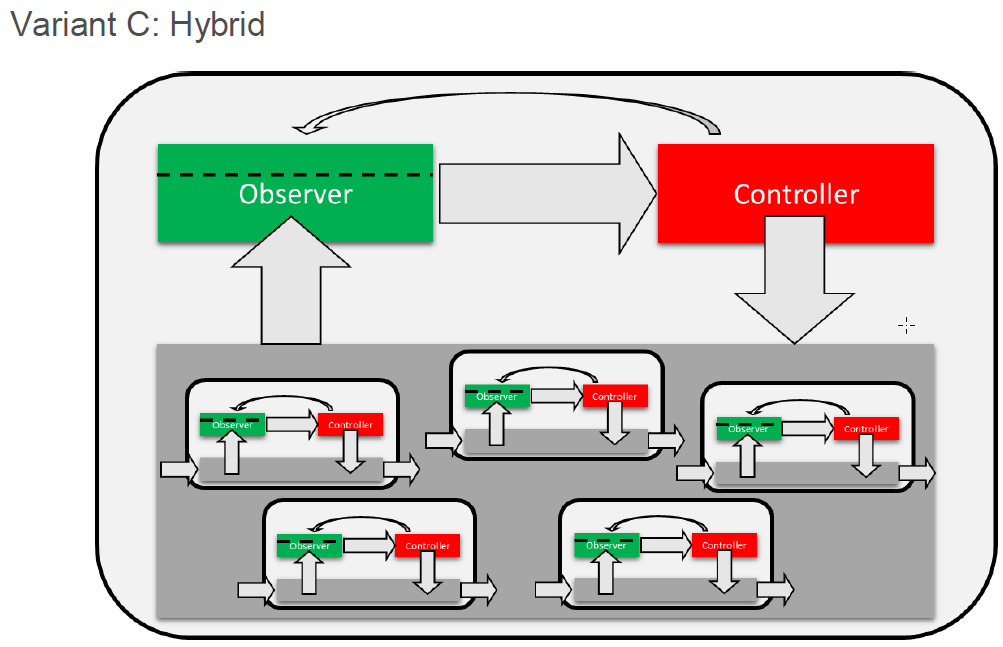
* Regelbasiertes System (Condition | Action | Evaluation )
* Regeln sind kompetitiv, Evaluationskriterium entscheidet welche Regel genommen wird

Ampelbeispiel:

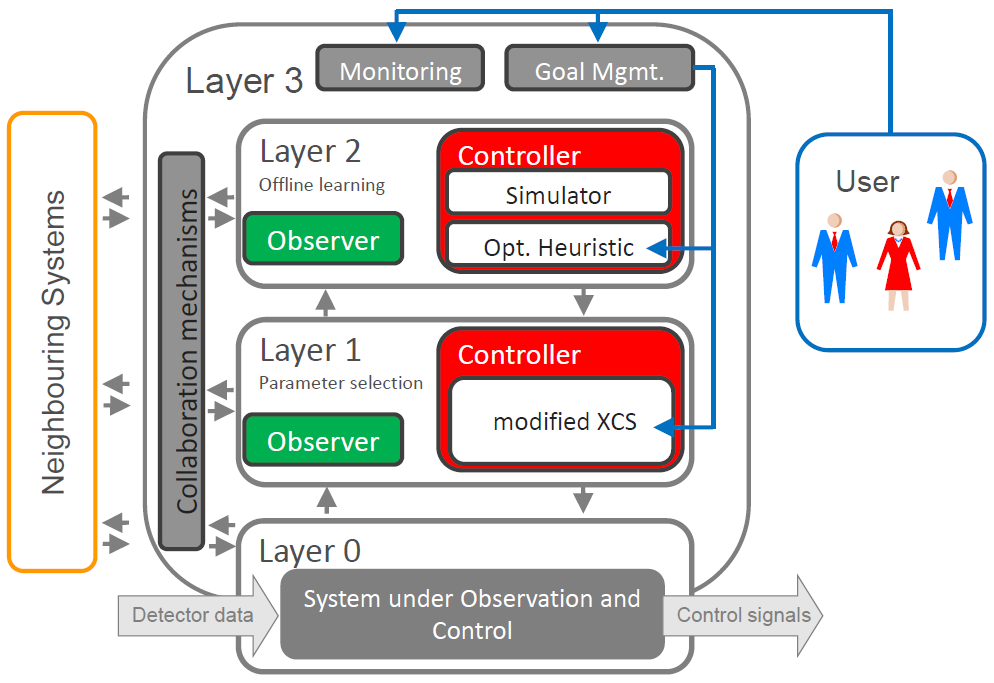
* Condition: Vektor mit Trafficflowintervallen
* Action: Grünphasen pro Ampel
* Evaluation: Durchschnittliche Verzögerungen an Kreuzungen







## Multi-level Observer/Controller framework

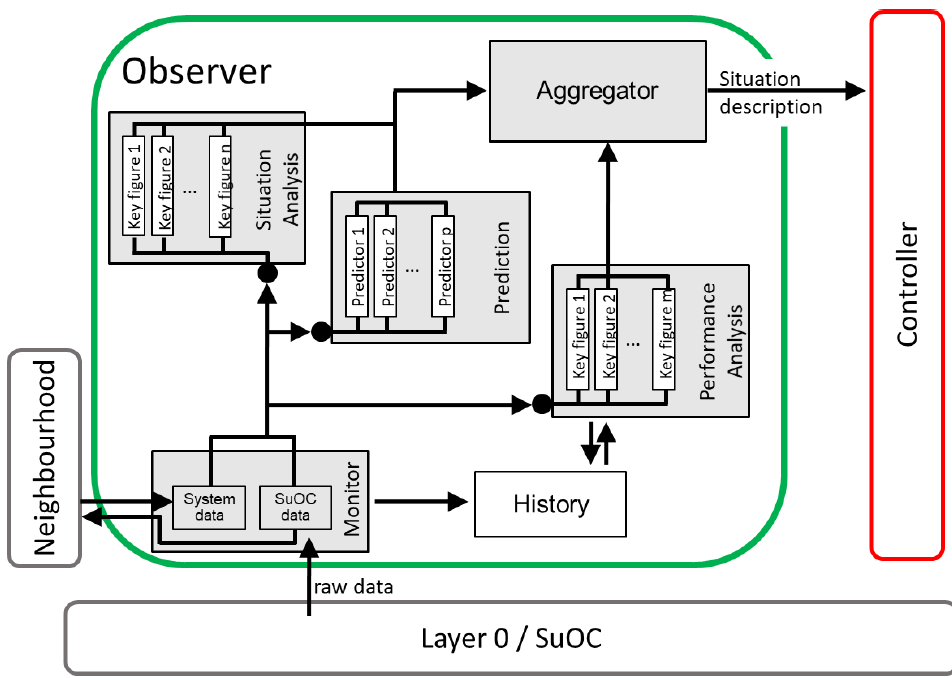


### Layer 0: Productive System

* Enthält SUOC, stellt Zugriff auf Beobachtung und Kontrollinterfaces zur Verfügung

### Layer 1: Parameter Auswahl und Online Learning

* Überwachung von Layer 0: Monitoring, Datenanalyse, Voraussicht, Situationsbeschreibung

Monitor: Sensor und Statusdaten sammeln

Preproc: Missing Values, Wrong Values abarbeiten usw

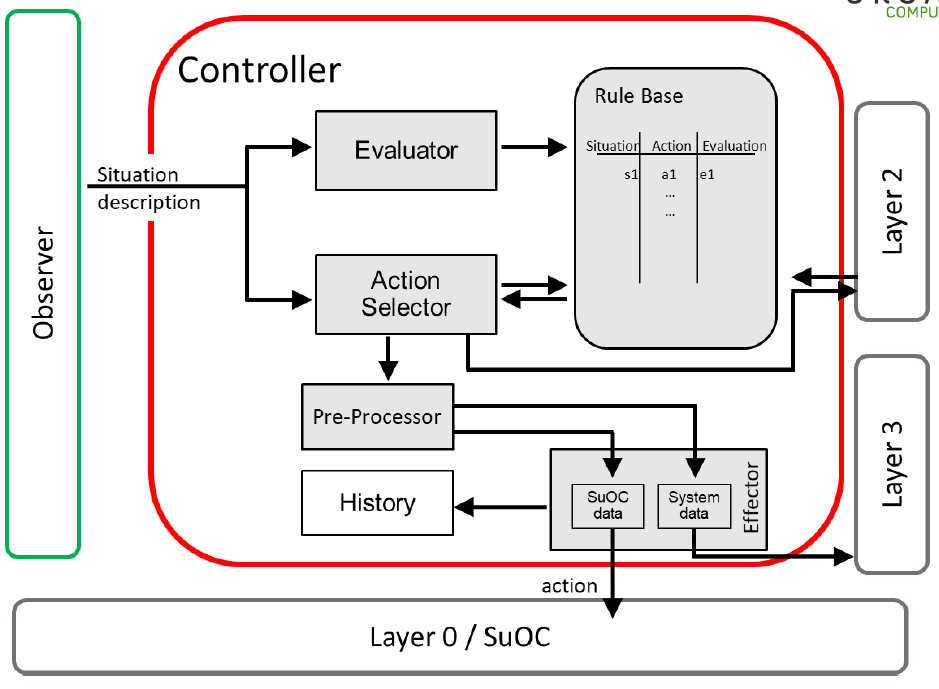
Situation A: Pattern und Emergenz finden

Performance: Werte extrahieren die mit Zielen korrelieren

Aggregator: Situationsbeschreibung

* Kontrolle: Parameter, Struktur und Techniken bei momentanen Bedingungen ändern,

Auswahl aus Regelset und Update von Evaluation

Effektor: Aktivieren Aktionen

Evaluator: Wertet Aktionserfolg aus

### Layer 2: Offline Learning

* Beobachtet Layer 1
* Kontrolle: generiert neue Regel und fügt sie zu Layer 1 hinzu

### Layer 3: Zusammenarbeit

* User: Selbsterklärung, Zielanpassung
* Andere Systeme: Verhandeln, Kommunizieren, Vertrauen

# Optimisierung

Im Layer2 Controller ist Optimiser verankert