MULTIAGENTENSYSTEME: MODELLE, PROGRAMMIERUNG, PLATTFORMEN

PD Stefan Bosse

Universität Koblenz, Fak. Informatik
Universität Bremen, FB Mathematik & Informatik

SS 2018

Version 2018-07-12

sbosse@uni-bremen.de

INHALT

ÜBERBLICK

SCHWERPUNKTE IN DIESEM KURS

- Grundlagen von autonomen Agenten und selbstorganisierenden Systemen
- Konzepte der Programmierung von Agenten: Eher abstrakt oder besser praktisch?
- Praktische Relevanz und Anwendung von agentenbasierten Systemen
- Plattformen und Technologien

Begleitet von Übungen um obige Techniken konkret anzuwenden

Vorlesung
2 SWS mit Grundlagen und Live Programming

Übung
2 SWS mit Programmierung und angewandter Vertiefung

Voraussetzungen
Grundlegende Programmierfähigkeiten

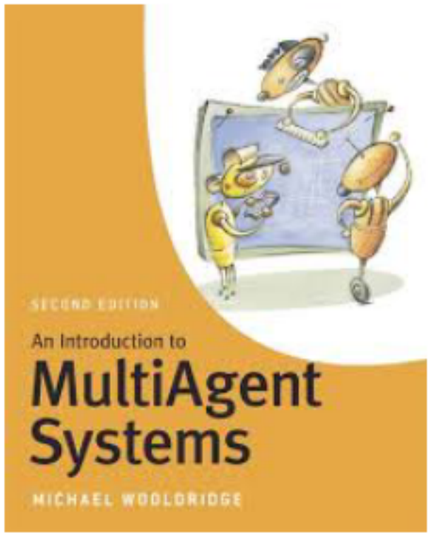
LITERATUR

Vorlesungsskript

Die Inhalte der Vorlesung werden sukzessive bereitgestellt

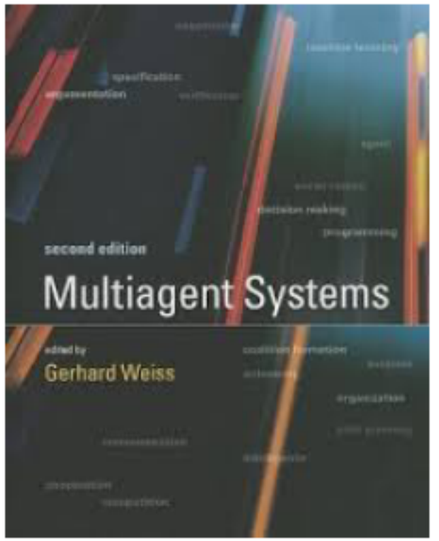
MultiAgent Systems

Michael Wooldridge, John Wiley & Sons, 2002



Multiagent Systems – A Modern Approach to Distributed Artificial Intelligence

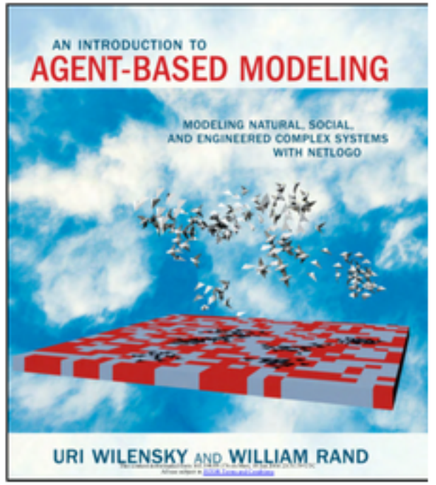
Gerhard Weiss (Ed.), The MIT Press, 2000



LITERATUR

An Introduction to Agent-Based Modeling

Uri Wilensky, William Rand, William, MIT Press, 2015



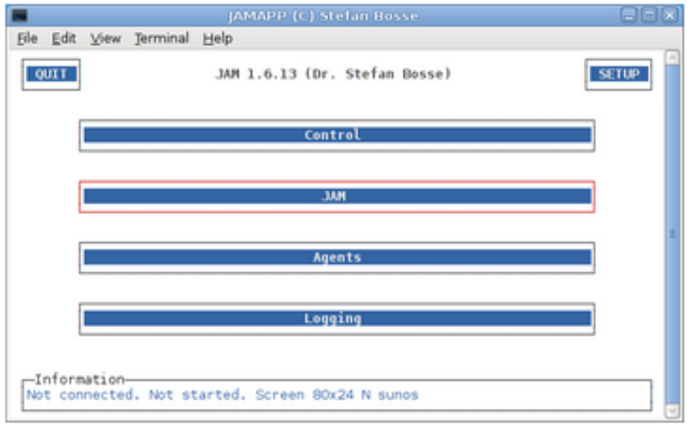
SOFTWARE

Verwendete Software (Vorlesung und Übung):

JAM

sun45.informatik.uni-bremen.de

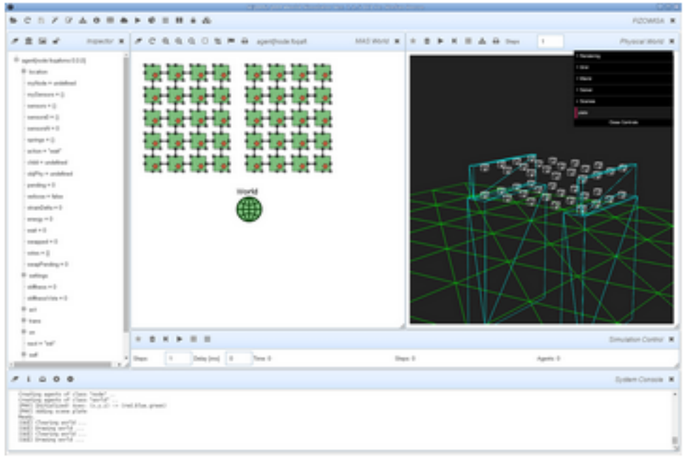
- JAM: JavaScript Agent Machine
- Vollständig in JavaScript programmiert (+Agenten: *AgentJS*)
- Einsatz auf verschiedenen Hostplattformen: PC, Smartphone, Embedded PC, Server, ..



SEJAM2

sun45.informatik.uni-bremen.de

- SEJAM: Simulation Environment for JAM
- Simulationsumgebung und Entwicklungs IDE
- Bottom-up Modellierung



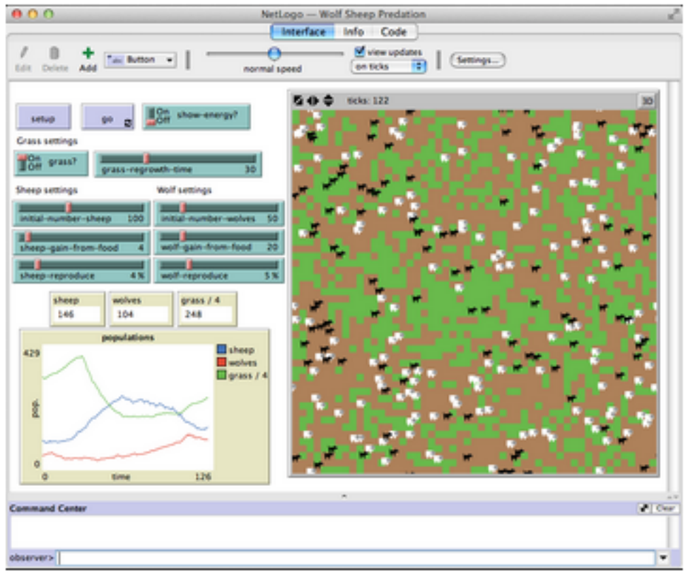
SOFTWARE

Verwendete Software (Vorlesung und Übung):

NetLogo

ccl.northwestern.edu/netlogo
agentscript.org

- Simulation und Evaluierung von Multiagentensystemen
- Vollständig in JAVA programmiert
- Einsatz auf verschiedenen Betriebssystemen: Windows, Unix, MacOS, ..
- Top-down Modellierung
- *AgentScript*: JavaScript Modellierung



ZIELE

Die Studenten erwerben/gewinnen/lernen

1. Grundverständnis von Agenten und deren Verhaltensmodelle
2. Grundlagen verteilter perzeptiver und reaktiver Systeme und Fähigkeit der Programmierung: Wie können komplexe verteilte Systeme mit einfachen Methoden entworfen werden?
3. Grundverständnis und Anwendung der Kommunikation, Kooperation, und Kollaboration zwischen Agenten
4. Fähigkeit der praktischen Anwendung und Abbildung der Agentenmodelle mit Programmierung in einfachen Einsatzszenarien unter Verwendung von JavaScript
5. Verständnis und Anwendung an Beispielen von selbstorganisierenden Systemen und deren Adaptivität
6. Praktische Umsetzung einfacher MAS mit der JAM Plattform, JavaScript (AgentJS), und dem SEJAM Simulator
7. Einblicke in die technologische Umsetzung von Multiagentensystemen und Agentenplattformen

INHALTE

A. Einführung in Agenten und Agentensysteme
B. Anwendungsbeispiele
C. Agentenmodellierung und Agentenmodelle mit einfachen Architekturen
D. Entwurf von Agenten mit Programmierung
E. Praktische Agentenbasierte Modellierung mit NetLogo
F. Agentenkommunikation: Koordination und Kooperation
G. Agentenplattformen
H. Mobile Agenten als mobile Prozesse
I. Praktischer Einsatz von Agenten mit der JAM Plattform
J. Simualtion von Agentensystemen (u.A. mit SEJAM)

PROGRAMMIERUNG

Abstrakt (deklarativ)

Jason (BDI)

```
+!leave(home)
  : not raining & not ~raining
  <- !location(window);
    ?curtain_type(Curtains);
    open(Curtains);
    ..
+!leave(home)
  : not raining & not ~raining
  <- .send(mum,askIf,raining);
  ..
@shopping(1)[chance_of_success(0.7),
  usual_payoff(0.9),
  source(ag1), expires(autumn)]
+need(Something)
  : can_afford(Something)
  <- !buy(Something).
```

Praktisch (prozedural)

AgentJS (JAM)

```
function AgentShopper (charge) {
  this.bank=charge;  this.money=0;
  this.act = {
    init: function () {..},
    percept: function () {..},
    buy:  function () {..},
    gohome: function () {..}
  }
  this.on = {
    'error': function (e) {..},
    'PRICE':  function (val) {..},
  }
  this.trans = {
    init: percept,
    percept: function () {return this.money?buy:gohome},
    buy: percept
  };
  this.next=init;
}
```

EINFÜHRUNG IN DIE AGENTENWELT

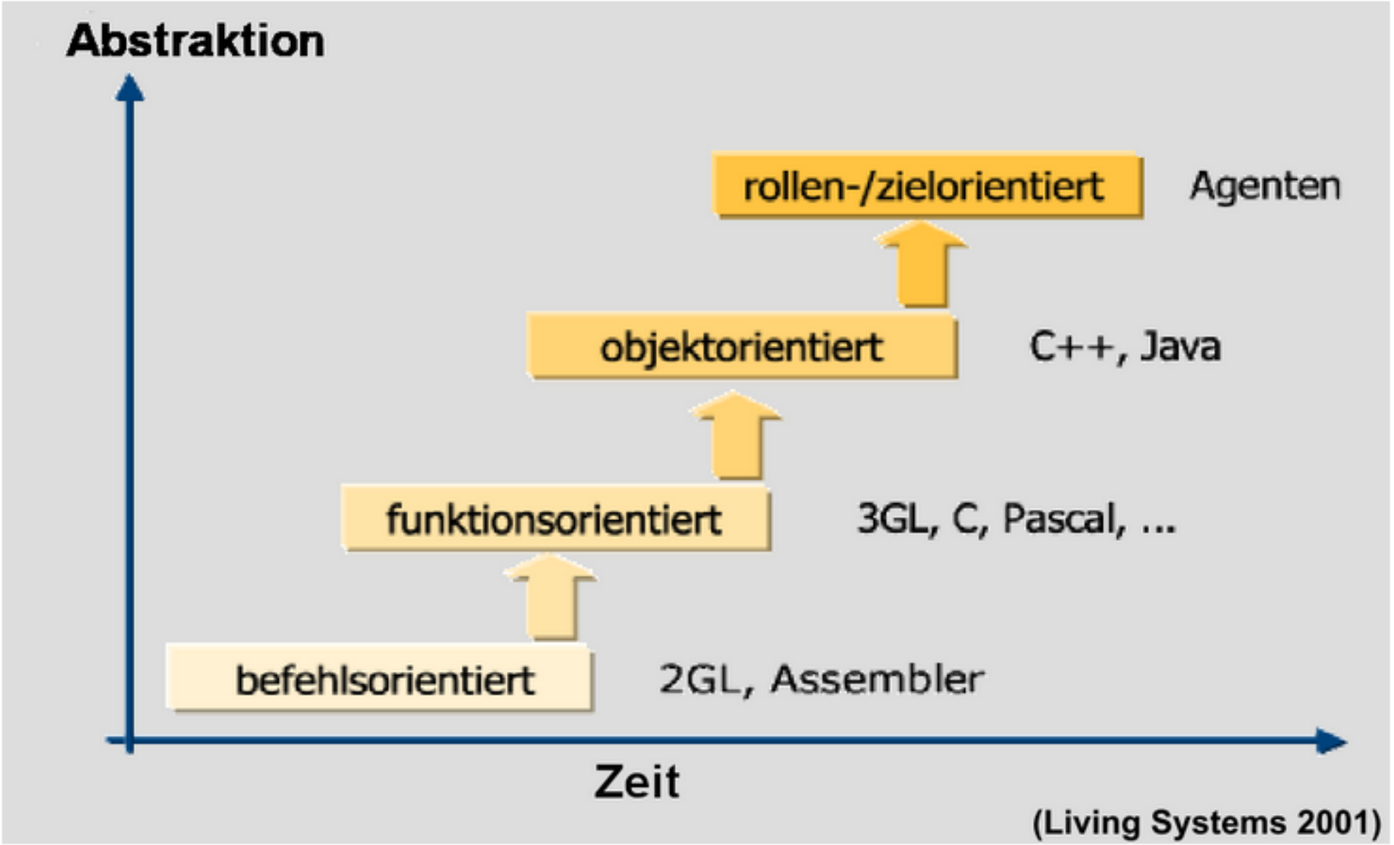
HERAUSFORDERUNGEN

Bei der Entwicklung von modernen Informatiksystemen gibt es verschiedene Herausforderungen:

- **Ubiquität** → 1. Nichtgebundensein an einen Standort 2. Allgegenwart
- **Pervasivität** → Durchdringung der Informatik in Dinge und Geräte
- **Vernetzung** von Geräten und Programmen
- **Verteiltheit** und **Parallelisierung** von Programmen
- **Intelligenz** und **Lernen**
- **Autonomie** → Ohne zentrale Instanzen und Steuerung
- **Robustheit** → 1. Die Welt ändert sich 2. Die Welt verhält sich unsicher
- **Adaptivität** → Die Welt hat sich verändert
- **Delegation** von Aufgaben und Hierarchien
- **Menschen-Maschine** Schnittstelle

SOFTWARE IM WANDEL

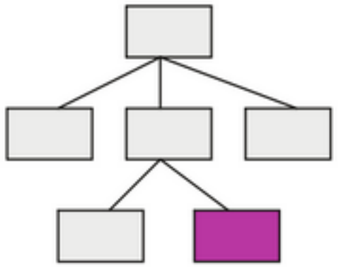
- Die drei Paradigmen der Programmierung: Befehlorientiert → Funktions-orientiert → Objektorientiert
- Zukünftige Softwareentwicklung → Agentenbasiert?



VERGLEICH TRADITIONELLE VS. MULTIAGENTENSYSTEME

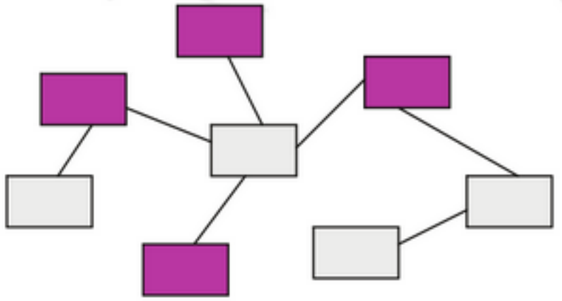
Traditionelle Systeme

- Hierarchien großer Programme
- Sequenzielle Ausführung von Operationen
- Anweisungen von oben nach unten
- Zentrale Entscheidung
- Datengesteuert
- Vorhersagbarkeit
- Stabilität
- Verringerung der Komplexität
- Vollständige Kontrolle



Multiagentensysteme

- Große Netzwerke kleiner Agenten
- Parallele Ausführung von Operationen
- Verhandlungen
- Verteilte Entscheidungen
- Wissensgesteuert
- Selbstorganisation
- Evolution
- Behandlung von Komplexität
- Fähigkeit zum Wachstum



AGENTEN

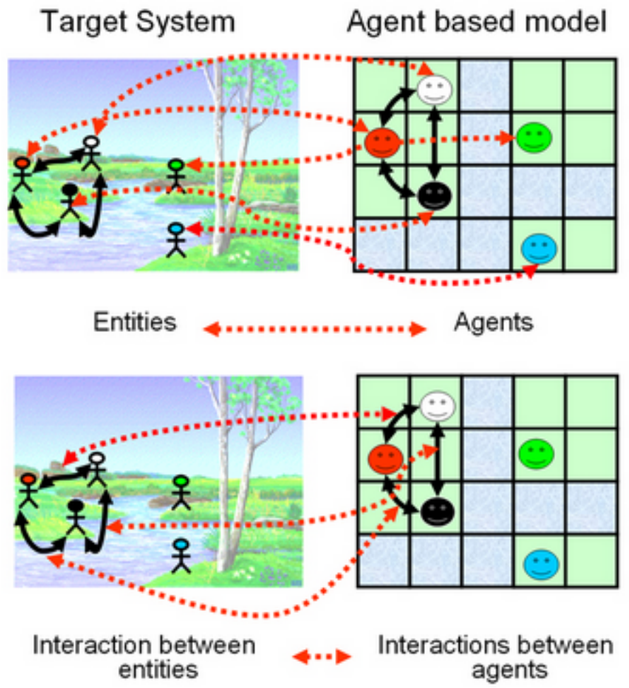
Agenten besitzen eine Vielzahl von Fähigkeiten, die sie von klassischen Programmen unterscheiden - obwohl Agenten auch Programme sein können!

Merkmale

- Fähigkeit zu eigenständiger Aktivität (**Nicht Nutzeraktiviert**)
- Autonomes, "selbstbestimmtes" Verhalten (**Nicht durch zentrale Intsanz gesteuert**)
- Fähigkeit zum selbstständigen Schlussfolgern (**Umgang mit unsicheren Wissen**)
- Flexibles und rationales Verhalten (**Adaptivität an veränderliche Weltbedingungen**)
- Fähigkeit zu Kommunikation und Interaktion (**Synchronisation**)
- Kooperatives oder konkurrierendes Verhalten (**Lösung von Wettbewerbskonflikten**)
- Fähigkeit zur ziel- und aufgabenorientierten Koordination (**Kooperation**)

AGENTEN

- Man unterscheidet zwischen realen und virtuellen Welten;
- Agenten können natürliche (reale) Welten abbilden oder in realen Welten agieren



[Galan,2009]

Abb. 1. Beziehung realer natürlicher Welt mit Lebewesen zu virtueller Welt mit Agenten

MULTI-AGENTEN SYSTEME - ENTWURF

Entwurf von Agenten

- Wie konstruiert man Agenten,
  - die unabhängig und autonom von Nutzeren und Systemadminstratoren agieren,
  - um die an sie delegierten Aufgaben zu erledigen?

Entwurf von Gesellschaften

- Wie konstruiert man Agenten,
  - die mit anderen Agenten interagieren und sich austauschen,
  - um ihre Aufgaben mit dem Ziel einer gloablen Augabe zu erfüllen,
  - auch wenn manche dieser Agenten gegensätzliche Interessen haben und ihre eigenen (konkurrierenden) Ziele verfolgen?

MULTI-AGENTEN SYSTEME - ENTWURF

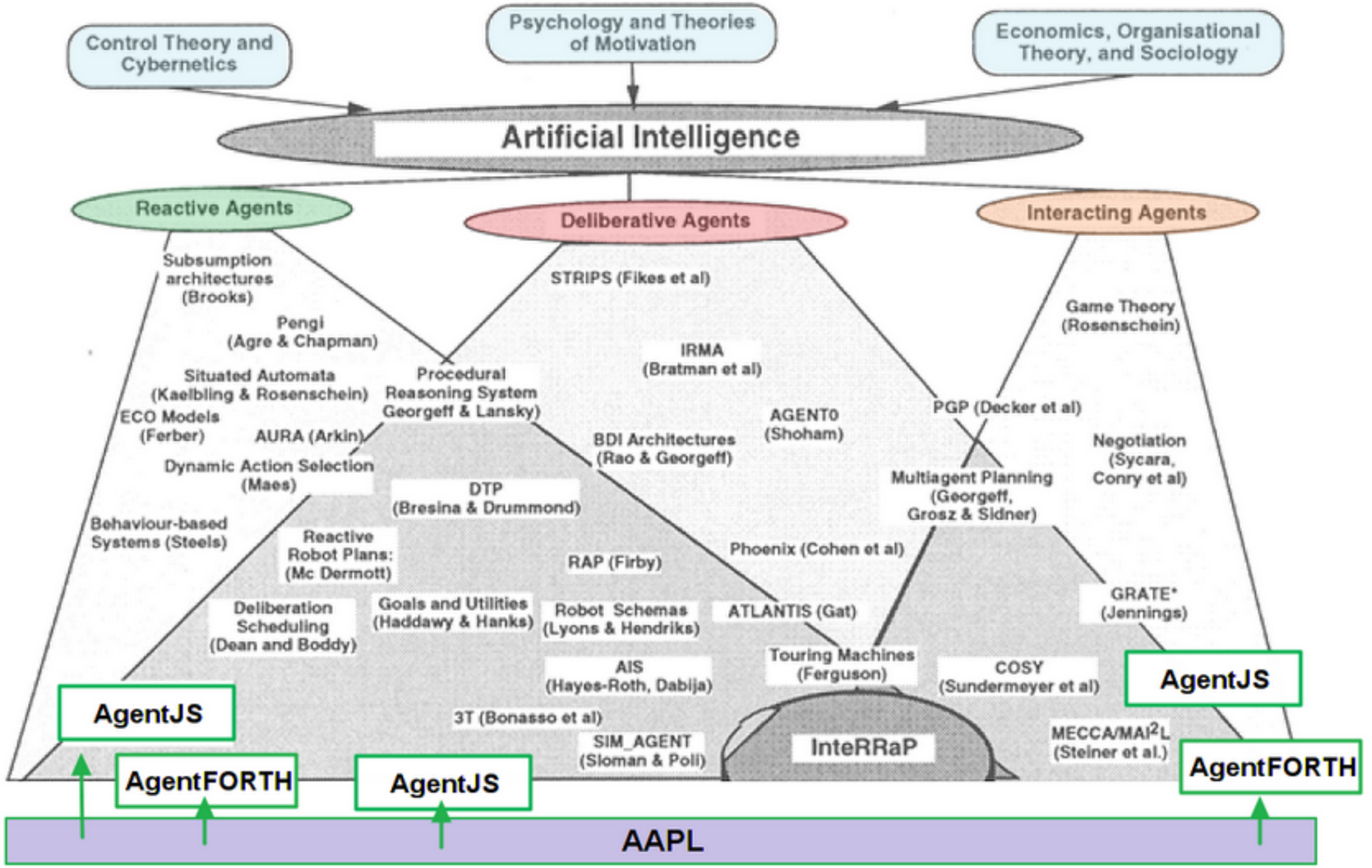


Abb. 2. Entwurf von Intelligenten Agenten: Eine Roadmap [A] und AAPL als Basis

MULTI-AGENTEN SYSTEME - TAXONOMIE

Reaktiver Agent

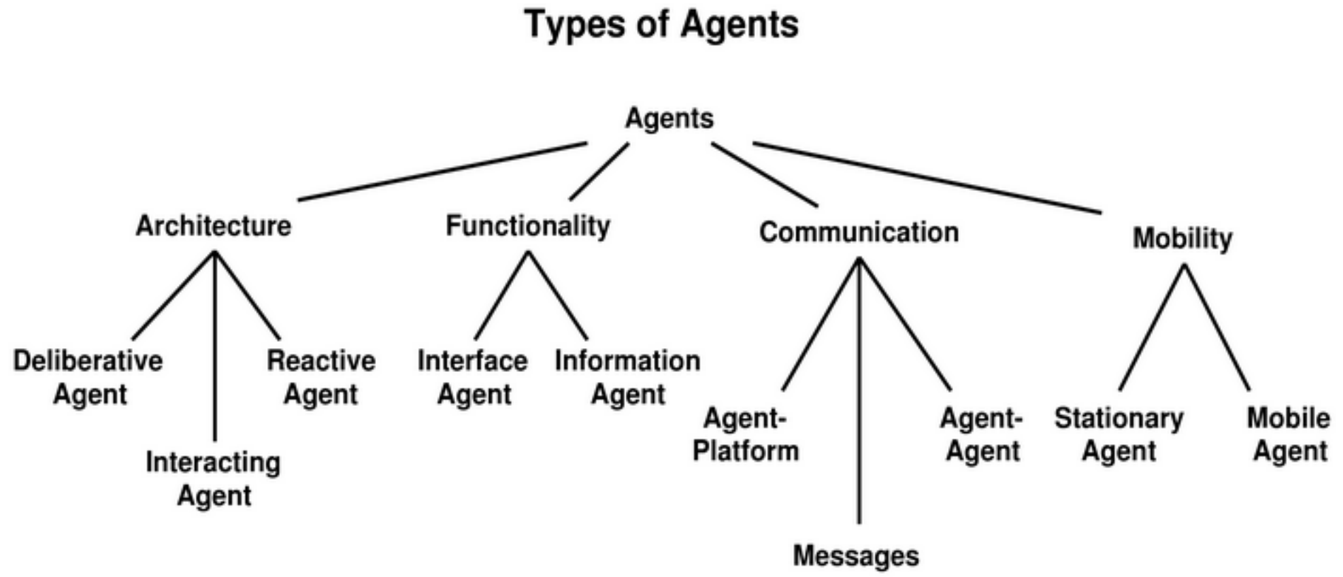
Ein Agent der seine Umgebung wahrnimmt und zeitnah darauf reagiert

Deliberativer Agent

Ein Agent mit explizit dargestelltem, symbolischen Weltmodell mit dem Entscheidungen über symbolische Argumentation getroffen werden

Interagierende Agenten

Multiagentensysteme mit kommunizierenden Agenten



MULTI-AGENTEN SYSTEME - PARADIGMEN UND REPRÄSENTATION

Erweiterung des Paradigmas der intelligenten Systeme

Individuelle Intelligenz

- Ist die Umsetzung von Fähigkeiten wie z.B. Planen, Lernen, Schlussfolgern

Sozialfähigkeit

- Welche gemeinsame **Sprache** können Agenten nutzen, um ihren Wissensstand (Daten) und ihre Absichten mitzuteilen?
- **Onthologien**: Wie muss die Welt (digital, real, virtuell) informatorisch repräsentiert werden, damit sie von Agenten "verstanden" wird?
- Wie erkennen Agenten, dass ihr **Wissen**, ihre **Ziele** oder ihre **Aktionen** mit denen anderer Agenten stimmig ist?
- Wie erzielen und verhandlen sie **Vereinbarungen** oder **Absprachen**?
- Wie koordinieren sie ihre **Aktivitäten**, um gemeinsam (globale) Ziele zu erreichen?
- Wie entsteht **Kooperation** für globale Ziele in **Gesellschaften** "eigennütziger" Agenten?

MULTI-AGENTEN SYSTEME - EMERGENZ

> ***Emergenz*** *ist eine wichtige Eigenschaft in Ensembles von Systemen wo individuelle lokale Aktionen zu einem globalen zielgerichteten Verhalten führen sollen (Schwarmverhalten)*

- Man spricht von Emergenz wenn es ein **Attribut** (Eigenschaft/Ziel) auf **Systemebene** gibt was *nicht* auf individueller Ebene definiert wurde/existiert!
- Komplexe kommunizierende Systeme sind meistens durch emergente Phänomene gekennzeichnet.
- Aber: Ist das emmergente Verhalten gewünscht und mit den Systemzielen (Aufgaben) vereinbar???

MULTI-AGENTEN SYSTEME - INTELLIGENZ

Verteilte Künstliche Intelligenz

Verteilte Künstliche Intelligenz befasst sich mit der Untersuchung, Konstruktion und Anwendung von Multi-Agenten Systemen, in denen mehrere interagierende, intelligente Agenten verschiedene Ziele verfolgen oder eine Reihe von Aufgaben bearbeiten [Biundo-Stephan 2001]

- Intelligente Agenten sind
  - autonome **Software-** oder
  - **Hardwareeinheiten**;
  - die ggf. **mobil** sind (zwischen verschiedenen Hostplattformen migrieren können) sind,
  - die **flexibel** (adaptiv in ihrem Verhalten) basierend auf gelernten Wissen,
  - **robust** (Fehlererkennung, Umgang mit unsicheren Wissen) sind,
  - und mit anderen Agenten sowie ihren Nutzern **interagieren**.

MULTI-AGENTEN SYSTEME - EIGNUNG

- Multi-Agenten Systeme sind geeignet für Anwendungen
  - in großen/ausgedehnten,
  - **verteilten**,
  - **heterogenen** Umgebungen (bezgl. Plattformen, Betriebssystemen, Programmiersprachen, Netzwerktopologien, Performanz..), z.B., im Internet, in Cloud Umgebungen oder Sensornetzwerken,
  - die ein hohes Mass an **Interaktion** erfordern,
  - die technisch **unzuverlässig** und störanfällig sind,
  - die sich in ihrer Konfiguration **ändern** können (d.h. die Welt und ihre Onthologie ändern sich)
  - die einen **Divide-and-Conquer** Ansatz erlauben, d.h. die Zerlegung eines großen Problems, großer Datenmengen, und großer Algorithmen auf immer kleinere Einheiten.

MULTI-AGENTEN SYSTEME - WOFÜR?

Sichten auf Multi-Agenten Systeme

- Agenten als **Softwareentwicklungsparadigma** (agentenorientierte Programmierung)
  - Moderne Softwarearchitekturen setzen sich aus vielen, dynamisch interagierenden Komponenten zusammen
  - Weiterentwicklung der Konzepte der Modularisierung und Objektorientierung
- Agenten als Mittel zur **Modellierung** und **Simulation** natürlicher/menschlicher Gesellschaften (**Soziologie**)
  - Erforschung gesellschaftlicher Entwicklungen in Vergangenheit und Zukunft
  - Verhalten von Menschenmengen (Notfälle, Flucht, ..)

MULTI-AGENTEN SYSTEME - WOFÜR?

- Agenten als Mittel zur **Modellierung** von Netzwerken und **verteilten Systemen**
- Agenten als Mittel zur **Modellierung** von **parallelen Systemen** und Konkurrenz
- **Mobile Agenten** als **Verteilungsparadigma** und verteiltes Datenverarbeitungsmodell (Sensornetzwerke, Internet der Dinge, Cloud Computing, …)

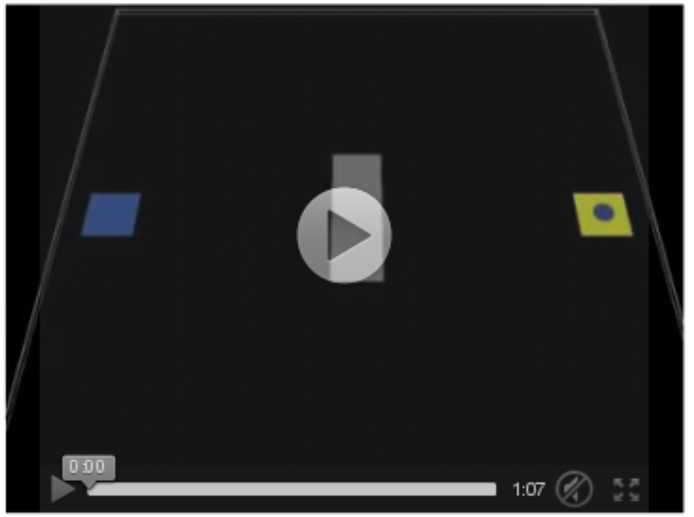
MULTI-AGENTEN SYSTEME - DAFÜR!

Simulation komplexer Welten

- Wolf und Schaf Population und deren Wechselwirkung
- Verwendung von NetLogo zur Simulation und Analyse

Modellierung komplexer Welten

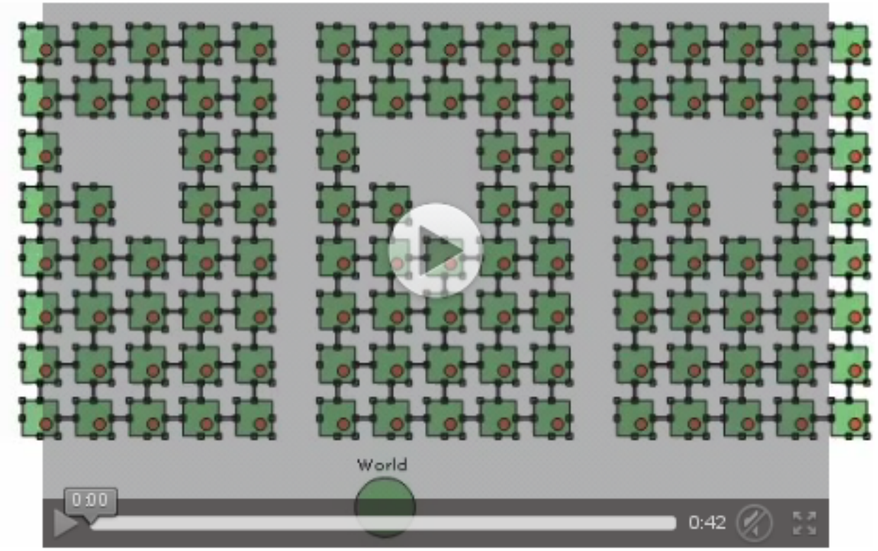
- Versorgungswege von Ameisen (Nahrungsbeschaffung)
- Übertragung auf und Einsatz in Verkehrslenkung und Planung



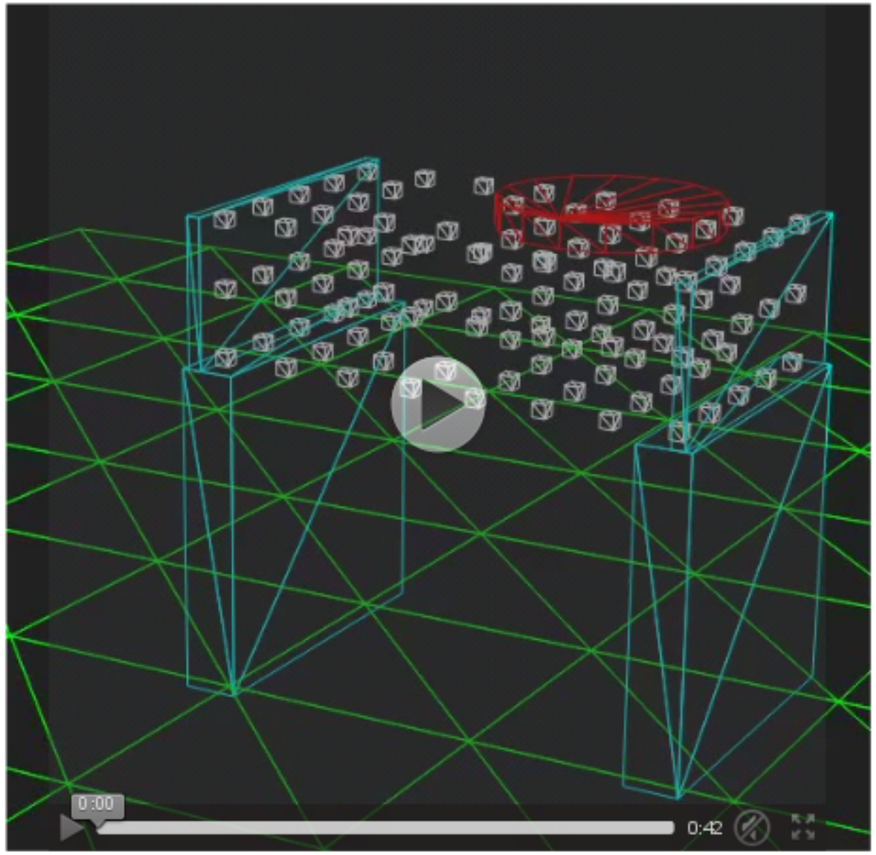
MULTI-AGENTEN SYSTEME - DAFÜR!

MAS Welt

- Ereignisbasierte Sensorverarbeitung in einem Sensornetzwerk
- Kopplung der Agenten mit Physikalischen System (Adaptives Material)



Physikalische Welt



MULTI-AGENTEN SYSTEME - KLASSIFIZIERUNG

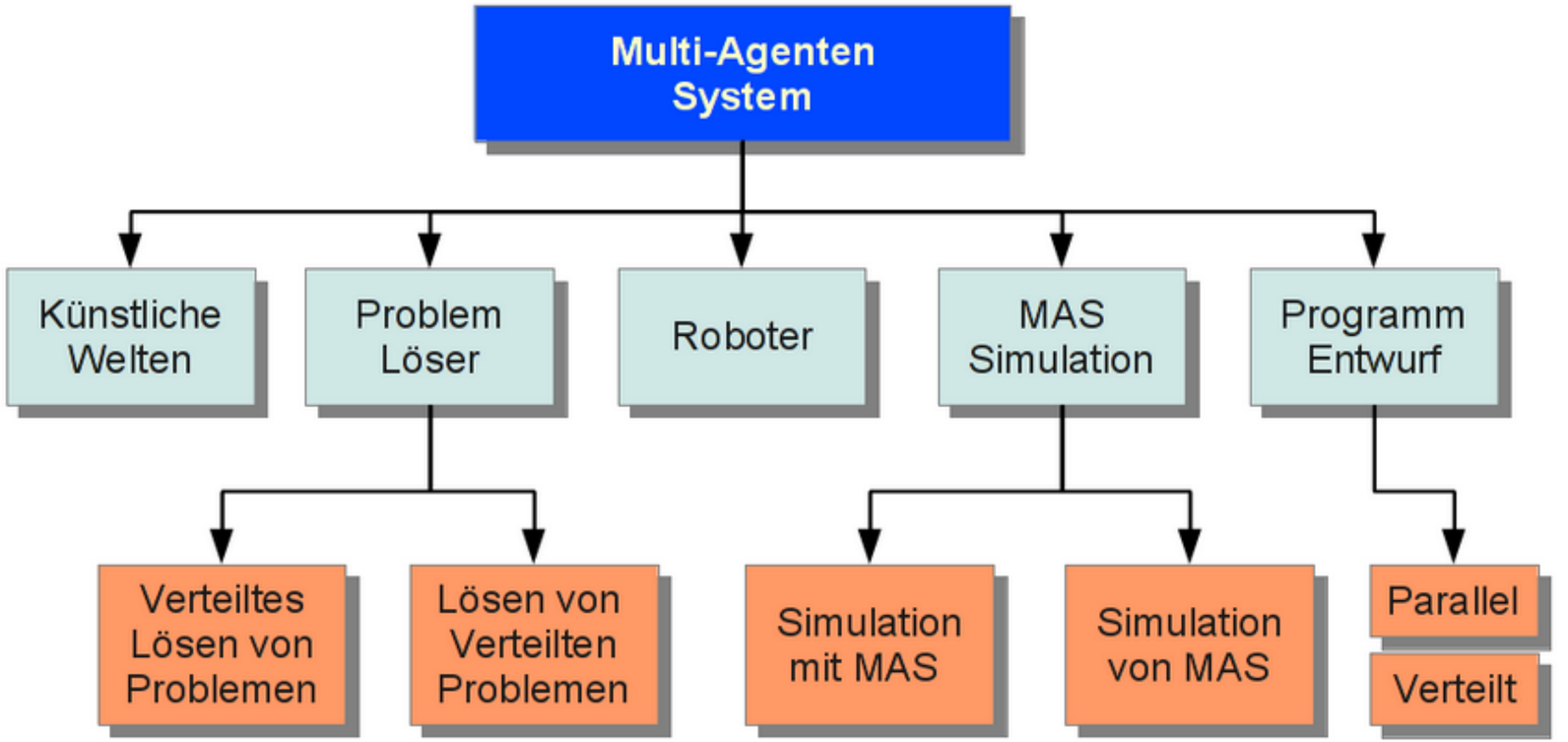


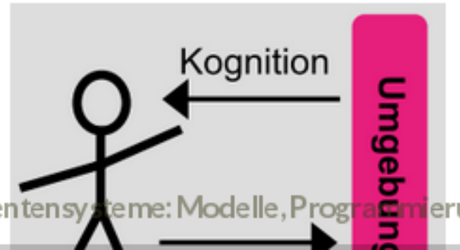
Abb. 3. Klassifizierung und verschiedene Typen der Anwendung von MAS

MULTI-AGENTEN SYSTEME - ZUSAMMENFASSUNG

Agent

Definitionen

- Jeder im Auftrag oder Interesse eines anderen Tätige [Meyer 1994]
- Ein Software-Agent ist ein Programm, das seine Umgebung wahrnimmt und in dieser Umgebung agiert [Schneeberger 2001]



Eigenschaften

Autonomie

- Kontrolle über internen Zustand

Reaktivität

- Wahrnehmung der dynamischen Umgebung
- Reaktion auf die dynamische Umgebung
- Proaktivtät

Initiative

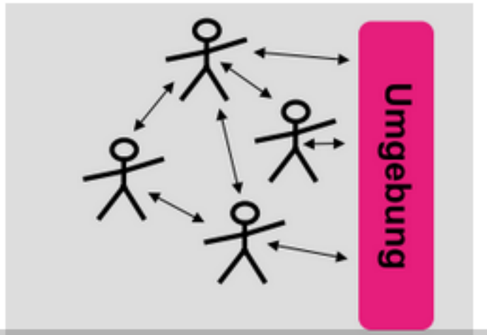
- Zielorientierung
- Planung

Kommunikation

- Mit Menschen

Multiagentensysteme

- Systeme aus einer Vielzahl von gleichen oder verschiedenen Agenten
- Interagierende, intelligente Agenten die verschiedene Ziele verfolgen oder eine Reihe von Aufgaben bearbeiten.

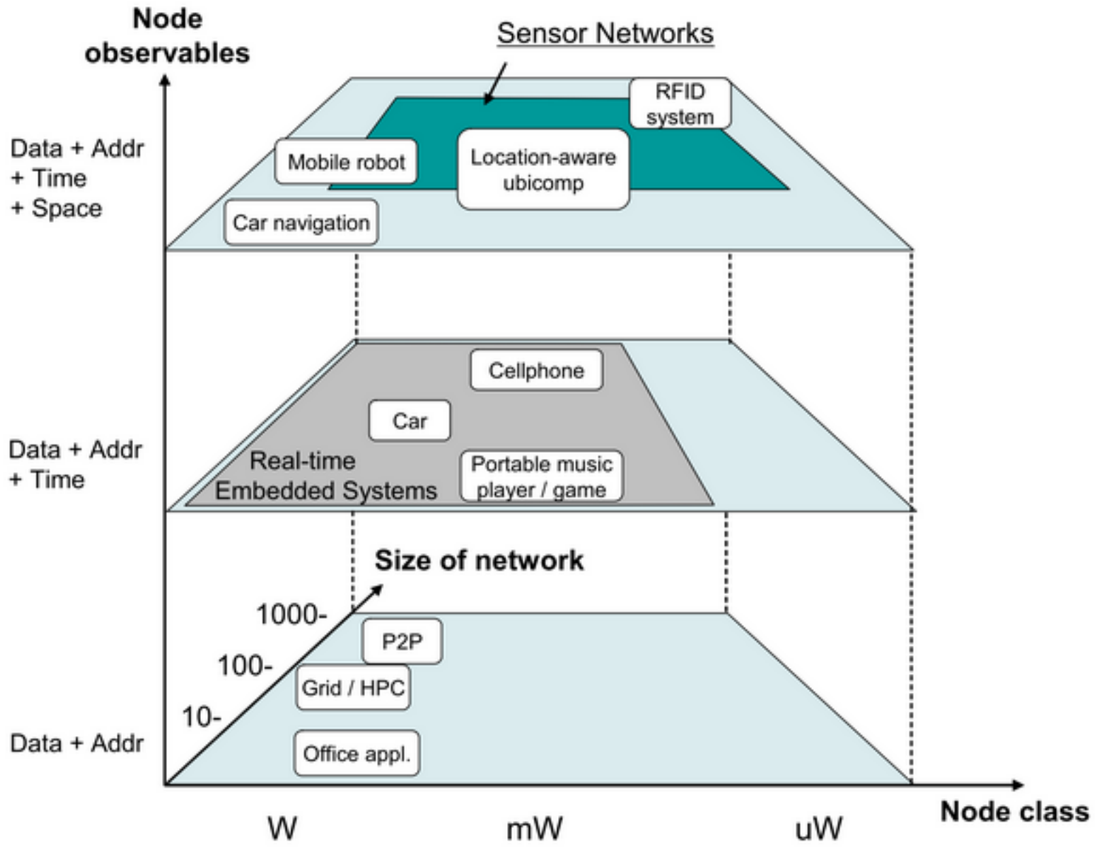


Klassifikation von Netzwerken

Klassifikation nach den **Observablen** und dem elektrischen Leistungsbedarf
→ Datenverarbeitung in Netzwerken und Sensornetzwerken

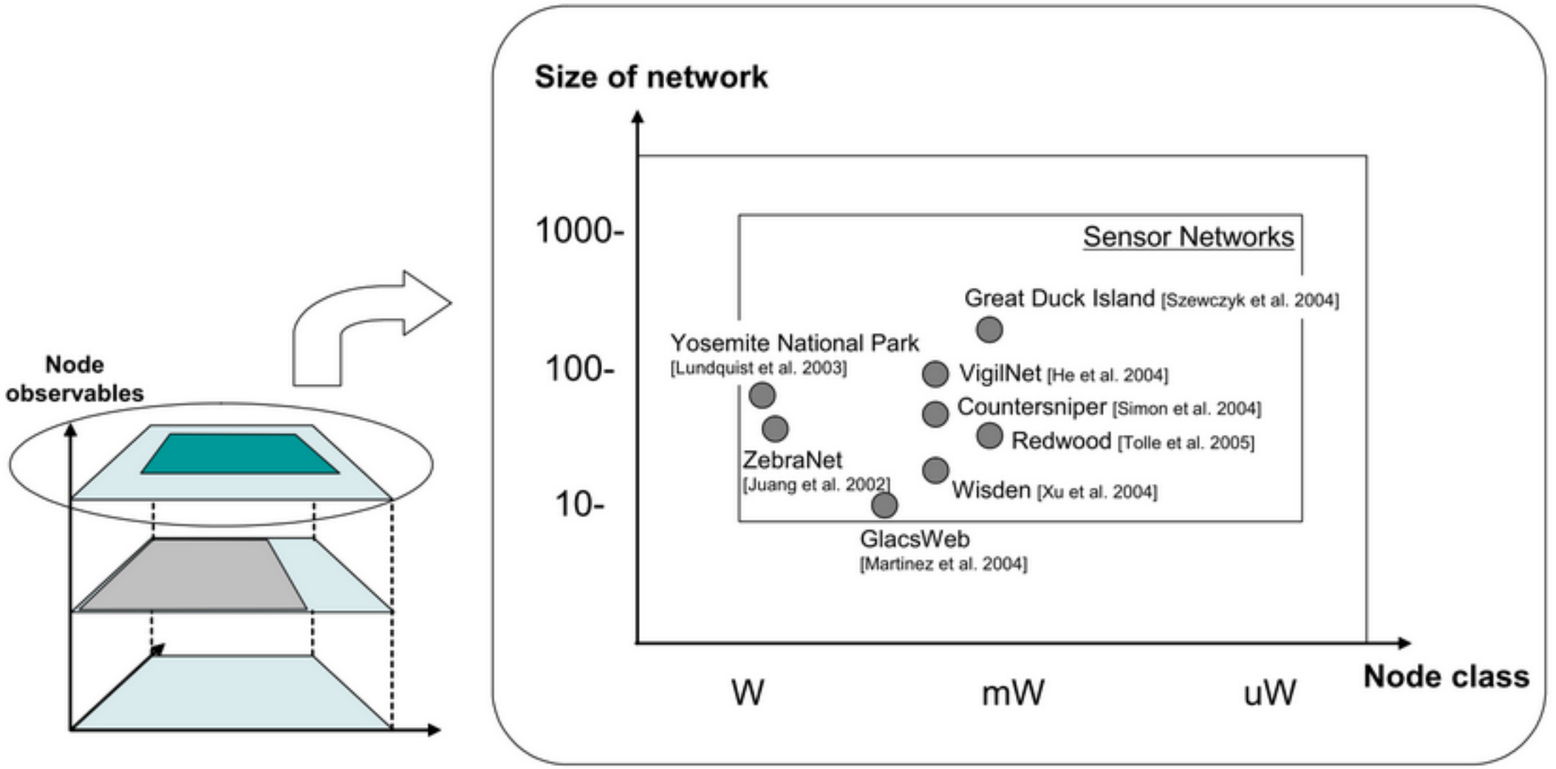
Relevante Observablen von Netzwerkknoten:

- Daten (Sensoren)
- Geräteaddresse; Identifikation
- Zeit (Latenz, Synchronisation)
- Raum (Ort, Ausdehnung)



[Sugihara, 2008]

Sensornetzwerke



[Sugihara, 2008]

Abb. 4. Beispiele von Sensornetzwerken nach Anzahl der Knoten / Knoten Klasse

MULTI-AGENTEN SYSTEME IN SENSORNETZWERKEN

Multi-Agenten Systeme können für

- **autonome**,
- **zuverlässige**,
- **selbstorganisierende**, und
- **adaptive** Datenverarbeitung und Kommunikation in Netzwerken genutzt werden.

> *Multi-Agenten Systeme mit reaktiven mobilen Agenten können für die Sensorverarbeitung in Sensornetzwerken eingesetzt werden, die aus Sensorknoten bestehen die unzuverlässig arbeiten (z.B. wegen Energiemangel) und unzuverl. verbunden sind.*

MULTI-AGENTEN SYSTEME IN SENSORNETZWERKEN

Funktionalen Schichten in Sensornetzwerken

- **Vertikale Schichten**

Sensing
Akquisition and Vorverarbeitung von Sensordaten sowie Sensordatenfusion

Aggregation
Verteilung und Sammlung von Sensordaten, Informationsgewinnung, Fusion

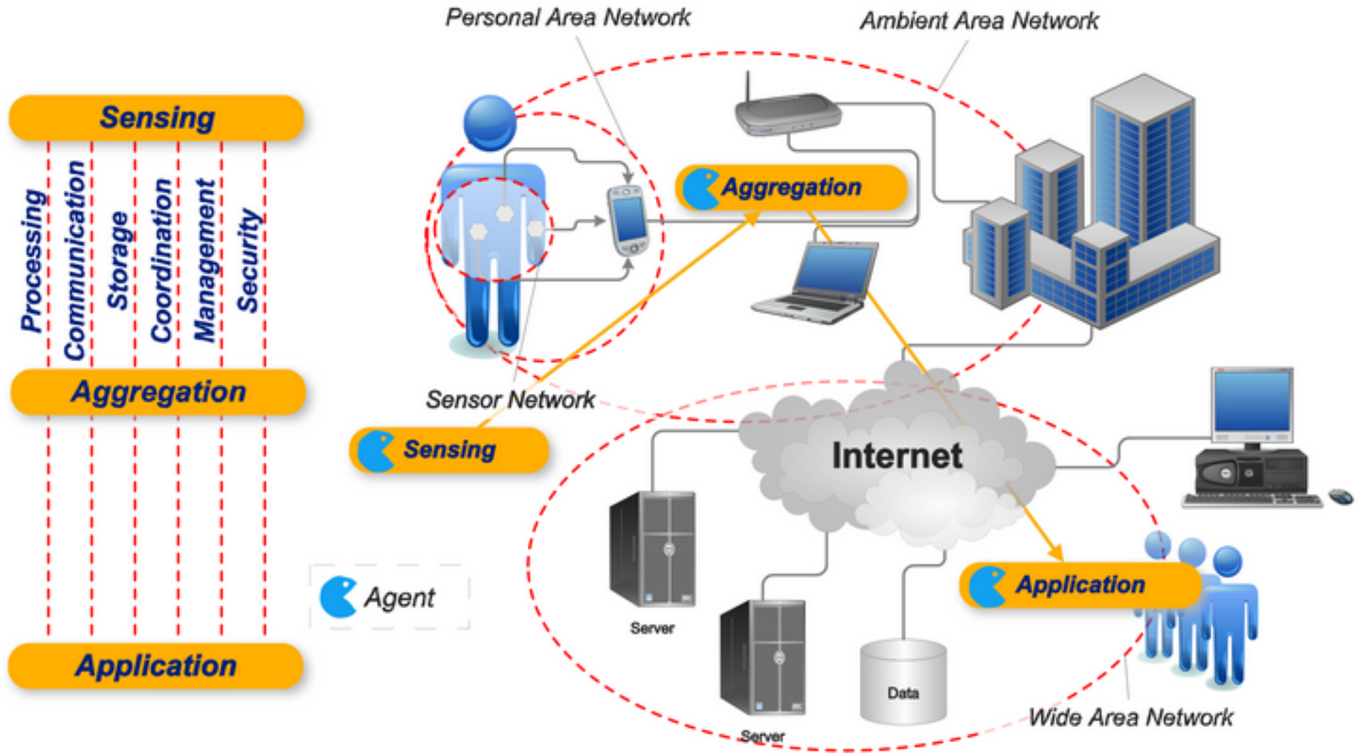
Applikation
Analyse, Speicherung, Visualisierung, Mensch-Interaktion, Datenbanken, Server

- **Horizontale Schichten**

- Datenverarbeitung
- Kommunikation
- Speicher
- Kooridnation
- Management
- Sicherheit

MULTI-AGENTEN SYSTEME IN SENSORNETZWERKEN

Funktionale Schichten in Sensornetzwerken stellen Aufgaben, Ziele, und Kooperation von Agenten dar! Mobile Agenten können alle Schichten abdecken!



ANWENDUNGSBEISPIELE

CAPNET

CAPNET: Component Agent Platform based on .NET

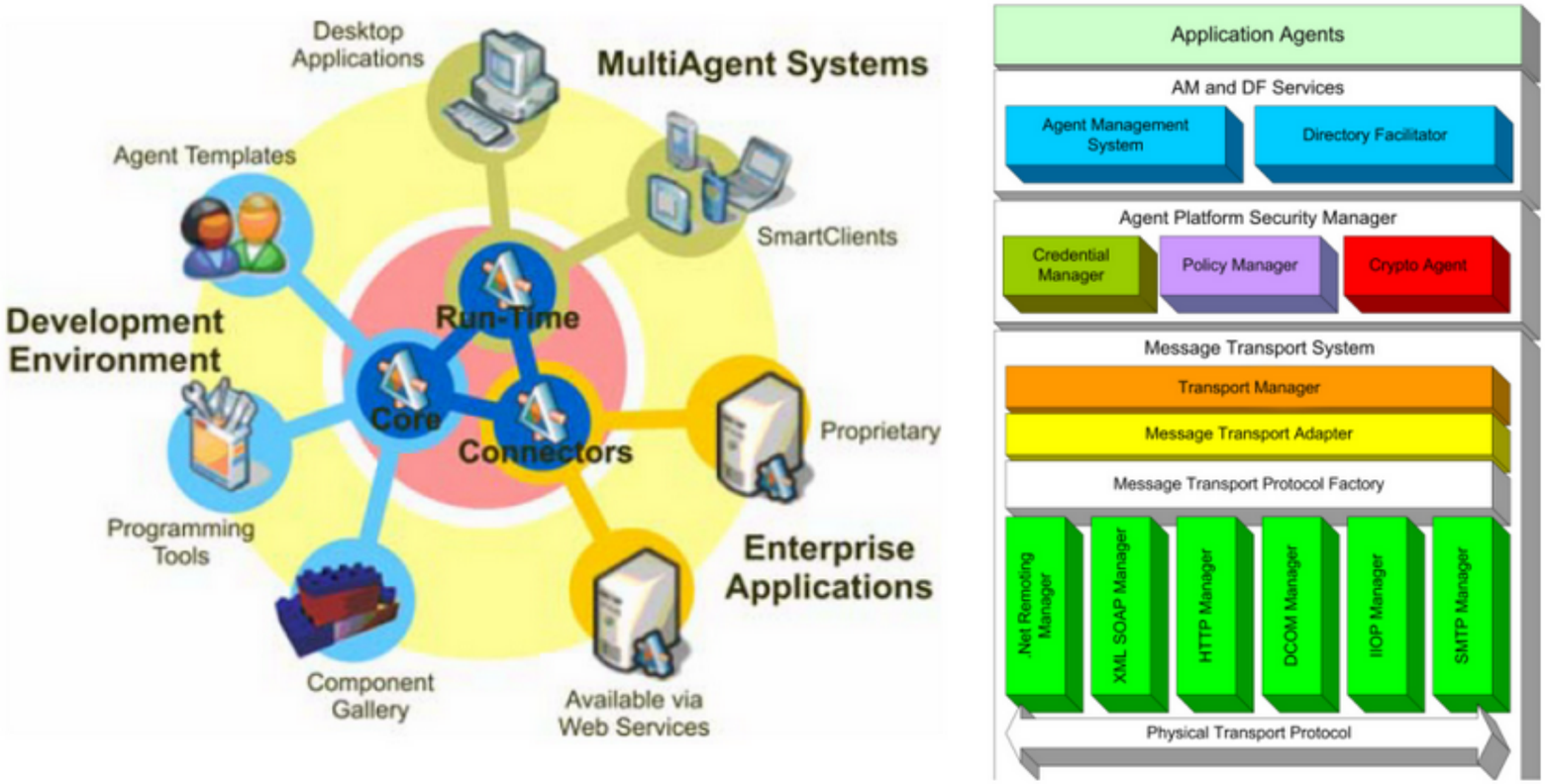


Abb. 5. (Links) Stark heterogene und verteilte Systeme (Rechts) Plattform [Contreras, 2004]

HERA

Hera: Healthcare and Homecare Services System

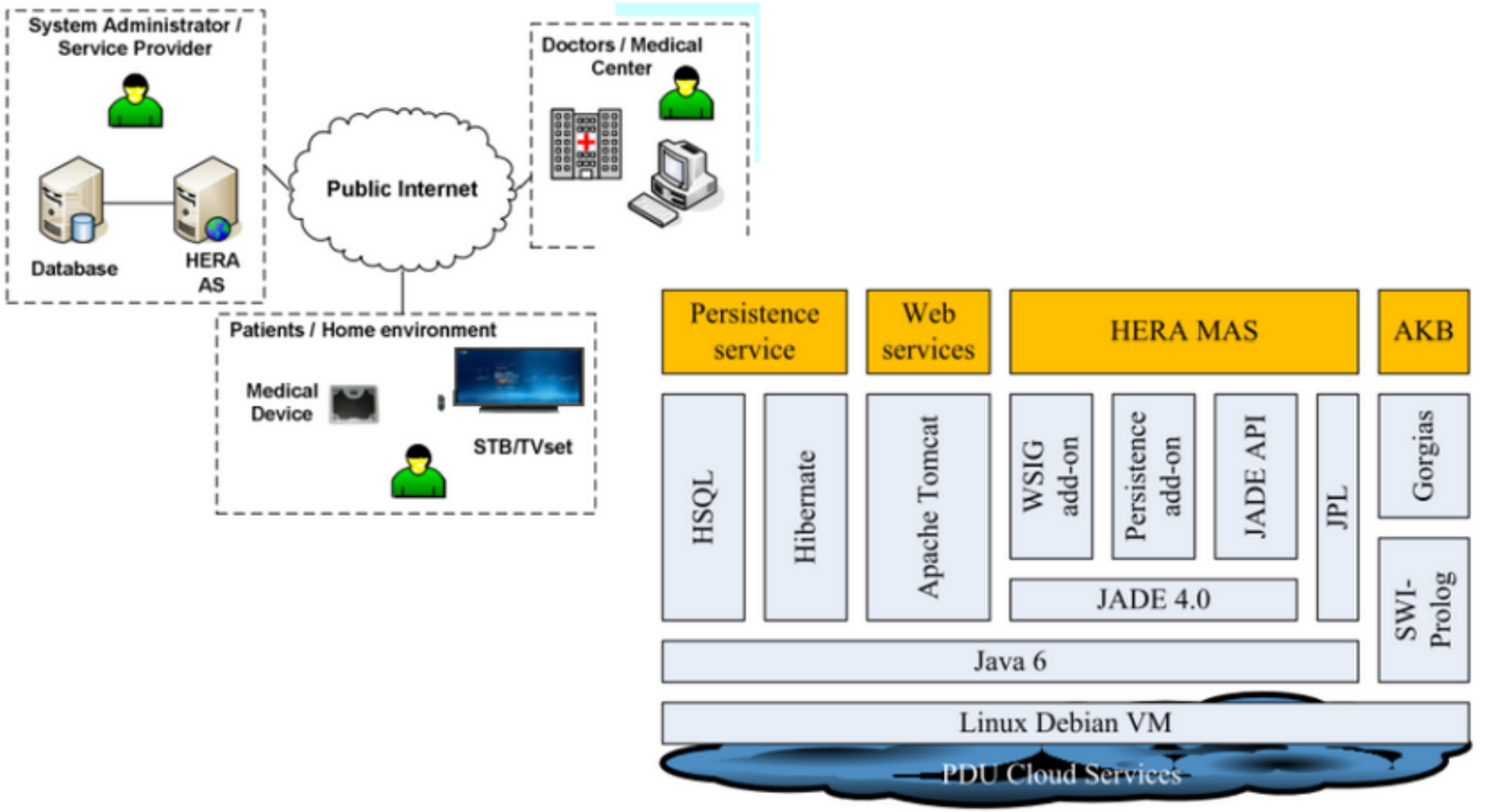


Abb. 6. (Links) HERA Helathcare and Homecare Services System (Rechts) Plattform: JAVA und JADE basierend [Spanoudakis, 2015]

MAGENTA: Multi-Agenten Systeme für Logistik

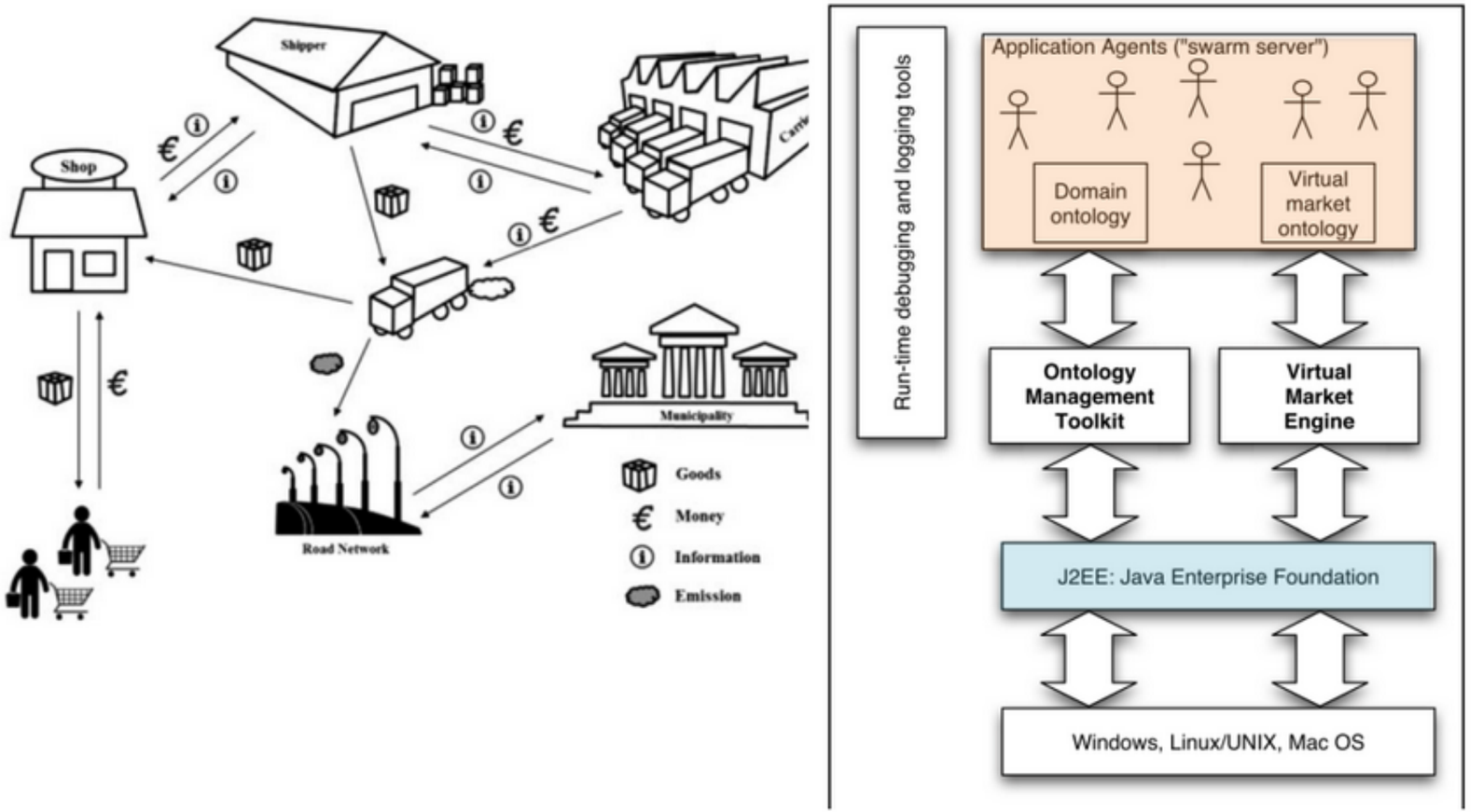
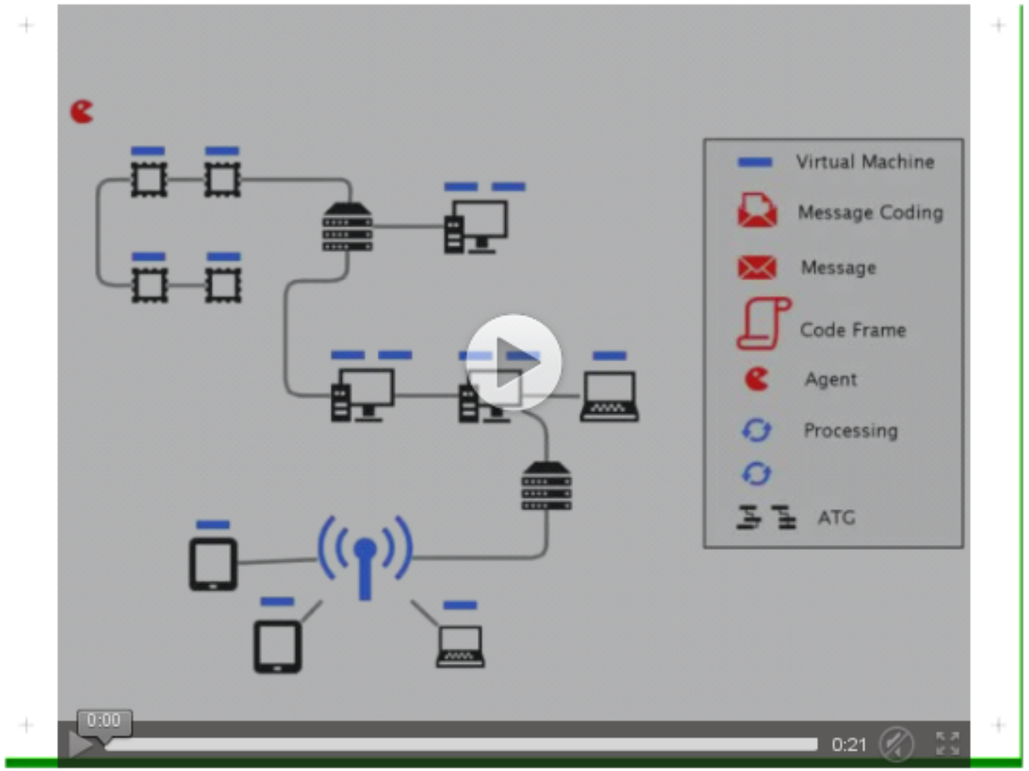


Abb. 7. (Links) Multi-Agenten Systeme für Logistik [Anand, 2014] (Rechts) Plattform [Himoff, 2005]

MOBILE CLOUD

- Traditionelles verteiltes Rechnen verwendet lokalisierte Prozesse und mobile Daten (Nachrichtenübertragung)
- Notwendiger Paradigmenwechsel: Von mobilen Daten zu mobilem Code (Prozesse)
- Agent als Mobiler Code in stark heterogenen Netzwerken



CROWD SENSING

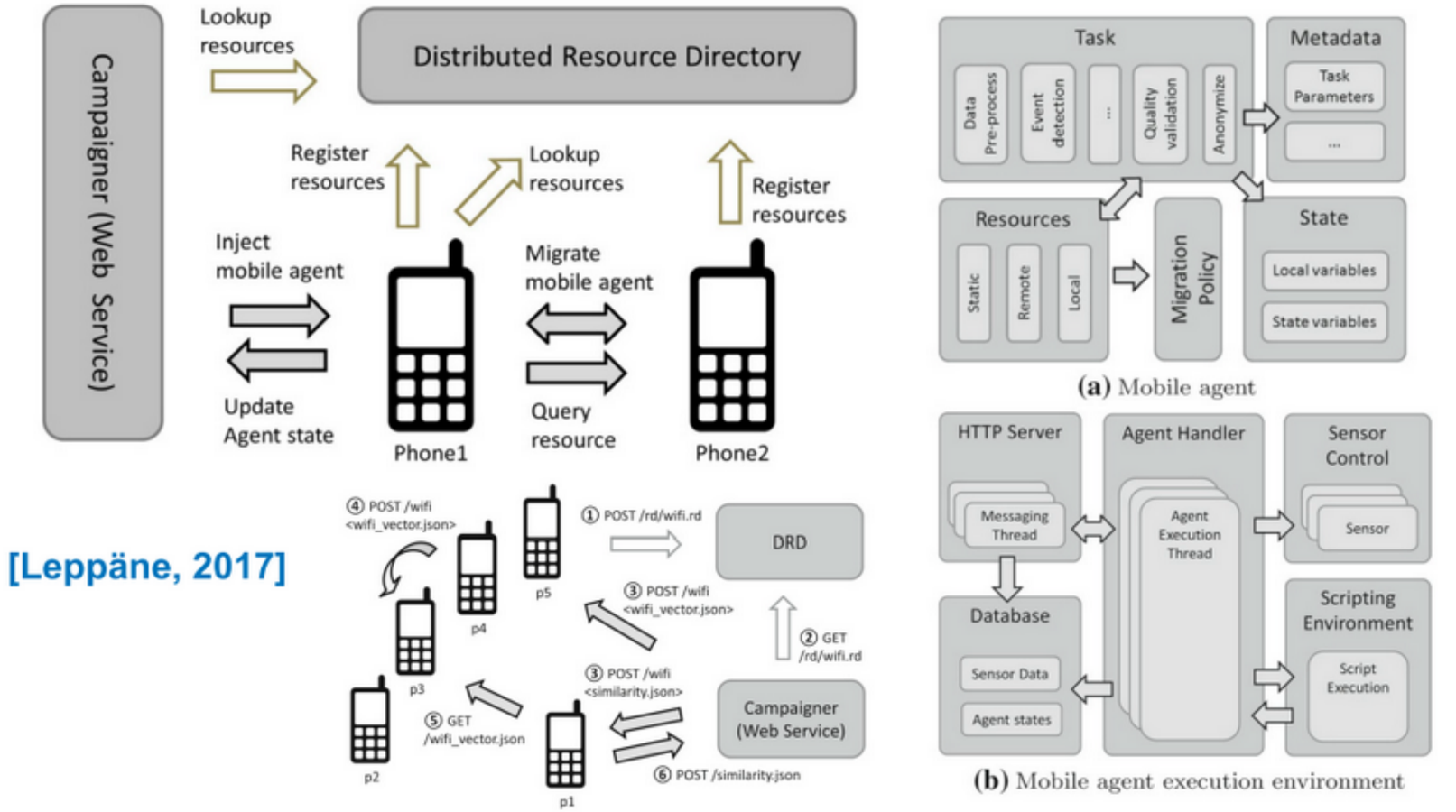
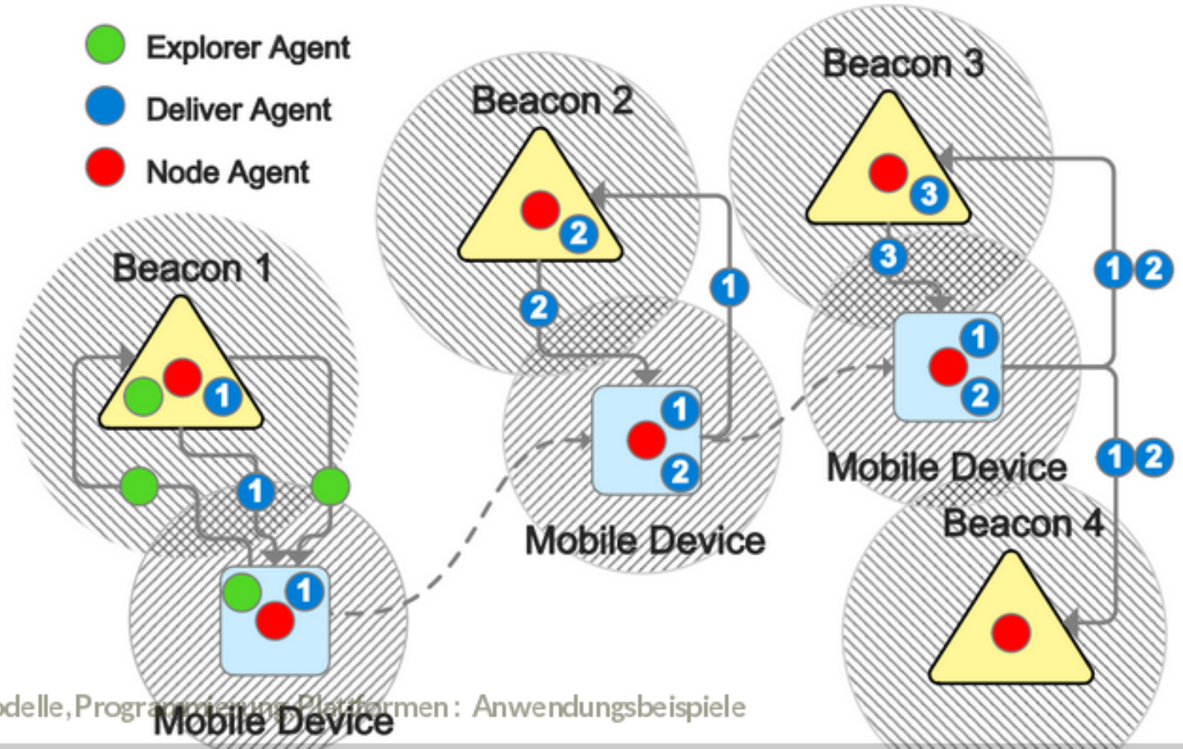


Abb. 8. (Links) Datenaustausch zwischen Smartphones und WEB Service (Rechts) Agent & Plattform

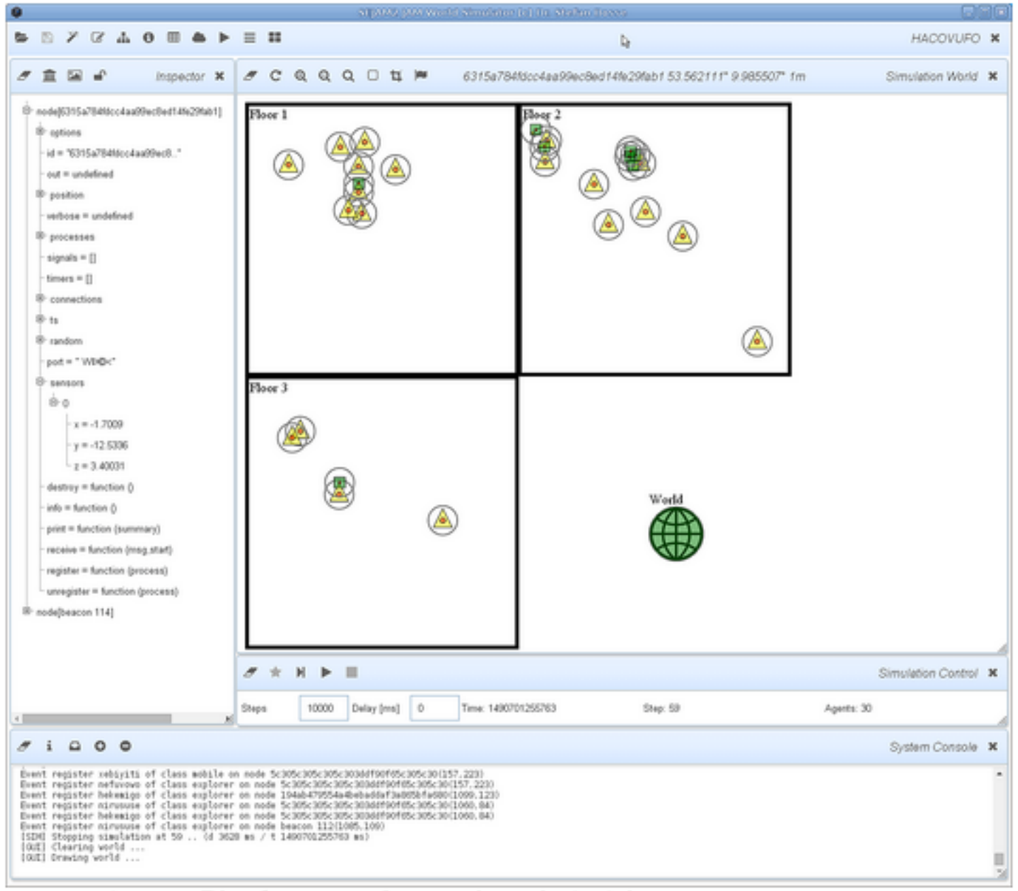
CROWD SENSING

- Crowd Sensing mit der JavaScript Agent Machine (JAM) und mobilen Agenten [2]
- JAM kann auf mobilen Geräten und stationären Stationen (Beacons) ausgeführt werden
- Agenten können nahtlos zwischen Plattformen in einem Netzwerk migrieren (“wandern”)
- Mobile Agenten werden zur Sensorfusion und Sensorverteilung genutzt



CROWD SENSING

- Simulation eines komplexen und verteilten Crowd Sensing Szenario mit SEJAM (Simulation Environment for JAM)
- Reales Ereignis: Nutzerdaten vom Chaos Communication Congress (2014) in Hamburg

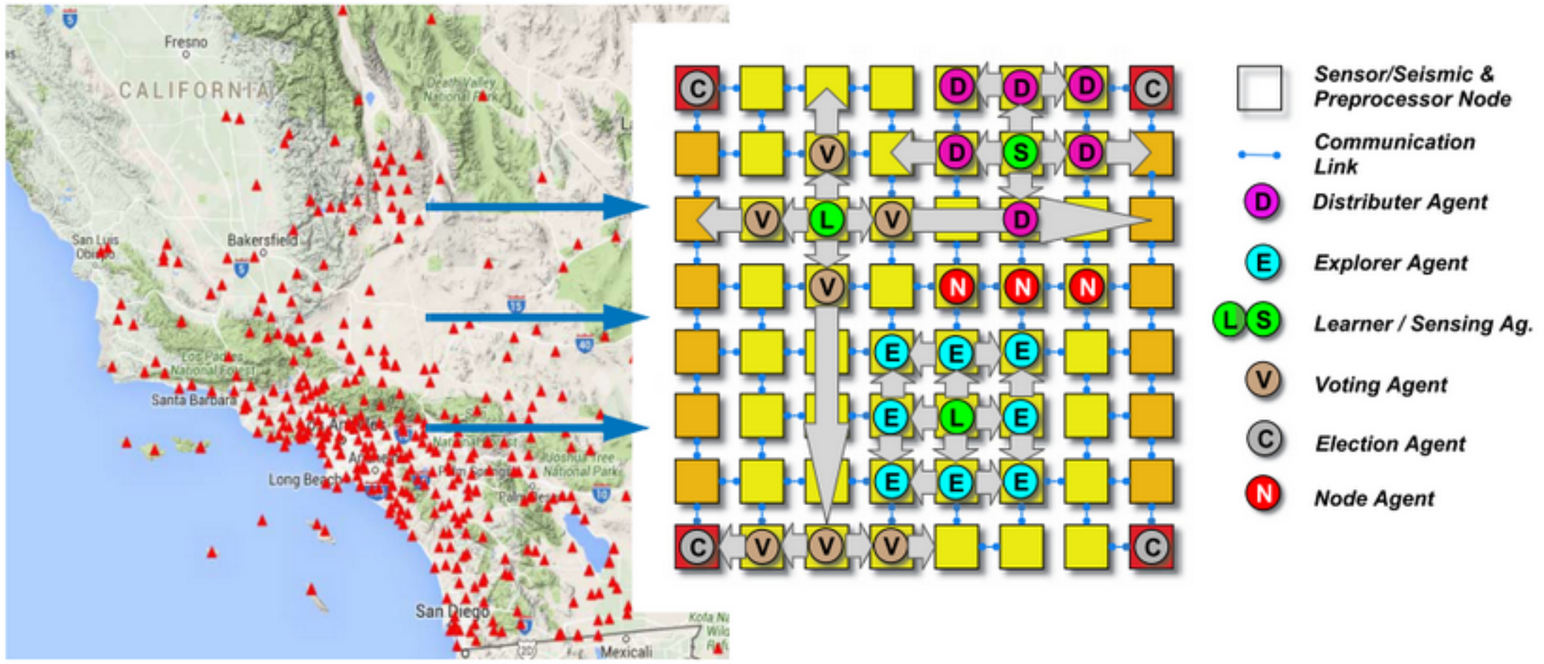


ISENSNET

iSENSNET: Intelligent Sensor Processing in Sensor Networks

Erdbebenüberwachung und Ereignisklassifizierung

- Verteiltes seismisches Netzwerk-CI (South California), das auf einem 2D Mesh-Netzwerk abgebildet ist (JAM Agentenplattform)
- Mobile Agenten werden zum verteilten und inkrementellen Lernen eingesetzt [1]



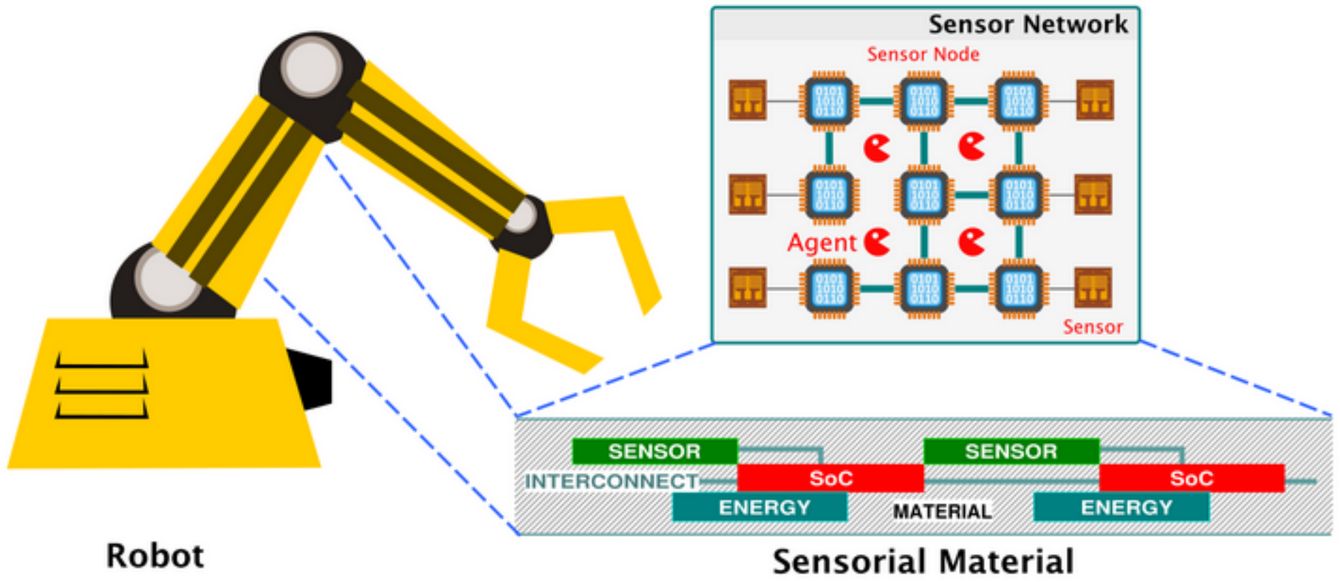
ISENSNET

- Erweiterung des seismischen Netzwerkes mit mobile Geräten wie Smartphones
- Smartphones können bei Erdbeben zusätzliche Informationen liefern
- Mobile Agenten können zwischen seismischen und mobilen Netzen/Geräten migrierien

ISENSNET

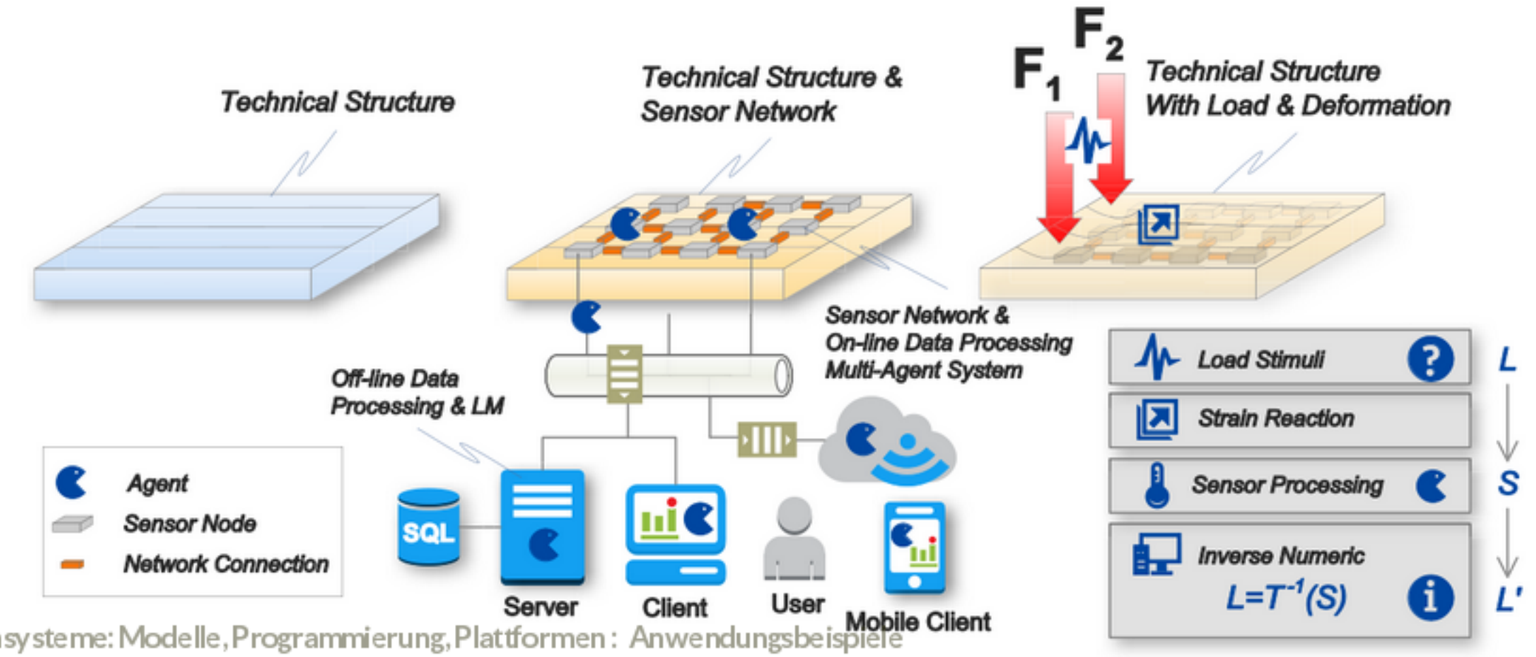
Sensorisches Material

- Materialien mit integrierten Sensornetzwerken (Sensor + Informations-Kommunikations Technologien)
- Einsatz von mobilen Agenten für die Sensorverarbeitung, Sensorverteilung, und verteiltes agentenbasiertes Lernen
- Verwendung einer low-resource Agentenplattform AFVM



ISENSNET

- Einbindung sensorischer Materialien in das Internet, das Internet der Dinge (IoT), und Cloud Umgebungen
- Anwendung:
  - Überwachung von sicherheitskritischen Strukturen, z.B. Bauwerken, in Flugzeugen, Windrädern
  - Überwachung von technischen Geräten und Produkten mit Sammlung von Ensembledaten zur "fliessenden" Verbesserung von Produkteigenschaften (Cloud-based Design & Manufacturing)



AGENTENMODELLE UND ARCHITEKTUREN

AGENTENBASIERTES MODELLIEREN (ABM)

Eine wichtige Eigenschaft des agentenbasierten Modellierens ist die Möglichkeit Zufälligkeit in die Modelle mit einzubeziehen (Monte Carlo Simulation)

- Viele numerische (gleichungsbasierte) Ansätze erfordern das Entscheidungen im Modell deterministisch ausgeführt werden sollen (wie allg. Softwareentwurf)
- In agentenbasierten Ansätzen können Entscheidungen auf Wahrscheinlichkeiten und Zufall beruhen

Beispiel: Im Ameisenmodell ist die Bewegung der Ameisen nicht vollständig vorherbestimmt und deterministisch. Jede Ameise wird auf ihrem Weg immer wieder zufällig eine kleine Richtungsänderung vornehmen.

VERHALTENSMODELLE

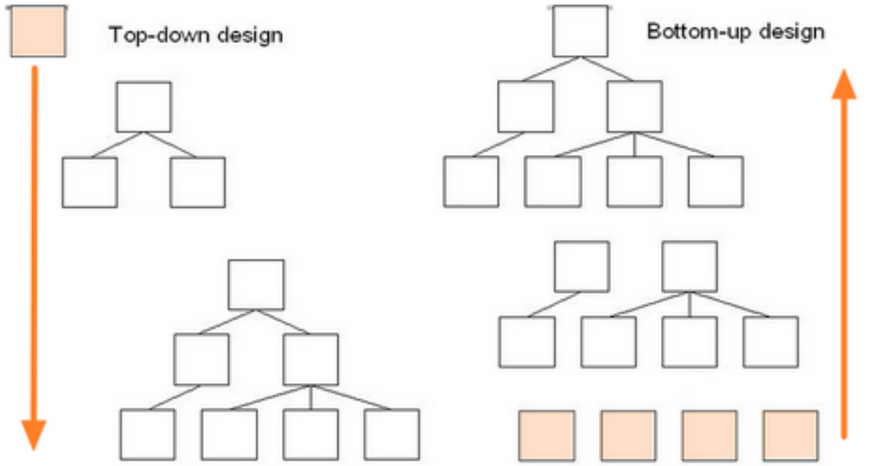
Man unterscheidet zwei grundsätzliche Methoden der Modellierung von Agenten:

Top-down Ansatz → Strukturen bilden sich nach unten

Ein **globaler** Ansatz wo die Systemebene und das Verhalten des Ensembles (bekannt) modelliert und das Verhalten von Individuen und deren Wechselwirkung untersucht (unbekannt) wird.

Bottom-up Ansatz → Strukturen bilden sich nach oben

Ein **lokaler** Ansatz wo die Individualebene und das Verhalten einzelner Agenten (bekannt) modelliert und das Systemverhalten (Emergenz, unbekannt) untersucht wird.



VERHALTENSMODELLE

Beispiel Ameisen

Ein einfaches Verhaltensmodell für Ameisen in einer Kolonie könnte textuell wie folgt beschrieben werden:

Verhalten einer Ameise

1. Wenn die Ameise kein Futter trägt dann
   - überprüft sie ob an ihrem momentanen Ort Futter zu finden ist,
   - wenn Futter gefunden wurde wird es aufgenommen; wenn nicht
   - dann sucht sie in der Umgebung nach einem Pheromon und
   - geht entlang der Pheromonspur (Gradient) bis zur stärksten Quelle
2. Wenn die Ameise Futter trägt dann geht sie zurück zum Nest und hinterlässt Pheromone entlang der Spur
3. Es wird zufällig eine Richtungsabweichung ausgewählt und einen Schritt in diese Richtung gegangen (Random walk)

VERHALTENSMODELLE

NetLogo Beschreibung

- NetLogo ist eine Modellierungssprache auf Systemebene und eine Simulationsumgebung

```
;; if not carrying food, look for it
If not carrying-food? [ look-for-food ]
;; if carrying food turn back towards the nest
if carrying-food? [ move-towards-nest ]
;; turn a small random amount and move forward
wander
```

NETLOGO - EINFÜHRUNG

Prozeduren

- Die programmatische Modellierung von Agenten in NetLogo ist prozedural. Die Definition einer **Prozedur** oder **Funktion** (mit Rückgabewert) erfolgt mit dem `to` Schlüsselwort.

Definition 1.

```
to procedure-name
  Anweisung
  Anweisung
  ..
end
```

Die Welt und Patches

- Die Simulationswelt (zweidimensional) ist in NetLogo in **patches** unterteilt
- Ein Patch hat eine Koordinate (*x*,*y*) und kann eine Ressource sein (passive Agenten)
- Ein Patch besteht aus einer *Visualisierung* (Farbe) und optional *Variablen*

NETLOGO - EINFÜHRUNG

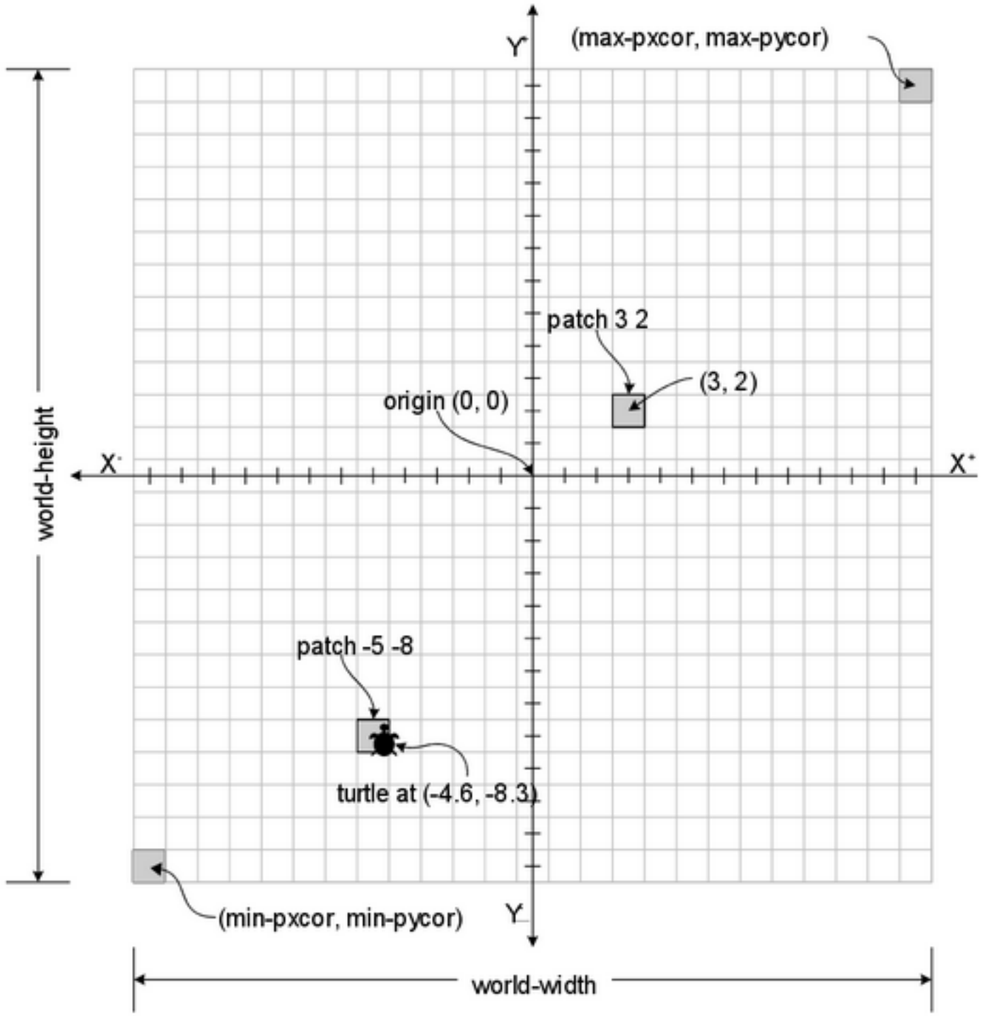


Abb. 10. Simulationswelt und kartesische Koordinaten

Patches

- Patches können einen Satz von Variablen besitzen die mit der `patches-own` Anweisung definiert werden und mit der `set` Anweisung verändert werden können.
- Vordefinierte Variablen (Parameter): `pcolor`

Definition 2.

```
patches-own [
  var1
  var2
  .. ]
..
[ set var1 ε ]
```

Anweisungen

Definition 3.

```
[ set var ε ]
if cond [
  Anweisung
  Anweisung
]
ifelse cond
  Anweisung cond=True
  Anweisung cond=False
```

NETLOGO - EINFÜHRUNG

Agenten

- In NetLogo werden Agenten durch **Turtles** implementiert
- Ein Agent ist gekennzeichnet durch seine aktuelle Patchposition, die initial gesetzt werden kann mit der Anweisung `setxy` *x y*
- Agenten werden durch die `create-turtles` Anweisung erzeugt. Agenten können private Variablen haben die mit der `turtles-own` Anweisung deklariert werden.

Definition 4.

```
turtles-own [var]
create-turtles num [
  Anweisung
  Anweisung
  [ setxy xpos ypos ]
  [ set var ε ]
  .. ]
```

NETLOGO - EINFÜHRUNG

- Ein Setup der Simulation besteht i.A. aus folgenden Schritten:
  1. Welt löschen → `clear-all`
  2. Agenten erzeugen und initialisieren → `create-turtles`
  3. Simulationszähler zurücksetzen → `reset-ticks`

Beispiel

```
to setup
  clear-all
  create-turtles 100 [ setxy random-xcor random-ycor ]
  reset-ticks
end
```

NETLOGO - EINFÜHRUNG

Aktionen

- In jedem Simulationsschritt (*tick*) können Aktionen auf Patches und Agenten (Turtles) angewendet werden
- Die Auswahl der Agenten (oder Patches) erfolgt mit der ask Anweisung die eine Menge von Anweisungen auf die ausgewählten Menge von Agenten anwendet → **Iterator**

Definition 5.

```
askagentset [
  Anweisung
  Anweisung
  .. ]
```

- Vordefinierte Mengen: `turtles`, `patches`, `turtle` *num*
- Alle Anweisung in der ask Anweisung wie z.B. das Verändern von Agentenvariablen werden auf den jeweiligen Agenten aus der zu iterierenden Menge angewendet!

NETLOGO - EINFÜHRUNG

Beispiele

```
ask turtles [ fd 1 ]
  ;; all turtles move forward one step
ask patches [ set pcolor red ]
  ;; all patches turn red
ask turtle 4 [ rt 90 ]
  ;; only the turtle with id 4 turns right
```

Mobilität

- Agenten (Turtles) können auf dem Patchfeld verschoben werden. Dazu gibt es zwei Parameter: *Richtung* und Anzahl der *Schritte* (Felder)

Definition 6.

`forward` *num*	`fd` *num*
`right` *degree*	`rt` *degree*
`left` *degree*	`lt` *degree*

NETLOGO - EINFÜHRUNG

Simulationsschleife

- Die Simulation wird in Schritten ausgeführt und durch die `tick` Anweisung wird ein Simulationszähler erhöht.
- Der Schrittzähler kann mit `ticks` abgefragt und mit `reset-ticks` zurück gesetzt werden.

```
to go                to move-turtles
  move-turtles        ask turtles [
  tick                 right random 360
end                    forward 1
                      ]
                     end
```

Agenten löschen

- Ein Agent kann aus der Simulationswelt durch die `die` Anweisung entfernt werden.
- Ist nur im Zusammenhang mit einer `ask` Anweisung anwendbar!

AGENTEN UND WELTUMGEBUNG

- Agenten interagieren mit der Weltumgebung (dem Environment) mittels:
  - » Sensoren (Daten)
  - » Aktoren (Daten)
- Sensoren und Aktoren sind über den Zyklus *Wahrnehmung* → (*Planung* →) *Entscheidung* → *Aktion* verbunden

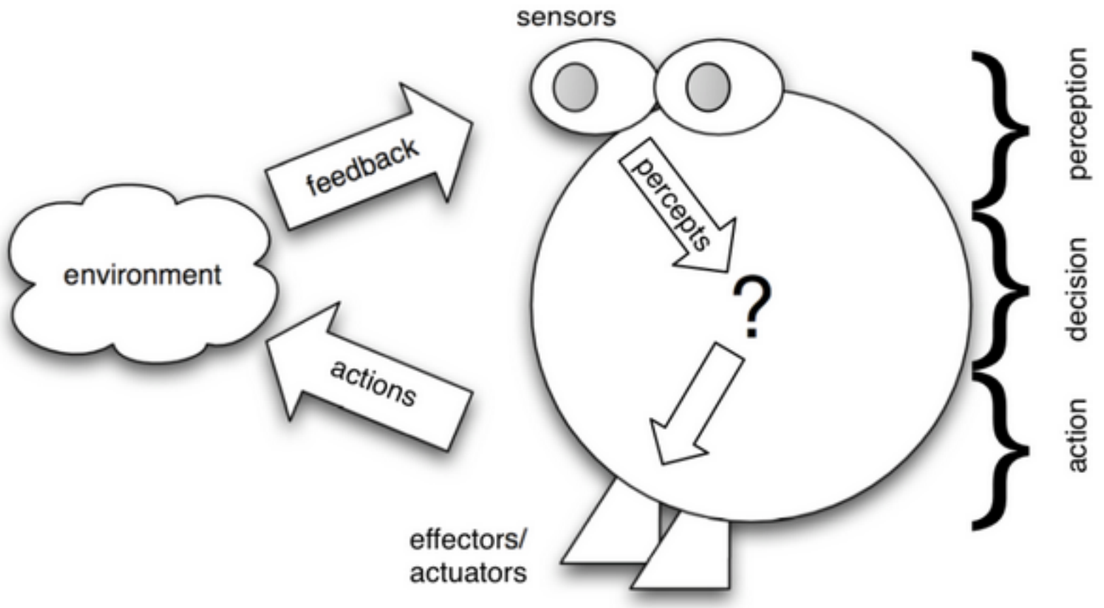


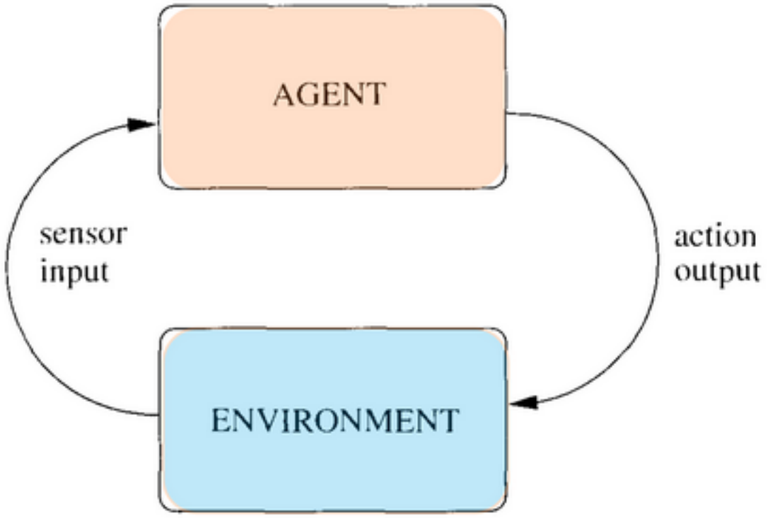
Abb. 11. Wechselwirkung von Agent und Umgebung [B]

AGENTEN UND WELTUMGEBUNG

Agentenmodell

- Ein Agent in seiner Umgebung nimmt sensorische Eingaben aus der Umgebung und produziert sie als Output-Aktionen, die ihn *und* die Umgebung beeinflussen.
- Die Interaktion ist gewöhnlich fortlaufend und nicht-terminierend!

Einfachstes Agentenmodell



[B]

Beispiel

```
Umgebung: Raum in Gebäude
Sensor: Temperatur T
Aktor:  Heizung

Verhalten:
(1) Temperatur zu niedrig → Heizung an
(2) Temperatur angenehem →  Heizung aus
```

KLASSEN VON VERHALTENSWEISEN

Das Verhalten von Agenten und ihre Modellierung wird maßgeblich durch diese Interaktion und den Zyklus bestimmt!

Verhaltensweisen

Reaktiv

Ein reaktives System ist eines, das eine fortlaufende Interaktion mit seiner Umgebung aufrecht erhält und auf Änderungen reagiert, die darin auftreten (rechtzeitig, damit die Antwort nützlich ist).

- » Auf ein Ereignis zu reagieren ist einfach: *Stimulus → Antwort*
- » Aber nützlicher (für UNS) ist ein zielgerichtetes Verhalten …

Proaktiv

Proaktivität ist die Schaffung und der Versuch Ziele zu erreichen; nicht nur von Ereignissen getrieben; die Initiative ergreifen.

- » Chancen erkennen!

KLASSEN VON VERHALTENSWEISEN

Sozial

Die reale Welt ist eine Umgebung mit einem Ensemble aus Agenten: Sie werden nur erfolgreich sein Ziele zu erreichen wenn sie andere berücksichtigen.

- Einige Ziele können nur durch Interaktion mit anderen erreicht werden.

> *Soziale Fähigkeit in Agenten ist die Fähigkeit, mit anderen Agenten (und möglicherweise Menschen) durch Kooperation, Koordination und Verhandlung zu interagieren.*

- Zumindest bedeutet es die Fähigkeit zu kommunizieren …

SOZIALE FÄHIGKEIT: KOOPERATION

- Kooperation arbeitet als Team zusammen, um ein gemeinsames Ziel zu erreichen.
- Häufig entweder durch die Tatsache, dass kein Agent das Ziel alleine erreichen kann, oder dass die Kooperation ein besseres Ergebnis erzielt (z.B. schnelleres Ergebnis).

⇒ **Emergentes Verhalten** ⇒

Arten

» **Zufällig**, *nicht beabsichtigt*
» **Einseitig** *beabsichtigt*, ein Agent hilft beabsichtigt dem anderen zur Zielerfüllung
» **Gegenseitig** *beabsichtigt*, mehrere Agenten helfen sich beabsichtigt gegenseitig

SOZIALE FÄHIGKEIT: KOORDINATION

Wettbewerb

- Koordination behandelt die Abhängigkeiten zwischen Aktivitäten.
- Wenn es beispielsweise eine nicht gemeinsam teilbare Ressource gibt (*geteilt aber nicht teilbar*), die zwei Agenten verwenden möchten, müssen sie koordinieren.

Mutualer Ausschluss

Auflösung des Wettbewerbskonflikts durch *entweder oder* (EXOR) Entscheidung.

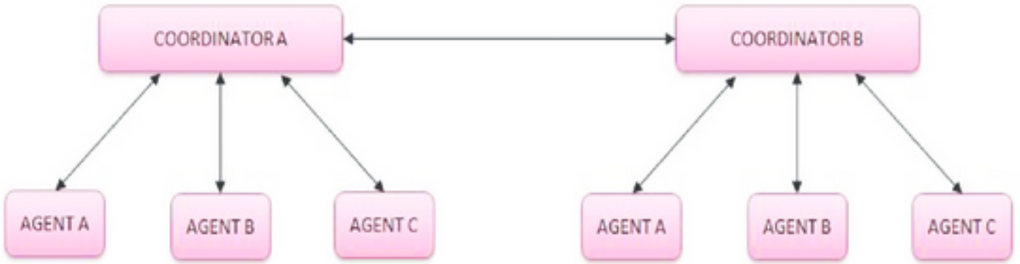
- » Koordination in verteilten asynchronen Systemen ist nicht einfach.
- » Invariante des mutualen Ausschlusses: Eine geteilte Ressource darf niemals von mehr als einem Agenten gleichzeitig verwendet werden!
- » Sichere und robuste Gruppenkommunikation ist erforderlich

SOZIALE FÄHIGKEIT: KOORDINATION

Architekturen

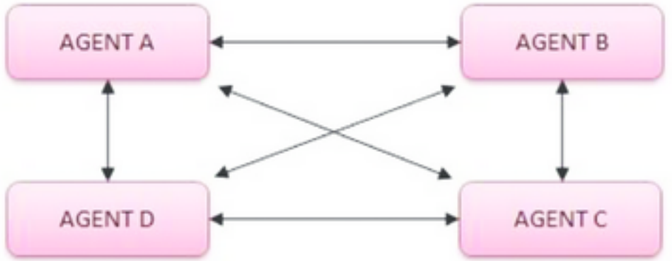
Zentralisiert und hierarchisch

Es gibt ausgewiesene (oder gewählte) zentrale Koordinatoren die Entscheidungen treffen



Dezentralisiert

Es gibt eine Selbstorganisation ohne ausgewiesene zentrale Koordination



SOZIALE FÄHIGKEIT: VERHANDLUNG

- Verhandlung ist die Fähigkeit, Vereinbarungen über Angelegenheiten von einem gemeinsamen Interesse zu treffen.
- Zum Beispiel: Sie haben einen Fernseher in Ihrem Haus; Sie wollen einen Film sehen, Ihr Mitbewohner möchte Fußball schauen.
  - » Ein möglicher Deal: Schau heute Fußball und morgen einen Film.
  - » Mein bevorzugter Deal: Es wird immer nur ein Film geschaut (Ego als Motivation ;-)!
- Beinhaltet in der Regel Angebot und Gegenangebot mit Kompromissen der Teilnehmer.

FORMALES MODELL VON AGENTEN

Umgebung (Welt)

- Die Welt soll aus einer endlichen Menge von Zuständen bestehen:

$$E = \{e_1, e_2, .., e_k\}$$

» Die reale Welt ist kontinuierlich → aber jede Welt kann mit hinreichender Genauigkeit diskretisiert werden
» Weiterhin betrachten wir ohnehin hier *digitale Welten*
» Die Umgebung hat eine *Historie* und ist i.A. nicht *deterministisch*

Aktionen

- Jeder Agent kann eine Aktion *a* (Aktivität) aus einer endlichen Menge von Aktionen *A* ausführen:

$$Ac = \{a_1, a_2, .., a_n\}$$

FORMALES MODELL VON AGENTEN

Durchlauf

- Eine Ausführung eines Agenten mit einer iterativen Ausführung von Aktionen wird die Welt in ihrem Zustand schrittweise verändern (Durchlauf oder run):

$$r \in R : e_a \xrightarrow{a \in Ac} e_b \xrightarrow{a \in Ac} e_c \xrightarrow{a \in Ac} ..$$

Agenten

- Ein Agent *Ag* ist eine **Abbildungsfunktion** die jede Durchführung $r \in R$ mit einem Endzustand $e \in E$ der Welt auf eine Aktion $a \in Ac$ abbildet:

$$Ag : R^E \rightarrow Ac$$

» Agenten wählen Aktionen in Abhängigkeit vom Weltzustand aus.
» Die Welt ist undeterministisch; aber ein Agent soll sich deterministisch verhalten!

REAKTIVE AGENTEN

Modell

- Das allgemeine Wahrnehmung-Aktions Modell kann in Subsysteme unterteilt werden:
  - » Eine Funktion *see*
  - » Eine Prozedur *action* die auf den Ergebnissen der Funktion *see* Aktionen auswählt

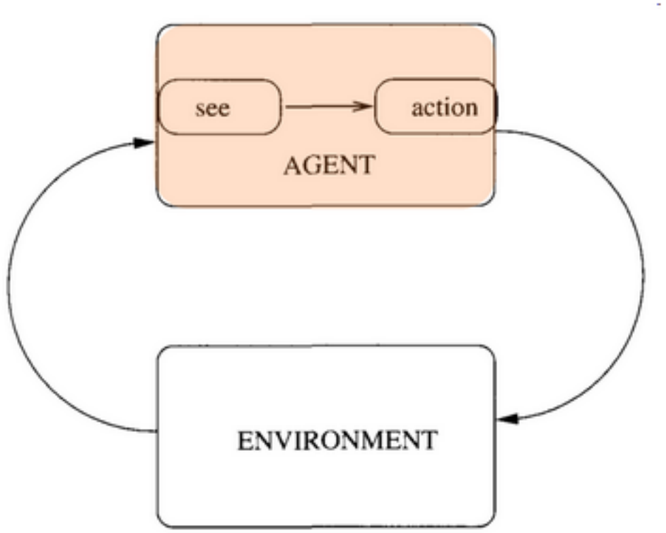


Abb. 12. Verkettete Wahrnehmungs- und Aktionssubsysteme

REAKTIVE AGENTEN

- Rein reaktive Agenten werden eine Aktion *a* nur auf dem *aktuellen* Zustand *e(t)* ∈ *E* basierend auswählen:

$$Ag_r : E \rightarrow Ac$$

» Die Historie der Welt wird nicht in Betracht gezogen

Beispiel: Heizungsagent

$$Ag(T) : \begin{cases} \text{Heizung aus, wenn } T > T_0 \\ \text{Heizung ein, wenn } T \leqslant T_0 \end{cases}$$

REAKTIVE AGENTEN

Wahrnehmung

- Die Wahrnehmung kann mit einer Funktion *see* modelliert werden:

$$see : E \rightarrow Per, \text{ mit}$$
$$\{Sen_0, Sen_1, ..\} \Rightarrow Per$$

» *Per* ist die Menge aller Wahrnehmungen (Perzeption) die der Agent durch seine Sensoren erhält, und *see* bildet Weltzustände auf Perzeptionen ab.

Aktion/Ausführung

- Die Aktionen die ein Agent auswählt basieren auf der Perzeption:

$$action : Per* \rightarrow Ac$$

REAKTIVE AGENTEN

Beispiel Heizungsagent 2.0

- Zwei Sensoren:
  A. Temperatur mit S_T={Hoch=1, Niedrig=0}
  B. Luftfeuchtigkeit mit S_H={Hoch=1, Niedrig=0}
- Die Perzeptions- und Aktionsfunktion lauten dann:

$$e = e(S_T, S_H), E = \{e_1 : (0,0), e_2 : (0,1), e_3 : (1,0), e_4 : (1,1)\}$$

$$see(e) = \begin{cases} p_1, \text{ wenn } e = e_1 \vee e = e_2 \vee e = e_4 \\ p_2, \text{ wenn } e = e_3 \end{cases}$$

$$action(p) = \begin{cases} a_1 = Heizen, \text{ wenn } p = p_1 \\ a_2 = \neg Heizen, \text{ wenn } p = p_2 \end{cases}$$

WELTZUSTAND UND PERZEPTION

- **Problem: Ein Agent sieht immer nur einen Ausschnitt des Weltzustandes über seine Perzeption!**

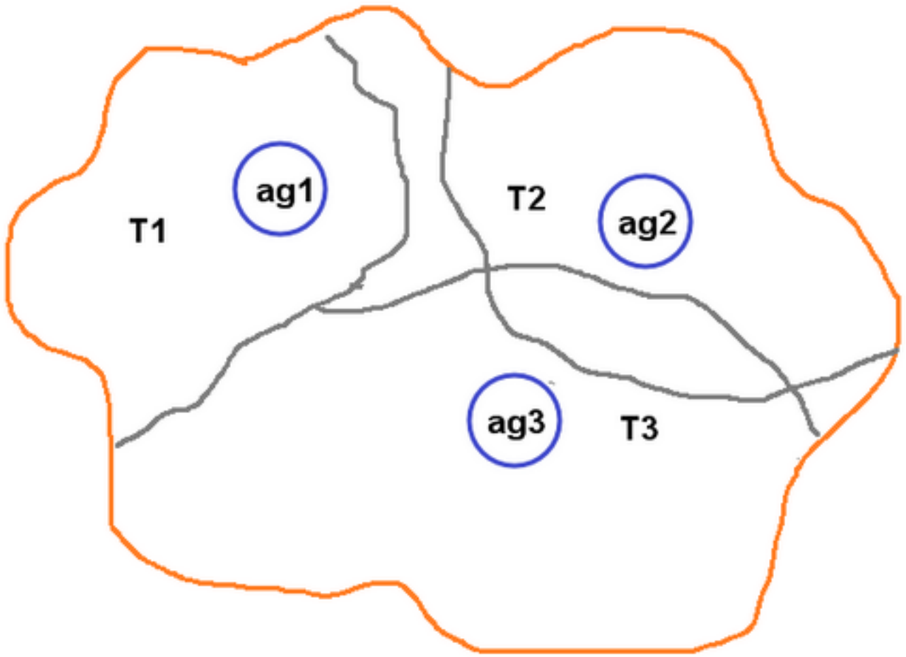
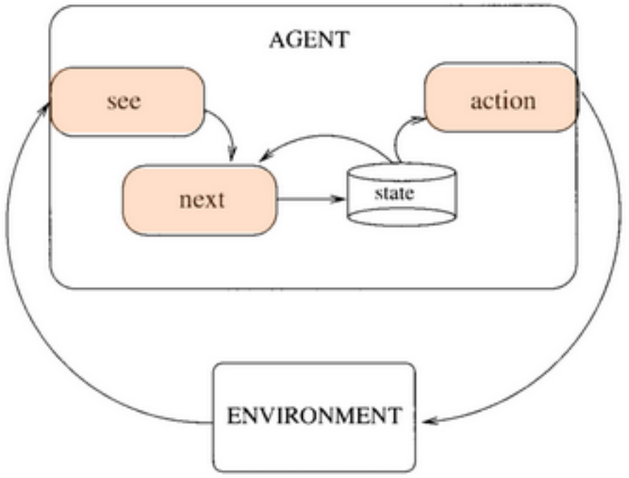


Abb. 13. Die Welt besteht aus drei Agenten und zwei Bereichen mit den Perzeptionen T_1,T_2, und T_3. Der Weltzustand $e_i \in E(T_1,T_2,T_3,ag_1,ag_2,ag_3)$ wird von den einzelnen Agenten unterschiedlich und nur teilweise wahrgenommen (Perzeption $ag_1=\{T_1\}$, $ag_2=ag_3=\{T_2,T_3\}$)

REAKTIVE ZUSTANDSBASIERTE AGENTEN

Modell

- Das Wahrnehmung-Aktions Modell kann durch einen Zustandsspeicher erweitert werden und besteht dann aus:
  - Einer Wahrnehmungsfunktion *see*
  - Einem Zustandsspeicher *state*
  - Einer Zustandsübergangsfunktion *next* die den nächsten Zustand berechnet (aus Ergebnissen von *see* und *state*)
  - Einer Prozedur *action* die Aktionen nach dem aktuellen Zustand auswählt



REAKTIVE ZUSTANDSBASIERTE AGENTEN

Interne Zustände

- Aktionen werden nicht direkt aus Perzeptionen abgeleitet. Stattdessen werden interne Zustände $I=\{i_1,i_2,..\}$ verwendet.

Zustandsübergang

- Neben der *see* und *action* Funktion gibt es eine *next* Funktion die die internen Zustände und die aktuellen Zustand auf interne Zustände abbildet:

$$see : E \rightarrow Per$$
$$next : I \times Per \rightarrow I$$
$$action : I \rightarrow Ac$$

REAKTIVE ZUSTANDSBASIERTE AGENTEN

Verhalten

- Das Verhalten eine zustandsbasierten reaktiven Agenten ist wie folgt:
  1. Der Agent startet in einem Anfangszustand e_0
  2. Nachdem der aktuelle Weltzustand e aufgenommen wurde berechnet er $p=see(e)$
  3. Der nächste interne Zustand des Agenten wird bestimmt: $i_{n+1} := next(i_n,p)$
  4. Der Agent wählt eine Aktion aus: $action(next(i_n, p))$
  5. Schleife nach 2.

Verhaltensäquivalenz

» Zustandsbasierte Agenten sind nicht ausdrucksstärker als Standardagenten (ohne Zustand) (Wooldridge) !?

NÜTZLICHKEITSFUNKTION

- Wie soll ein Agent bewerten dass seine Aktionen sinnvoll und nützlich sind?
- Eine Nützlichkeitsfunktion *u* kann verwendet werden um die Aktionsauswahl und die Konvergenz des Agenten seine Ziele zu erreichen zu verbessern.
- Mögliche Nützlichkeitsfunktionen können Weltzuständen (zu den der Agent auch gehört) oder Durchläufen eine reelle Bewertungzahl zuordnen:

$$u : E \rightarrow \mathbb{R}$$
$$u : R \rightarrow \mathbb{R}$$

» Letztere kann die Historie einbeziehen, wohingegen die erste Funktion immer nur den aktuellen Zustand bewertet.

NÜTZLICHKEITSFUNKTION

Fliesenwelt ist ein Beispiel für eine dynamische Welt!

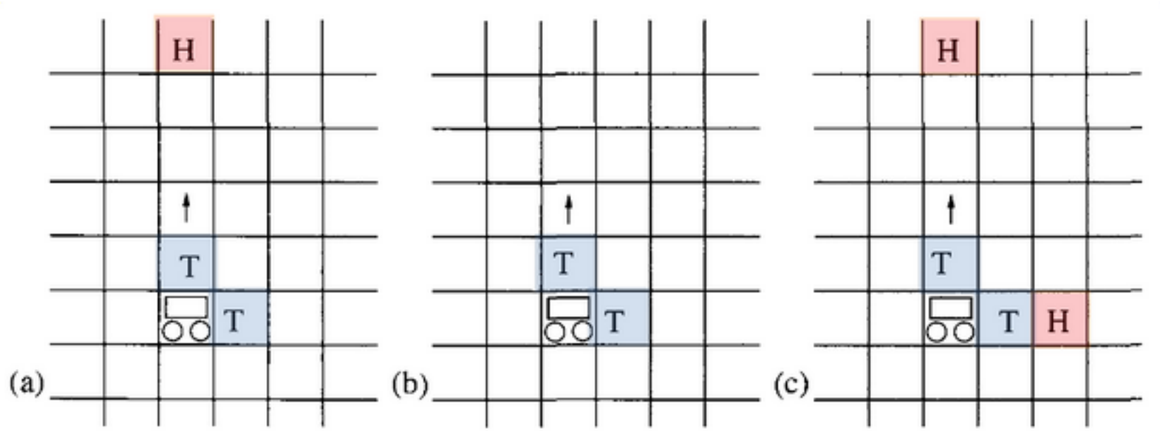


Abb. 14. Ein Agent in einer 'Fliesenwelt' mit Fliesen (T) und Löchern (H): Verschiedene Situationen (a) - (c)

a. Ein Agent detektiert ein Loch und beginnt mit der Aktion Verschiebung einer Fliese
b. Das Loch verschwindet bevor der Agent mit seiner Tätigkeit fertig ist. Der Agent muss auf die veränderte Umgebung reagieren und neue Aktionen planen.

NÜTZLICHKEITSFUNKTION

c. Es taucht während der Aktion Verschiebung einer Fliese ein weiteres Loch auf. Der Agent sollte auf die veränderte Situation reagieren indem er z.B. erst eine Fliese nach rechts schiebt (näher dran)

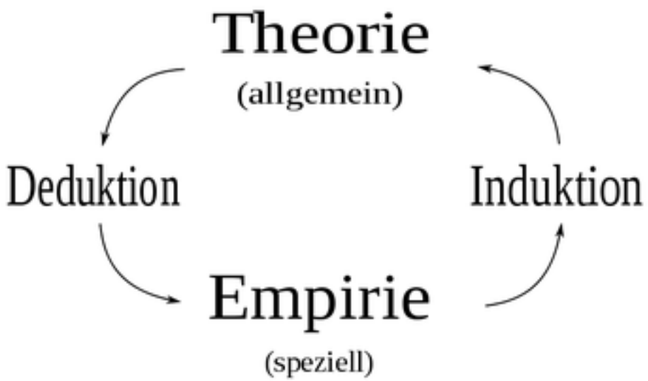
- Veränderte Weltumgebungen führen bestenfalls zu eine Reorganisation von Aktionen.
- Kann mit Rückgekoppelten Lernen kombiniert werden, d.h. die Verbesserung der Aktionsplanung durch "Belohnung" (wenn mehr Löcher gefüllt)
- Eine mögliche Nützlichkeitsfunktion könnte das Verhältnis der aufgetauchten Löcher zu den erfolgreich während eines Durchlaufs *r* gefüllten in Relation stellen und das Agentenverhalten beeinflussen:

$$u(r) = \frac{Num(\{h \in H | filled\})}{Num(H)}$$

- Dieses Funktion *u(r)* liefert eine *normalisierte Performanzmessung* der Aktionen und Aktionsplanung des Agenten.

DEDUKTIVE AGENTEN

- **Deduktion** oder deduktiver Schluss: Ableitung aus Wissen und Logiken



- Dabei spielen Logik und formale Systeme eine wesentliche Rolle:

```
p     (Prämisse 1)
p → q     (Prämisse 2)
———
q     (Konklusion)
```

» D.h., sind p und p → q (sprich: wenn p, dann q) wahre Aussagen, so ist auch q eine wahre Aussage.

DEDUKTIVE AGENTEN

- Traditionell werden künstliche intelligente Systeme mit symbolischer KI umgesetzt.
- Intelligentes Verhalten entsteht bei diesem Ansatz durch:
  - **Symbolische Repräsentation** der Umgebung mit dem gewünschten Verhalten
  - **Syntaktische Veränderung** dieser symbolischen Repräsentation
- Deduktive Schlussfolgerung bedeutet: Agenten als Theorembeweiser!
- Häufig sind symbolische Repräsentationen **logische Formulierungen**

Sei L die Menge der Sätze der klassischen Logik erster Ordnung und sei $D = \phi(L)$ der Satz von L-Datenbanken, d. h. die Menge von Mengen von L-Formeln. Der innere Zustand eines Agenten ist dann ein Element Δ von D. Der Entscheidungsprozess eines Agenten wird durch eine Reihe von Deduktionsregeln modelliert, ρ. Dies sind einfache Schlussfolgerungsregeln (Inferenz) für die Logik.

DEDUKTIVE AGENTEN

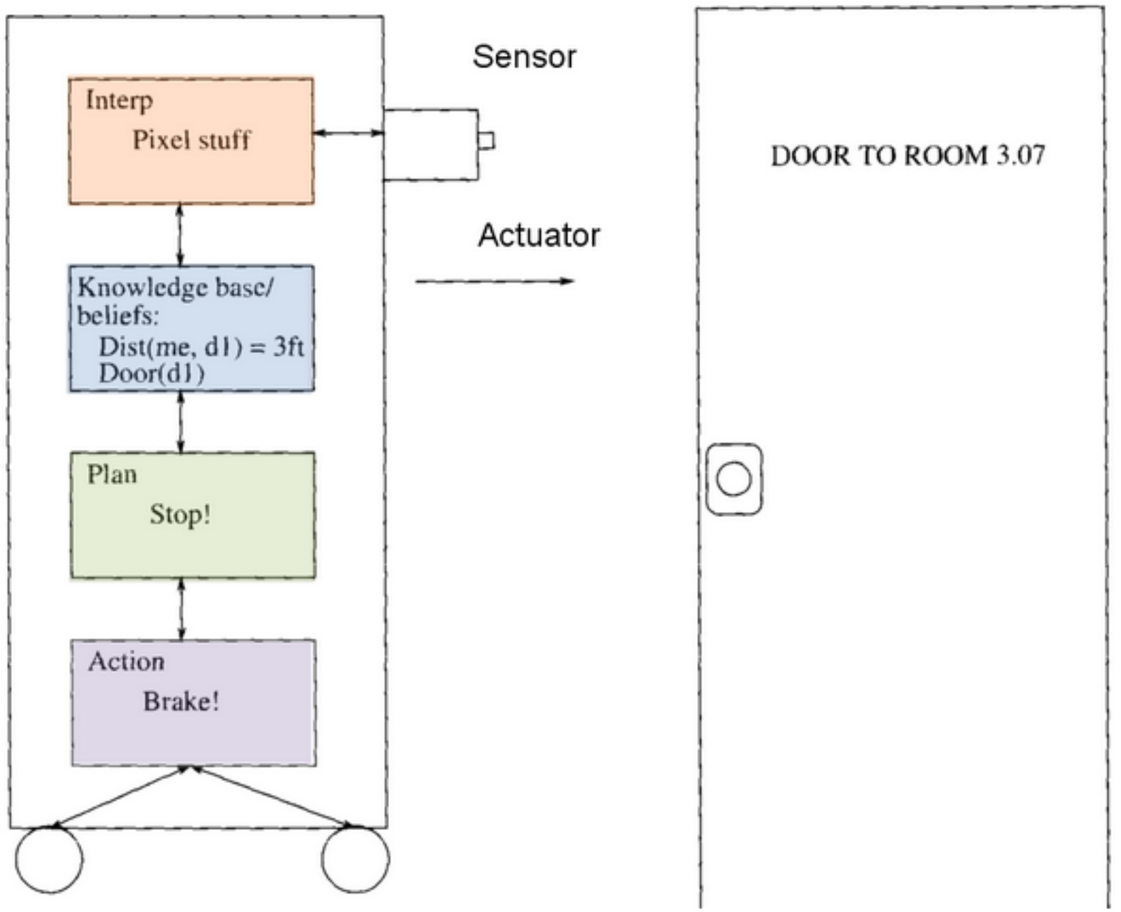


Abb. 15. Beispiel eines robotischen Agenten der eine symbolische Beschreibung der Welt besitzt [B]

DEDUKTIVE AGENTEN

- Am Anfang steht wieder die **Perzeption** über Sensoren und **Interpretation**
- Die Information des Agenten über die Welt ist in einer Datenbank enthalten → **Wissensbasis**
- Die Wissensbasis enthält Aussagen die entweder schon vorhanden sind oder aktuell/kürzlich aufgrund der Wahrnehmung erworben wurden und veränderlich sind (wie z.B. Abstand Roboter-Tür)
- Die Wissensbasis wird benutzt um einen Plan auszuwählen → **Planung**
- Schließlich werden nach der Planung geeignete **Aktionen** ausgeführt

Definition 7.
Verhalten deduktiver Agenten:

Perzeption → Interpretation → Wissen → Planung → Aktion

DEDUKTIVE AGENTEN

- Es gibt bei der Implementierung eines solchen Agenten folgende Probleme:

Transduktion
: Das Problem, die reale Welt in eine genaue, adäquate symbolische Beschreibung der Welt zu übersetzen, damit diese Beschreibung für den Agenten verwendbar und nützlich ist.

Repräsentation/Schlussfolgerung
: Das Problem, Informationen symbolisch darzustellen und Agenten dazu zu bringen, damit zu manipulieren / zu argumentieren, damit die Ergebnisse nützlich sind.

- Das Transduktionsproblem beschäftigt sich im wesentlichen mit Vision, Spracherkennung, Lernen, usw.
- Das Repräsentationsproblem behandelt Wissensrepräsentation (Ontologien!), Automatisches Schlussfolgern, Automatisches Planen usw.

DEDUKTIVE AGENTEN

- Das Agentenmodell erweitert sich durch die Datenbank zu:

$$see : S \rightarrow Per$$
$$next : D \times Per \rightarrow D$$
$$action : D \rightarrow Ac$$

- Die *next* Funktion verändert jetzt die Wissensdatenbank durch Perzeption!
- *Wenn* es eine Formel *Do(α)* für Aktionen $\alpha \in Ac$ mittels einer Deduktionsregel ρ driekt aus der Datenbank abgeleitet werden kann, *dann* wähle diese Aktion als beste aus
- *Wenn* das nicht möglich ist (keine Aktion gefunden) und wenn es eine Formel ¬*Do(α)* gibt die sich derzeit nicht aus der Datenbank ableiten lässt (d.h. es ist nicht verboten es zu tun), *dann* wähle diese Aktion aus.

DEDUKTIVE AGENTEN

```
Function: Action Selection as Theorem Proving
1.  function action(Δ : D) returns an action Ac
2.  begin
3.      for each α ∈ Ac do
4.          if Δ ⊢ρ Do(α) then
5.              return α
6.          end-if
7.      end-for
8.      for each α ∈ Ac do
9.          if Δ ⊬ρ ¬Do(α) then
10.             return α
11.         end-if
12.     end-for
13.     return null
14. end function action
```

Abb. 16. Aktionsauswahl als Theoremprüfung [B]

DEDUKTIVE AGENTEN

Beispiel: Staubsaugeragent

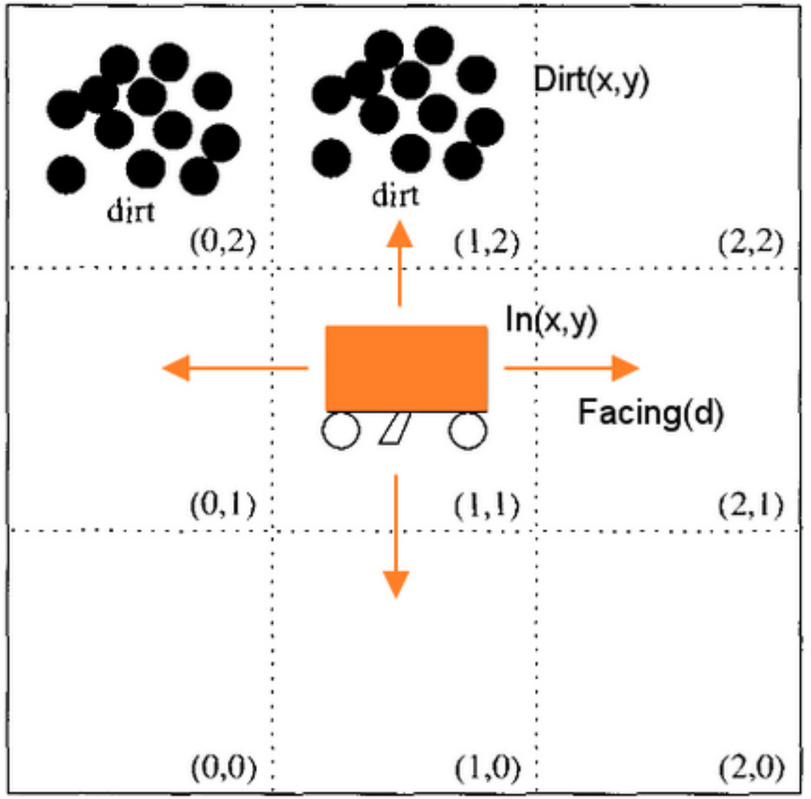


Abb. 17. Ein Agent in einer zweidimensionalen diskreten Welt aus Feldern (Position *x,y*) mit den Attributen *Dirt(x,y)*, *In(x,y)*, *Facing(dir)*

DEDUKTIVE AGENTEN

- Es gibt folgende Bereichsprädikate (Attribute):

```
In(x, y)   → Agent ist bei (x, y),
Dirt(x, y) → Dreck bei (x, y),
Facing (d) → Ausrichtung des Agenten
```

- Es gibt folgende Aktionen: *A={forwärts,saugen,drehen}*
- Nun werden alte Informationen der Datenbank *old*(Δ) identifiziert und mittels einer *new* Funktion durch die *next* Funktion aktualisiert:

$$\Delta \in D$$
$$old(\Delta) = \{P(t_1, .., t_n) | P \in \{In, Dirt, Facing\} \wedge P(t_1, .., t_n) \in \Delta\}$$
$$new : D \times Per \rightarrow D$$
$$next(\Delta, p) = (\Delta \backslash old(\Delta)) \cup new(\Delta, p)$$

DEDUKTIVE AGENTEN

- Regeln in der Wissensdatenbank (Auswahl):

$$In(x,y) \wedge Dirt(x,y) \rightarrow Saugen$$
$$In(x,y) \wedge \neg Dirt(x,y) \rightarrow Forwaerts(?), \text{ zu ungenau, genauer}$$
$$In(0,0) \wedge \neg Dirt(x,y) \wedge Facing(Nord) \rightarrow Forwaerts$$
$$..$$
$$In(0,2) \wedge \neg Dirt(x,y) \wedge Facing(Nord) \rightarrow Drehen$$
$$In(0,2) \wedge Facing(Ost) \rightarrow Forwaerts$$
$$..$$

- **Problem**: Wenn sich zwischen zwei Zeitpunkten t_1 und t_2 die Welt ändert und eine bei t_1 ausgewählte Aktion nicht mehr aktuell und optimal ist!
- *Agenten die strikt auf Logiken basieren sind nicht besonders performant und optimal (bzw. können völlig versagen)*

TEMPORALLOGIK

Operator	Meaning
$\bigcirc \varphi$	φ is true 'tomorrow'
$\bullet \varphi$	φ was true 'yesterday'
$\Diamond \varphi$	at some time in the future, φ
$\Box \varphi$	always in the future, φ
$\blacklozenge \varphi$	at some time in the past, φ
$\blacksquare \varphi$	always in the past, φ
$\varphi \mathcal{U} \psi$	φ will be true until ψ
$\varphi \mathcal{S} \psi$	φ has been true since ψ
$\varphi \mathcal{W} \psi$	φ is true unless ψ
$\varphi \mathcal{Z} \psi$	φ is true zince ψ

Abb. 18. Zeitliche Zusammenhänge von Bedingungen und Aussagen werden durch Operatoren beschrieben [B]

AGENTENORIENTIERTES PROGRAMMIEREN: AGENT0

- In *AGENT0* (eine Programmiersprache) wird eine Agent beschrieben durch:
  - Fähigkeiten (Capabilities, Aktionen)
  - Eine Menge von Wissen über ihre Umwelt (Beliefs)
  - Eine Menge initialer Verpflichtungen und Bindungen(Commitments)
  - Eine Menge Verpflichtungsregeln (Commitment Rules)
  - **Nachrichtenaustausch**
- Die Schlüsselkomponente, die bestimmt, wie der Agent handelt, ist der Commitment-Regelsatz.
  - Jede Commitment-Regel enthält eine Nachrichtenbedingung, eine mentale Bedingung und eine Aktion.
  - Die Nachrichtenbedingung wird mit den Nachrichten abgeglichen, die der Agent empfangen hat.
  - Der mentale Zustand wird mit den Überzeugungen des Agenten verglichen. Wenn die Regel ausgelöst wird, wird der Agent an die Aktion gebunden.

AGENTENORIENTIERTES PROGRAMMIEREN: AGENT0

- Aktionen können privat/intern sein oder kommunikativ → **Nachrichten**
- Nachrichtentypen:
  - Anfragen (Requests)
  - Abbruch von Anfragen (Unrequests)
  - Informativ (ohne Antwort)
- Die Anfragen verändern i.A. die Commitments (Ziele) des Agenten,
- Die informativen Nachrichten das Wissen (Beliefs)
- Beispiel:

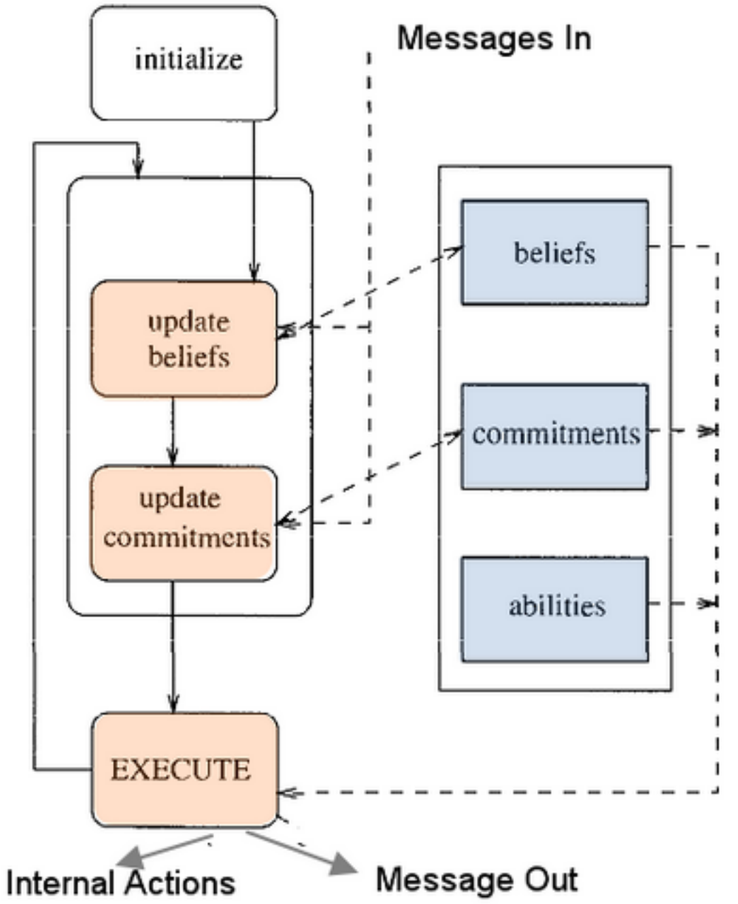
```
COMMIT(
( agent, REQUEST, DO(time, action)
), ;;; msg condition
( B,
  [now, Friend agent] AND
   CAN(se1f , action) AND
   NOT [time, CMT(se1f , anyaction)]
), ;;; mental condition self,
DO(time, action) )
```

AGENTENORIENTIERTES PROGRAMMIEREN: AGENT0

Textuelle Formulierung

'if I receive a message from agent which requests me to do action at time, and I believe that agent is currently a friend; I can do the action; at time, 1 am not committed to doing any other action, then commit to doing action at time.'

Nachrichtenbasierter Kontrollfluss in AGENT0



BELIEF-DESIRE-INTENTIONS ARCHITEKTUR

- Die BDI Architektur implementiert **Praktische und Prozedurale Schluss-folgerungsagenten**

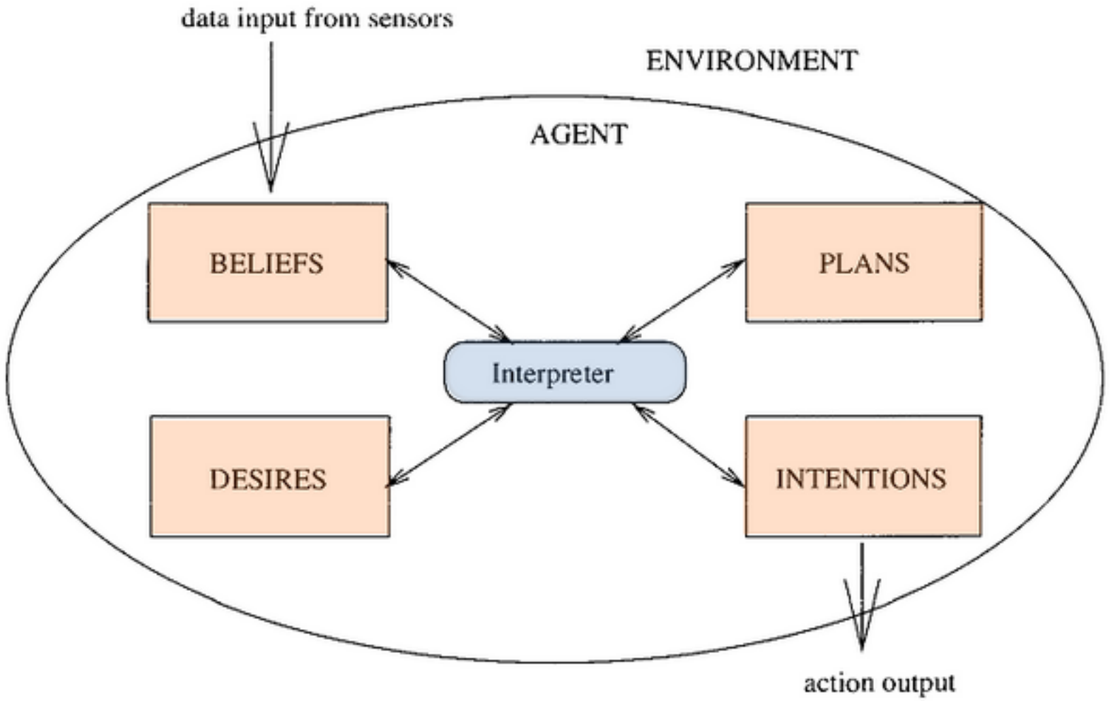


Abb. 19. BDI Architektur und prozedurale Schlussfolgerung über einen Interpreter

BELIEF-DESIRE-INTENTIONS ARCHITEKTUR

Die Belief, Desire, und Intentionsdatenbanken werden über vier Funktionen miteinander im Interpreter verknüpft: ***Belief Revision Function BRF***, ***Filter***, ***Option Generation Function***, *und die* ***Action Selection Function***

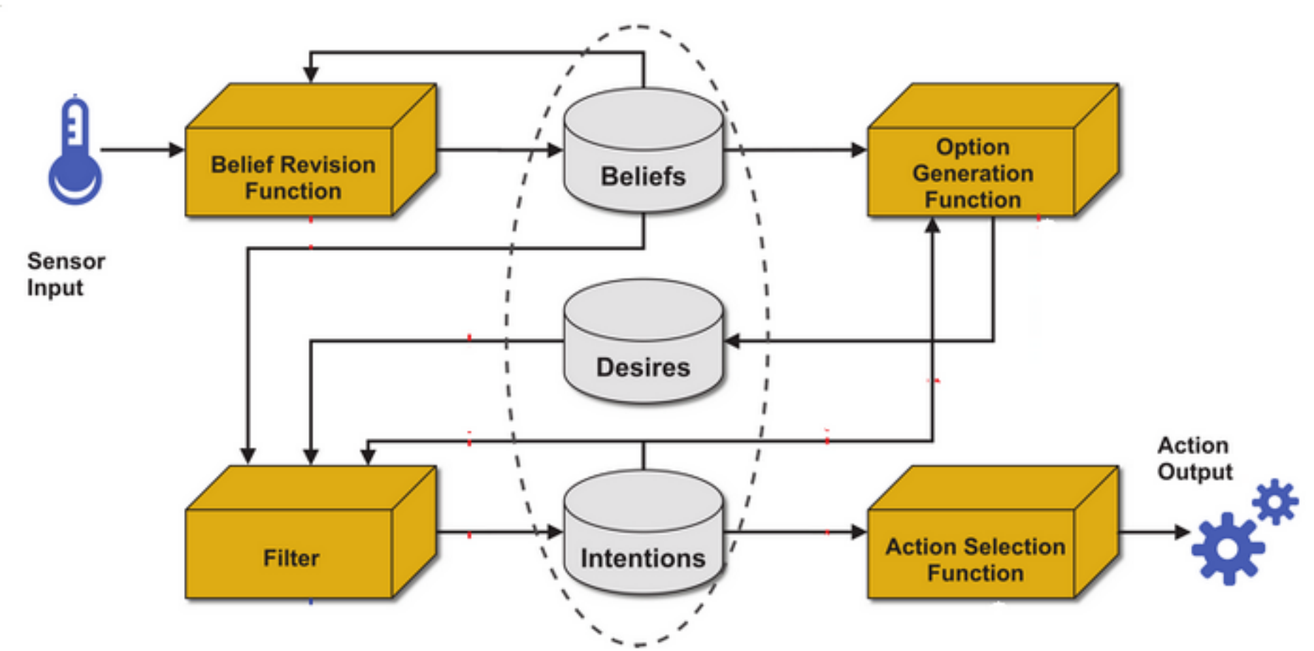


Abb. 20. Verfeinerte BDI Architektur

BELIEF-DESIRE-INTENTIONS ARCHITEKTUR

BDI Kontrollschleife

- Der Agent empfängt Ereignisse, die entweder
  - extern (aus der Umgebung, aus perzeptuellen Daten)
  - intern generiert sind
- Er versucht, Ereignisse zu behandeln, indem nach Plänen gesucht wird, die übereinstimmen
- Die Menge von Plänen, die dem Ereignis entsprechen, sind Optionen / Wünsche
- Wählt einen Plan aus um seine Absichten (Wünsche) zu erfüllen: wird dem dann verpflichtet - eine Absicht
- Wenn ein Plan ausgeführt wird, können neue Ereignisse generiert werden, die eine Behandlung erfordern

BELIEF-DESIRE-INTENTIONS ARCHITEKTUR

Beliefs

- Überzeugungen stellen Informationen dar, die der Agent über seine Umgebung hat
- Sie sind symbolisch dargestellt
  - » Logik erster Ordnung

Plans

- Im Vorfeld von den Entwicklern offline kodiert
- Gibt dem Agenten Informationen darüber,
  - » wie er auf Ereignisse reagiert
  - » wie er Ziele erreicht
- Planstruktur: `triggering_event : context <- body`
  - » Ereignis (`trigger_event`)
  - » Kontext (`context`)
  - » Körper (Implementierung `body`)

BELIEF-DESIRE-INTENTIONS ARCHITEKTUR

- Bedeutung: *Wenn das `trigger_event` gesehen wird, und geglaubt wird, dass der `context` wahr ist, dann kann der `body` ausgeführt werden!*

Ereignis/Triggerbedingung
: Ist ein Ereignis, mit dem der Plan umgehen kann

Kontext
: Definiert die Bedingungen, unter denen der Plan verwendet werden kann

Körper
: Definiert die Aktionen, die ausgeführt werden sollen, wenn der Plan ausgewählt wird

Ereignisse

+!*P* : Neues Ziel *P* erworben → “erreiche P”

-!*P* : Ziel *P* verworfen

+*B* : Neue Annahme *B* hinzufügen

-*B* : Annahme *B* verwerfen

BELIEF-DESIRE-INTENTIONS ARCHITEKTUR

BDI Kontrollschleife

```
1. B ← B0;/* B0 are initial beliefs */
2. I ← I0;/* I0 are initial intentions */
3. while true do
4.   get next percept ρ via sensors;
5.   B ← brf (B,ρ);
6.   D ← options(B, I);
7.   I ← filter(B,D, I);
8.   π ← plan(B, I , Ac); /* Ac is the set of actions */
9.   while not (empty(π) or succeeded(I , B) or impossible(I , B)) do
10.    α ← first element of π;
11.    execute(α);
12.    π ← tail of π;
13.    observe environment to get next percept ρ;
14.    B ← brf (B, ρ);
15.    if reconsider(I , B) then
16.      D ← options(B, I);
17.      I ← filter(B, D, I);
18.    end-if
19.    if not sound(π , I , B) then
20.      π ← plan(B, I , Ac)
21.    end-if
22.  end-while
23. end-while
```

BELIEF-DESIRE-INTENTIONS ARCHITEKTUR : AGENTSPEAK

- *AgentSpeak* ist eine BDI-basierte Programmiersprache für den Entwurf rationaler Agenten mit logischen Formeln

Beispiel: Hello World Agent

```
/* Initial beliefs and rules */
/* Initial goals */
!start.
/* Plans */
+!start : true <- .print("hello world.").
```

Beispiel: Faktorisierungsagent

```
/* Initial belief */
fact(0,1).
/* Additional belief: if x<5 then add new belief */
+fact(X,Y) : X < 5 <- +fact(X+1, (X+1)*Y).
/* If x==5 then print result */
+fact(X,Y) : X == 5 <- .print("fact 5 == ", Y).
```

BELIEF-DESIRE-INTENTIONS ARCHITEKTUR : AGENTSPEAK

Beispiel: Faktorisierungsagent 2

```
/* Initial and single goal */
!print_fact(5).
/* If this is the goal, first compute fact, then print it */
+!print_fact(N) <- !fact(N,F);
  .print("Factorial of ", N, " is ", F).
/* How to compute factorial */
+!fact(N,1) : N == 0.
+!fact(N,F) : N > 0 <- !fact(N-1,F1); F = F1 * N.
```

Kommunikation in AgentSpeak

- Z.B. gibt es zwei Agenten: Der eine weiß wie man die Faktorisierung berechnet, der andere nicht.
- Der Experte wird vom Idioten Anfragen empfangen und wird darauf antworten → **Client-Server Architektur**

BELIEF-DESIRE-INTENTIONS ARCHITEKTUR : AGENTSPEAK

Definition 8.

```
.send(rcvr, type, content)
.broadcast(type,content)
```

» Mit der .send Operation wird eine Nachricht an den Agenten *rcvr* mit dem Nachrichtentyp *type* und dem Inhalt *content* gesendet.

» Der Nachrichtentyp *type* ist dabei ein Typ aus der Liste:

```
type ∈ {tell,untell,achieve,unachieve,
                askOne,askAll,askHow, tellHow,untellHow}
```

- *tell/untell*: Hinzufügen/Entfernen von Annahamen
- *achieve/unachieve*: Hinzufügen/Entfernen von zu erreichenden Zielen
- *askOne,askIf*: Ziel testen

```
.send(obi_wan, askOne, ?father(luke), Answer)
.send(obi_wan, askIf, ?father(luke))
```

BELIEF-DESIRE-INTENTIONS ARCHITEKTUR : AGENTSPEAK

Beispiel: Der Idiot

```
/* Initial goals */
!start.
/* Plans */
+!start :
  true <- .print("starting..");
    !query_factorial(2);
    !query_factorial(4);
    !query_factorial(6);
    !query_factorial(10).

+!query_factorial(X) :
  true <- .send(expert, tell, giveme(X)).

+fact_result(X,Y) : true <-
  .print("factorial ", X, " is ", Y, " thank you expert").
```

BELIEF-DESIRE-INTENTIONS ARCHITEKTUR : AGENTSPEAK

Beispiel: Der Experte

```
+!giveme(X)[source(A)]:
  true <- !fact(X,Y);
          .send(A,tell,fact_result(X,y);
          .print("Factorial of ", X, " is ", Y).

+!fact(X,1) : X == 0.

+!fact(X,Y) : X > 0
    <- !fact(X-1,Y1);
    Y = Y1 * X.
```

KOMMUNIKATION UND INTERAKTION

Agent-Agent und Agent-Welt Kommunikation

SHARED MEMORY

- Agenten sind parallele und verteilte Systeme!
- Eine der einfachsten Kommunikationsmodelle und Architekturen für parallele Systeme ist der **geteilte Speicher**.
- Das Shared-Memory-Modell ist ein häufig verwendetes Interprozesskommunikationsparadigma für parallele, weniger für verteilte Systeme. Es ist eng verwandt mit dem Parallelregister und Random-Access-Maschine (PRAM),
- Das PRAM-Modell nimmt n identische Verarbeitungseinheiten (PU) an, die mit einer Shared-Memory-Ressource über Direktzugriff (SRAM) verbunden sind.
  - » Speicherzellen haben gleiche Breite, die durch eine numerische Adresse referenziert werden.
  - » Das SRAM-Modell unterstützt konkurrierende Lese- und Schreiboperationen.
  - » Lesen ist i.A. konfliktfrei, Schreiben kann zu Konflikte führen.

SHARED MEMORY

- Agenten können über den geteilten Speicher Daten austauschen!
- Aber: Es gibt Datenaustausch ohne *explizite Synchronisation* zwischen den Agenten.
- Und: Das SRAM Modell 'verletzt' das Autonomieparadigma von Agenten!

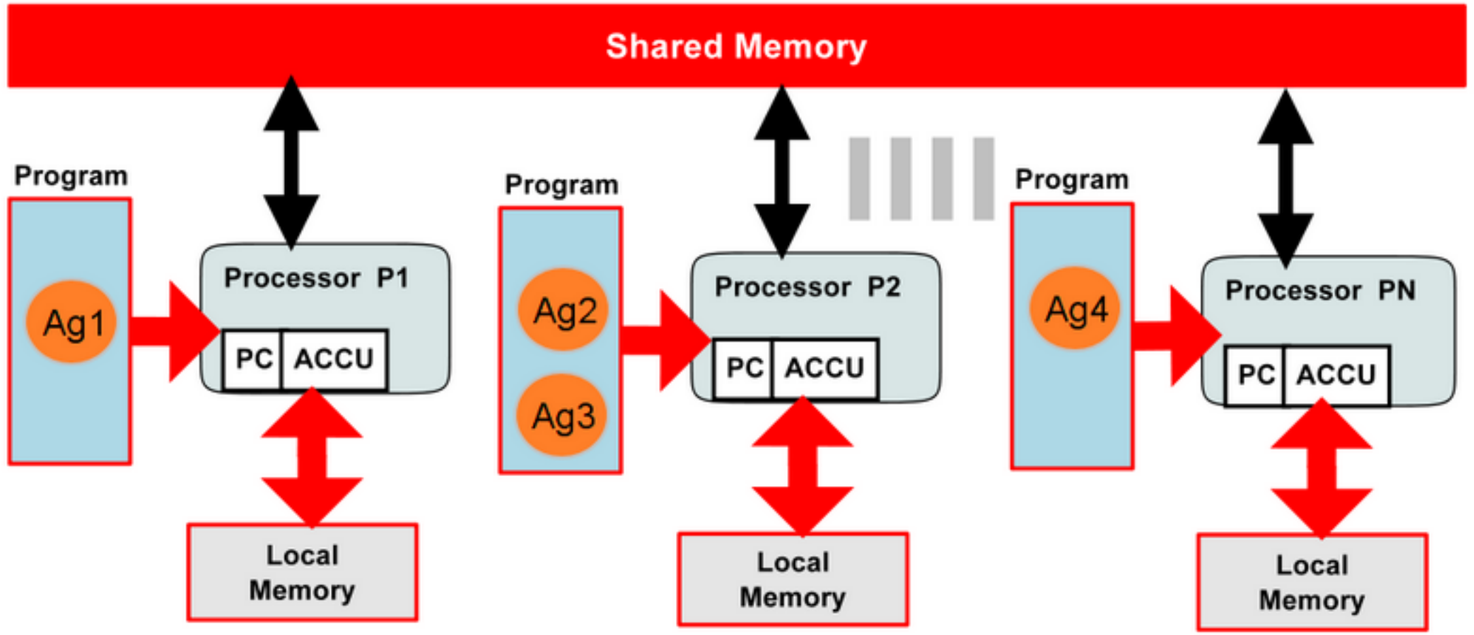
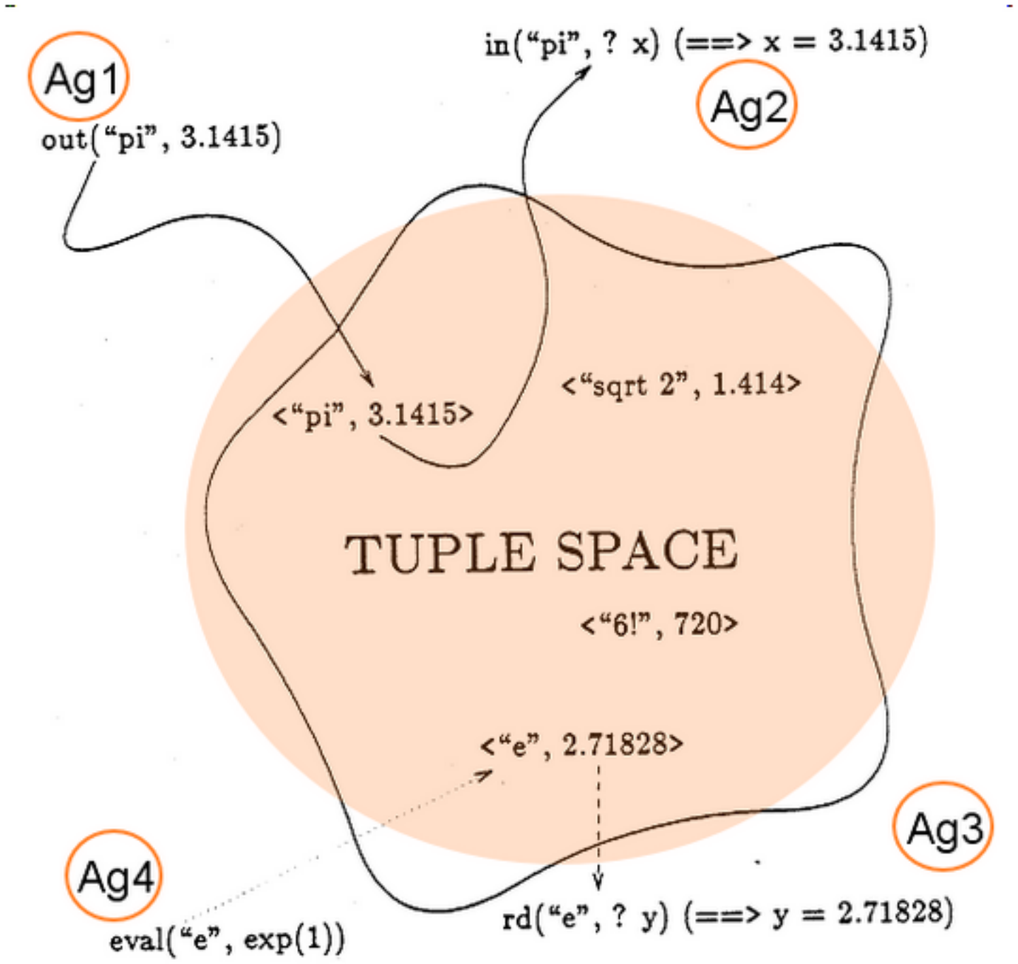


Abb. 21. Das PRAM Modell und die Verknüpfung mit Agenten die auf den PUs ausgeführt werden.

TUPELRÄUME

- Tupel-Räume stellen ein **assoziiertes Shared-Memory-Modell** dar, wobei die gemeinsam genutzten Daten als **Objekte** mit einer Reihe von **Operationen** betrachtet werden, die den Zugriff der Datenobjekte unterstützen
- Tupel sind in **Räumen** organisiert, die als abstrakte Berechnungsumgebungen betrachtet werden können.
- Ein Tupelraum verbindet verschiedene Programme, die **verteilt** werden können, wenn der Tupel-Space oder zumindest sein operativer Zugriff verteilt ist.
  - » Oder: **Mobile Agenten** als Tupel Verteiler!
- Das Tupelraum Organisations- und Zugangsmodell bietet **generative Kommunikation**, d.h. Datenobjekte können in einem Raum durch Prozesse mit einer Lebensdauer über das Ende des Erzeugungsprozesses hinaus gespeichert werden.
- Ein bekanntes Tupelraum-Organisations- und Koordinationsparadigma ist **Linda** [GEL85].

TUPELRÄUME



[3]

Abb. 22. Ein Schnappschuss eines Tupelraumes mit Tupeln und Tupeloperationen

TUPELRÄUME

- Kommunikation von Agenten über Tupelräume ist eine **Koordinations-sprache.**

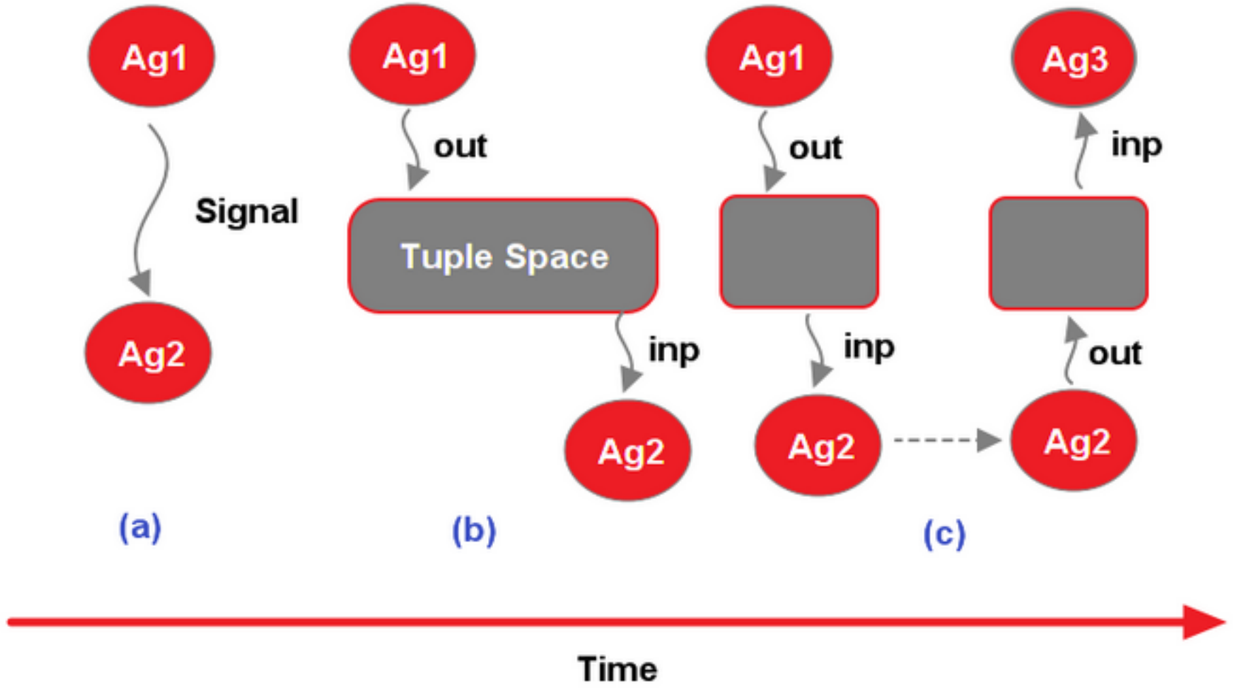


Abb. 23. Direkter Nachrichtenaustausch (a), z.B. durch Signale, im Vergleich zu generativer Kommunikation (b) und virtuelle verteilte Räume (c) durch mobile Prozesse (Agenten)

TUPELRÄUME - DATENMODELL

- Die Daten sind mit Tupeln organisiert.
- Ein Tupel ist eine lose gekoppelte Verbindung einer beliebigen Anzahl von Werten beliebiger Art /Typ/
- Ein Tupel ist ein Wert und sobald es in einem Tupelraum gespeichert ist, ist es persistent.
- Tupeltypen ähneln den Datenstrukturtypen, sie sind jedoch dynamisch und können zur Laufzeit ohne statische Beschränkungen erstellt werden.
- Auf die *Elemente von Tupeln* kann nicht direkt zugegriffen werden, was üblicherweise Mustererkennung und *musterbasierte Dekomposition* erfordert, im Gegensatz zu Datenstrukturtypen, die einen benannten Zugriff auf Feldelemente bieten, obwohl die Behandlung von Tupeln als Arrays oder Listen diese Beschränkung lösen kann.
- Ein Tupel mit *n* Feldern heißt *n*-stellig und wird in der Notation $<v_1, v_2, ..>$ angegeben.

TUPELRÄUME - DATENMODELL

Beispiele

```
<'SENSOR',1000>
<'SENSOR2',[10,100,2]>
<1,3,5,7,11>
<'SLEEPMODE',True,2500.34>
<0,'OFF'>
<1,'ON'>
```

- Formal werden Tupel als **Vektoren** durch die folgende Generierungsregel mit *Werten v*, *Ausdrücken* **ε** und *Variablen x* definiert, die als tatsächliche Parameter betrachtet werden (d.h. Variablen *x*, die mit Wertesemantik verwendet werden):

$$t = \left\langle \vec{d} \right\rangle, \text{ with } \vec{d} ::= d|d, \vec{d} \text{ and } d ::= v|\varepsilon|x$$

TUPELRÄUME - DATENMODELL

- Tupelwerte erfordern einen **Mustervergleich** basierend auf dem *Vorlagenmuster* mit der folgenden Generierungsregel, bestehend aus tatsächlichen (v, ε, x) und formalen Parametern ($x?$, Variablen, die mit Referenzsemantik verwendet werden):

$$p = \left\langle \overrightarrow{dt} \right\rangle, \text{ with } \overrightarrow{dt} ::= dt | dt, \overrightarrow{dt} \text{ and } dt ::= v | \varepsilon | x | x? | \perp$$

- Ein Suchmuster kann ein Wildcard ($\perp$) anstelle von formalen Parametern verwenden.
- Jedes Tupel t hat eine Typsignatur Sig (t) = S_t = <T_1; T_2; ...; T_n>, ein Tupel mit der gleichen Stelligkeit wie t, das den Typ jedes Tupelfeldes angibt.
- Ein Tupel kann nur durch seine Verknüpfung mit Templates p angesprochen werden.

TUPEL RÄUME - DATENMODELL

- Üblicherweise wird das **erste Feld** eines Tupels als symbolischer **Schlüssel** behandelt, der eine Tupelunterklasse identifiziert, indem Textzeichenfolgen oder aufgezählte symbolische Konstantenwerte verwendet werden.

Mustersuche

Definition 9.
Sei $t = <d_1, d_2, .., d_n>$ ein Tupel, $p = <dt_1, dt_2, .., dt_m>$ eine Vorlage; dann wird t durch p abgedeckt (bezeichnet durch match (t, p) = true), wenn die folgenden Bedingungen gelten: (i) m = n. (ii) $\forall\ dt_i = d_i$ oder $dt_i = \perp$, $1 \leq i \leq n$. Bedingung (1) prüft, ob t und p die gleiche Stelligkeit haben, während (2) prüft, ob jedes Nicht-Wildcard-Feld von p gleich ist dem entsprechenden Feld von t.

TUPELRÄUME - OPERATIONALE SEMANTIK

- Es gibt eine Reihe von Operationen, die von Prozessen angewendet werden können, bestehend aus
  - einer Reihe reiner Datenzugriffsoperationen, die Tupel als passive Datenobjekte behandeln,
  - und Operationen, die Tupel als eine Art von aktiven Rechenobjekten behandeln (genauer gesagt, zu berechnende Daten).
  - RPC-Semantik (Remote Procedure Call).

out(*t*)

Die Ausführung der Ausgabeoperation fügt das Tupel *t* in den Tupelraum ein. Mehrere Kopien desselben Tupelwerts können eingefügt werden, indem die Ausgabeoperation iterativ angewendet wird. Die gleichen Tupel können nach dem Einfügen in den Tupelraum nicht unterschieden werden.

Beispiel: `out("Sensor",1,100); out("Sensor",2,121);`

TUPELRÄUME - OPERATIONALE SEMANTIK

inp(*p*)

Die Ausführung der Eingabeoperation entfernt ein Tupel *t* aus dem Tupelraum, der der Mustervorlage *p* entspricht. Wenn kein passendes Tupel gefunden wird führt das zu einer Blockierung des aufrufenden Prozesses bis ein passendes Tupel eingestellt wird.

Beispiel: `inp("Sensor",1,s1?); inp("Sensor",i?,s?);`

rd(*p*)

Die Ausführung der Leseoperation gibt eine Kopie eines Tupels *t* zurück, dass der Vorlage *p* entspricht, entfernt sie jedoch nicht. Wenn kein passendes Tupel gefunden wird führt das zu einer Blockierung des aufrufenden Prozesses bis ein passendes Tupel eingestellt wird.

Beispiele: `rd("Sensor",1,s1?); rd("Sensor",i?,s?);`

TUPELRÄUME - OPERATIONALE SEMANTIK

inp?(*p*), rd?(*p*)

Nichtblockierende Version von *inp/rd*. Wird kein passendes Tupel gefunden wird die Operation ergebnislos terminiert.
Beispiel: `res:=inp?('SENSOR',a?,b?);`

inpw?(*tmo*,*p*), rdw?(*tmo*,*p*)

Teilblockierende Version von *inp/rd*, Wird einer Zeit von *tmo* kein passendes Tupel gefunden wird die Operation abgebrochen.
Beispiel: `res:=inpw?(1000,'SENSOR',a?,b?);`

- Die Verwendung von zeitlich unbegrenzt blockierenden Operationen kann unter Betrachtung der Lebendigkeit von Agenten nachteilig sein. Daher sollte immer eine zeitliche Begrenzung und anschließende Abfrage des Operationsstatus erfolgen (abgebrochen?)

TUPELRÄUME - OPERATIONALE SEMANTIK

test(t), testandset(*p*,function (*t*)→*t*)

Nicht blockierender Test eines Tupels und atomare Veränderung eines Tupels, dass der Vorlage *p* entspricht. Das zweite Argument ist eine Abbildungsfunktion. Das Ergebnistupel ersetzt das ursprüngliche.

Markierungen

- Tupel sind persistent und können für immer in einem Tupelraum verbleiben!
- Daher ist die Verwendung von *Markierungen* häufig sinnvoll.
- Eine Markierung ist ein Tupel mit einer Lebenszeit τ
- Nach Ablauf der Lebenszeit wird das Tupel - sofern es nicht entfernt wurde - durch einen Garbagecollector entfernt.

$$m = \left\langle \tau, \overrightarrow{d} \right\rangle, \text{ with } \overrightarrow{d} ::= d|d, \overrightarrow{d} \text{ and } d ::= v|\varepsilon|x, \ \tau : \text{timeout}$$

mark(*tmo*,*t*)

Ausgabe eines Tupels *t* mit einer Lebenszeit τ (im Tupelraum).

TUPELRÄUME - OPERATIONALE SEMANTIK

eval(*p*)

Diese Operation ermöglicht die Injektion von **aktiven Tupeln**, die derzeit nicht vollständig ausgewertet sind, indem ein erweitertes funktionales Tupel *t* verwendet wird (mit erweitertem dt :: = *v* | **ε** | *x* | *f* (*x*) mit einem Funktionsargument). *Das Tupel wird erst bei Bedarf in einem eigenen Prozess ausgewertet (durch inp oder rd Operation initiiert)*

Diese Operation nimmt eine Funktion *f* (*x*) an, die in den Prozessen vorhanden ist, die am Tupleraum teilnehmen und die für die vollständige Berechnung dieser Tupel verwendet werden kann.

Alternative Implementierung mit *eval* (Klientenseite) und *listen* und *reply* Operationen (Serverseite):

```
P1: eval("square",2,y?)
P2: def sq = fun x -> x*x;
    listen("square",x?,?);
    y=sq(x);
    reply("square",x,y);
```

TUPELRÄUME - SYNCHRONISATIONSMODELL

- Es gibt **Produzenten- (Generator) und Verbraucherprozesse**.

1. Ein *Produzent* erzeugt ein Tupel, das von einem Konsumentenprozess entfernt werden kann.
   - » Die Tupelausgabeoperation endet unmittelbar (asynchron), alternativ nachdem das Tupel im Tupelraum gespeichert wurde (synchron).
2. Ein Verbraucher-Prozess wird blockiert, wenn die Anfrage nicht bearbeitet werden kann, wenn im Tupel-Bereich tatsächlich kein passendes Tupel vorhanden ist.
3. Nachdem ein übereinstimmendes Tupel im Tupelraum gespeichert wurde, wird es sofort einen der wartenden Verbraucherprozesse zugewiesen.

» Daher ist die Eingabeoperation immer synchron. Einzige Ausnahme sind die nicht permanent blockierenden Versionen, die das Warten auf eine obere Zeitgrenze begrenzen (Timeout).

» Es gibt keine anfängliche zeitliche Anordnung von Erzeuger- und Verbraucheroperationen.

TUPELRÄUME - BEISPIELE

VERTEILTE TUPELRÄUME

- Die *Verteilung* von Tupel-Räumen auf verschiedenen Rechnerknoten impliziert *Synchronisationsprobleme* und erfordert normalerweise eine zuverlässige *Gruppenkommunikation*, die in Rechnernetzwerken nicht erwartet werden kann.
- Die Verteilung von Tupel-Räumen bedeutet die Verteilung und asynchrone Ausführung einer Menge von Tupelraum-Servern anstelle eines einzelnen Servers.
- Ein Tupelraum-Server bietet die notwendige Koordination für gleichzeitige oder verschachtelte In / Out-Anfragen.
  - » Die Verteilung der Server führt zu einer Verteilung der Koordination.
  - » Dieses Problem kann jedoch gelöst werden, indem der Tupelraum in **Unterräume** partitioniert wird und jeder Unterraum auf einem anderen Knoten von einem Server bedient wird.
  - » Problem: Tupel sind nicht gleichmäßig verteilt → schlechtes Load Balancing!

VERTEILTE TUPELRÄUME

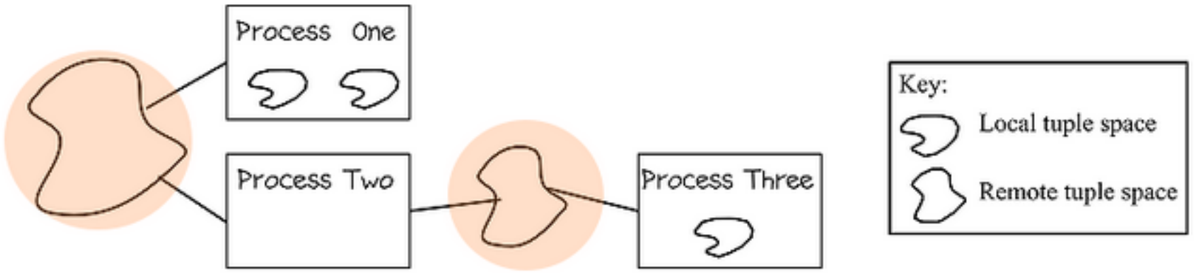


Abb. 24. Zusammenhang von lokalen und entfernten (verteilten) Tupelräumen

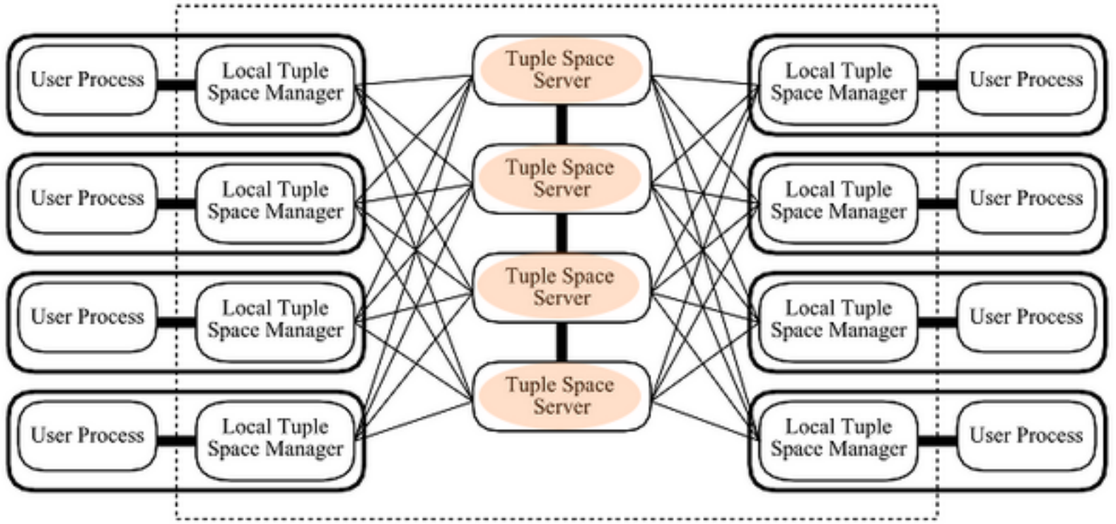


Abb. 25. Lokale und multiple globale Tupelraumserver

KOMMUNIKATIONSSIGNALE

- **Signale** sind einfache Nachrichten die
  - An einen bestimmten Prozess oder Agenten gerichtet sein kann → **Unicast**;
  - An eine bestimmte Gruppe von Prozessen oder Agenten → **Multicast**;
  - Oder an unbestimmte Gruppe von Prozessen oder Agenten in der Umgebung → **Broadcast**
- Ein Signal besteht aus einem Signaltyp (Nummer, Zeichenkette usw.) und einem (optionalen) Datenteil (Signalargument)
- Ein Signal kann gesendet und empfangen werden.
- Häufig wird auf den Empfang eines Signals nicht aktiv gewartet sondern mittels **Signalhandlern**, die eingehende Signale *asynchron* verarbeiten.
- Ein Agent kann verschiedene Signale aus einer Menge $\boldsymbol{S}=[Sig_1,Sig_2,..\}$ verarbeiten
- Für jedes Signal muss ein Signalhandler installiert werden!

KOMMUNIKATIONSSIGNALE

Verarbeitung von Signalen

```
signal SIGNAL1,SIGNAl2;
Ag1:
  on(SIGNAL1, function (arg,from) {
   do process signal SIGNAL1 from sender });
  on(SIGNAL2, function (arg,from) {
   do process signal SIGNAL2 from sender });
Ag2:
  send(Ag1,SIGNAL,ε;
```

- Signale sind gekennzeichnet durch das Tupel *<Absender,Empfänger,Name,Wert>*
- Nachteil gegenüber Tupeln: Der Agent kann die eingehenden Nachrichten nicht filtern bezüglich
  - » Inhalt
  - » Relevanz und Interesse
  - » Absender

KOMMUNIKATIONSSIGNALE

KOMMUNIKATIONSSIGNALE

- Anders als bei Tupeln muss der Empfänger (Agent) dem Absender (Agent) bekannt sein (Agentenreferenz)
  - » Eltern-Kind Beziehungen werden daher häufig für Unicast Signale verwendet
  - » Gruppenbeziehungen können durch Agentenklassen und räumliche Bereiche entstehen

Routing

- Signale adressieren mobile Agenten die ihren Standort, d.h. die Plattform, wechseln können
- Der Vermittlung (Routing) der Signalnachrichten kommt daher besondere Bedeutung zu.
- Eine Möglichkeit eine Signalnachricht zwischen zwei Agenten *A* und *B* zu vermitteln ist die Vermittlung entlang des Pfades von *A* und *B* (*Spuren*)
  - » Wenn die Spuren von *A* und *B* sich irgendwo kreuzen (Kreuzungspunkte sind die Plattformen) so kann die Nachricht über den Kreuzungspunkt zugestellt werden

KOMMUNIKATIONSSIGNALE

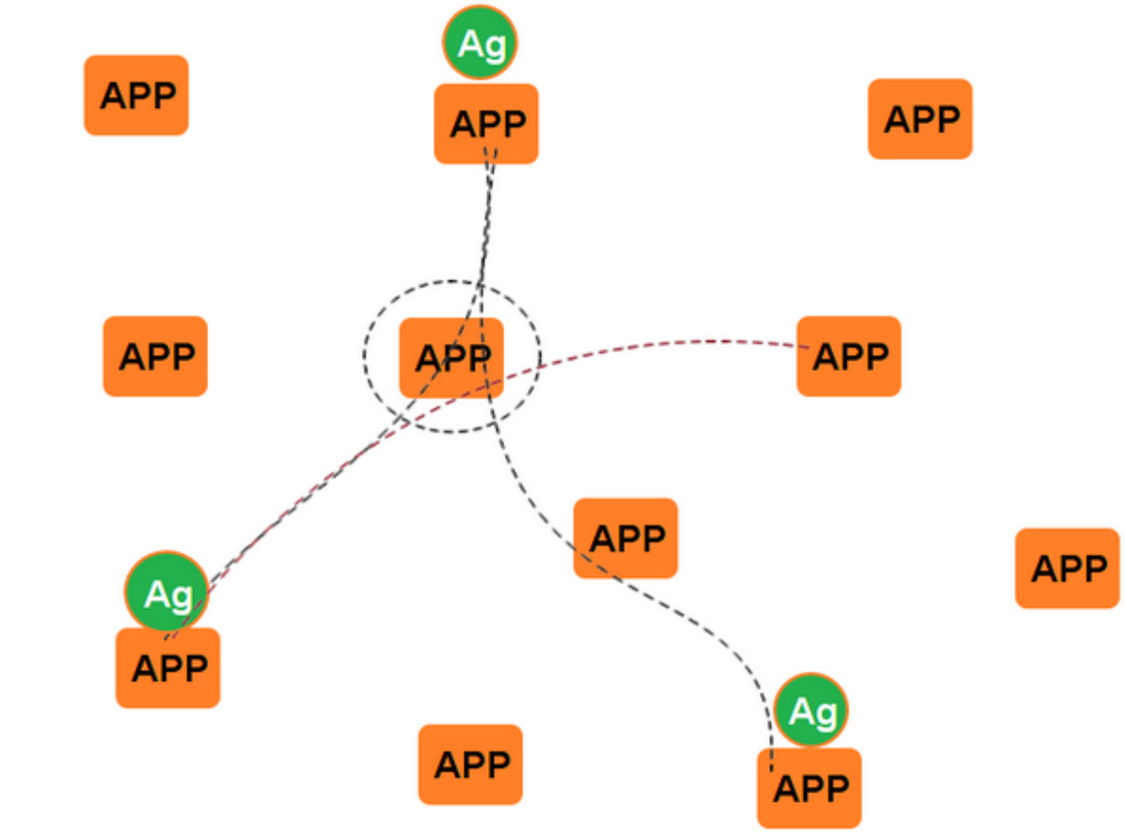


Abb. 26. Mobile Agenten in einem Netzwerk aus Plattformen (APP) und ihrer Spuren; Routing von Nachrichten über Kreuzungspunkte von Spuren

KOMMUNIKATIONSSIGNALE

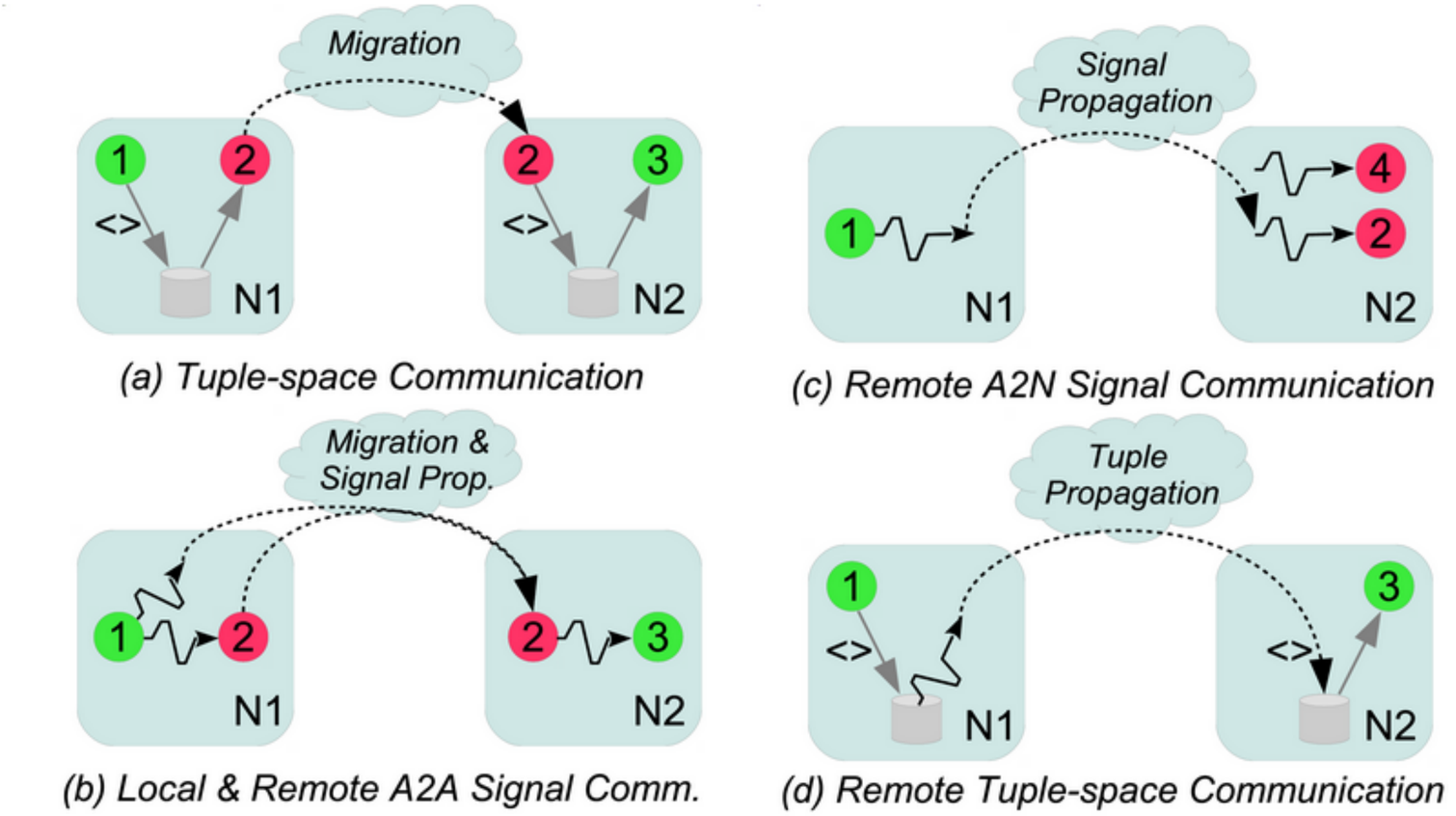


Abb. 27. Agentenkommunikation (a) Lokale Tupleräume (b) Agent-Agent Signale (c) Entfernte Agent-Knoten Signale (d) Entfernte Tupleraumoeprationen

HÖHERE KOMMUNIKATION: INTERAKTION

- Tupelräume und Signalnachrichten sind nur ein Werkzeug um höhere Ebenen der Kommunikation zu erreichen → Interaktion von Agenten.

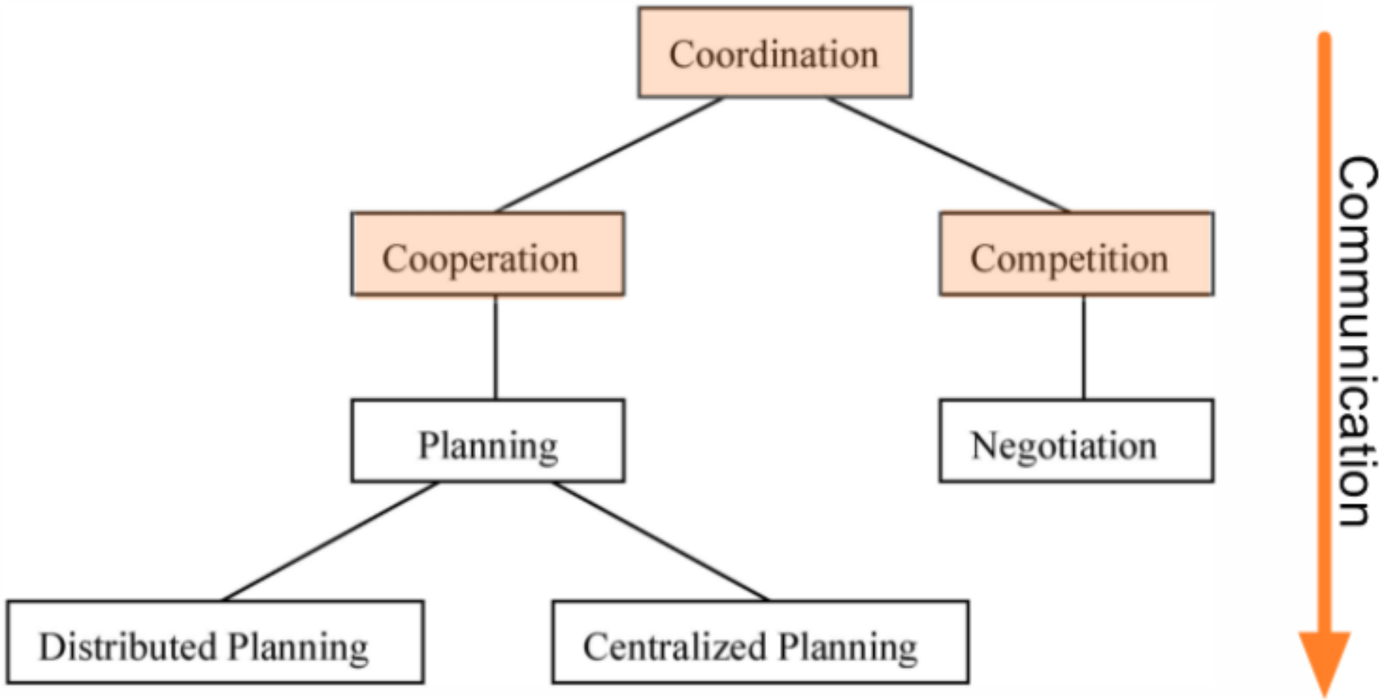


Abb. 28. Taxonomie der Agenteninteraktion die auf Kommunikation aufbaut

SPRACHE UND AKTIONEN

- Kommunikation als Sprache kann aktionsorientiert und wissensbasiert sein!
- Nachrichten verfolgen Absichten und sollen beim Empfänger Aktionen auslösen.

Speech Act Theory

Die Sprechakttheorie behandelt Kommunikation als Aktion. Es basiert auf der Annahme, dass Sprachhandlungen von Agenten genauso wie andere Aktionen ausgeführt werden, um ihre Absichten zu fördern.

» Das Ziel einer Anfrage/Ersuchens (Request) eines Sprechers ist die Ausführung einer Aktion des Zuhörers.

» Das Erreichen der Ziele über Sprachkommunikation hängt von Wissen und Planung des Sprechers und des Zuhörers ab.

» **Agentenkommunikationssprachen** spielen daher eine wesentliche Rolle bei der Planung und Aktionsausführung wissensbasierte Agenten (deduktive)

AGENTENKOMMUNIKATIONSSPRACHEN

- Wichtige Vertreter der sprachorientieren Kommunikation von Agenten sind:

KQML

KQML ist eine nachrichtenbasierte Sprache für die Agentenkommunikation. Daher definiert KQML ein allgemeines Format für Nachrichten. Eine KQML-Nachricht kann grob als Objekt betrachtet werden (im Sinne objektorientierter Programmierung): Jede Nachricht hat eine Performative (Zusammenhang zwischen Sprechen und Handeln, die man sich als die Klasse der Nachricht vorstellen kann) und eine Anzahl von Parametern (Attribut / Wert Paare, die man sich als Instanzvariablen vorstellen kann).

FIPA-ACL

Diese Sprache ähnelt oberflächlich KQML: Sie definiert eine "äußere" Sprache für Nachrichten, definiert 20 Performativen (wie z.B. inform), um die beabsichtigte Interpretation von Nachrichten zu definieren, und es ist keine spezifische Sprache für den Nachrichteninhalt vorgegeben. Darüber hinaus ähnelt die konkrete Syntax für FIPA-ACL-Nachrichten weitgehend der von KQML.

KQML

Performative	Meaning
`achieve`	S wants R to make something true of their environment
`advertise`	S claims to be suited to processing a performative
`ask-about`	S wants all relevant sentences in R's VKB
`ask-all`	S wants all of R's answers to a question C
`ask-if`	S wants to know whether the answer to C is in R's VKB
`ask-one`	S wants one of R's answers to question C
`break`	S wants R to break an established pipe
`broadcast`	S wants R to send a performative over all connections
`broker-all`	S wants R to collect all responses to a performative
`broker-one`	S wants R to get help in responding to a performative
`deny`	the embedded performative does not apply to S (anymore)
`delete-all`	S wants R to remove all sentences matching C from its VKB
`delete-one`	S wants R to remove one sentence matching C from its VKB
`discard`	S will not want R's remaining responses to a query
`eos`	end of a stream response to an earlier query
`error`	S considers R's earlier message to be malformed
`evaluate`	S wants R to evaluate (simplify) C
`forward`	S wants R to forward a message to another agent
`generator`	same as a `standby` of a `stream-all`
`insert`	S asks R to add content to its VKB

Abb. 29. KQML Nachrichtentypen (Performativen): Teil 1, VKB: Virtual Knowledge Base

monitor	S wants updates to R's response to a stream-all
next	S wants R's next response to a previously streamed performative
pipe	S wants R to route all further performatives to another agent
ready	S is ready to respond to R's previously mentioned performative
recommend-all	S wants all names of agents who can respond to C
recommend-one	S wants the name of an agent who can respond to a C
recruit-all	S wants R to get all suitable agents to respond to C
recruit-one	S wants R to get one suitable agent to respond to C
register	S can deliver performatives to some named agent
reply	communicates an expected reply
rest	S wants R's remaining responses to a previously named performative
sorry	S cannot provide a more informative reply
standby	S wants R to be ready to respond to a performative
stream-about	multiple response version of ask-about
stream-all	multiple response version of ask-all
subscribe	S wants updates to R's response to a performative
tell	S claims to R that C is in S's VKB
transport-address	S associates symbolic name with transport address
unregister	the deny of a register
untell	S claims to R that C is *not* in S's VKB

Abb. 30. KQML Nachrichtentypen (Performativen): Teil 2 [C]

KQML

- KQML Nachrichten besitzen Parameter, die u.A. eine **Sprache** des Inhaltes und eine **Ontologie** festlegen → gemeinsames Verständnis!

Parameter	Meaning
`:content`	content of the message
`:force`	whether the sender of the message will ever deny the content of the message
`:reply-with`	whether the sender expects a reply, and, if so, an identifier for the reply
`:in-reply-to`	reference to the `:reply-with` parameter
`:sender`	sender of the message
`:receiver`	intended recipient of the message

Abb. 31. KQML Nachrichtenparameter

- Eine **Ontologie** ist eine formale Definition eines Wissenskörpers. Die typischste Art von Ontologie beinhaltet strukturelle Komponenten. Im Wesentlichen eine Taxonomie der Klassen- und Unterklassenbeziehungen gekoppelt mit Definitionen der Relationen zwischen diesen Dingen.

Beispiele für Dialoge

```
(evaluate
 :sender A :receiver B
 :language KIF :ontology motors
 : reply-with q1 :content (val (torque m1)))
(reply
 :sender B :receiver A
 :language KIF :ontology motors
 :in-reply-to q1 :content (= (torque m1) (scalar 12 kgf)))

(stream-about
 :sender A :receiver B
 :language KIF :ontology motors
 :reply-with q1 :content m1)
(tell
 :sender B :receiver A
 :in-reply-to q1 :content (= (torque m1) (scalar 12 kgf)))
(tell
 :sender B :receiver A
 :in-reply-to q1 :content (= (status m1) normal))
(eos
 :sender B :receiver A :in-reply-to q1)
```

FIPA-ACL

Performative	Passing information	Requesting information	Negotiation	Performing actions	Error handling
accept-proposal			×		
agree				×	
cancel		×		×	
cfp			×		
confirm	×				
disconfirm	×				
failure					×
inform	×				
inform-if	×				
inform-ref	×				
not-understood					×
propagate				×	
propose			×		
proxy				×	
query-if		×			
query-ref		×			
refuse				×	
reject-proposal			×		
request				×	
request-when				×	
request-whenever				×	
subscribe		×			

Abb. 32. FIPA Nachrichtentypen (Performativen) und ihre Anwendungsgebiete [C]

FIPA-ACL

Implementierung

- Weit verbreitet ist die JADE Plattform in der Agenten als Java Threads ausgeführt werden.

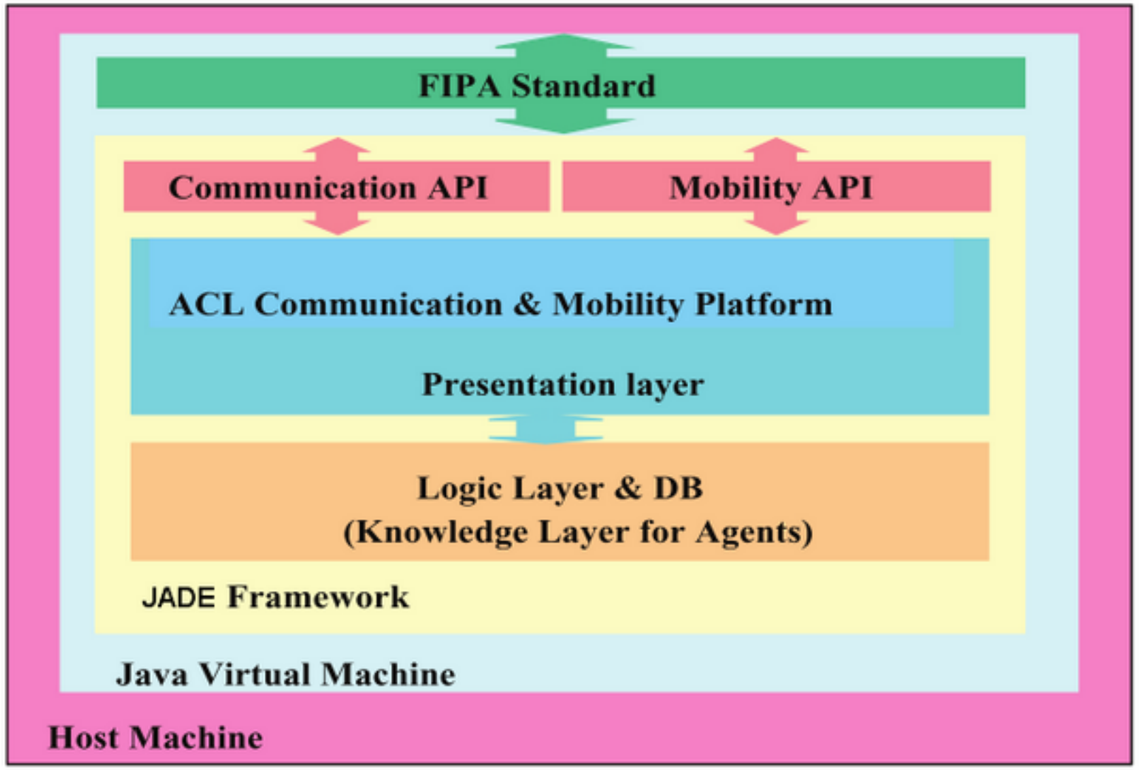


Abb. 33. Schichtenaufbau der APP mit FIPA-ACL über der Wissensdatenbank und Repräsentationsschicht [D]

ONTOLOGIEN

- Neben einer gemeinsamen Sprache bei der Kommunikation ist
  - Die richtige Interpretation der Inhalte, Informationen, und Wissensrepräsentation;
  - Und das Erkennen und Verstehen von Zusammenhängen (Strukturen)

  wichtig für ein zielgerichtete Emergenzverhalten von Agenten.
- Ontologien können die Informationen und Inhalte die ausgetauscht werden beschreiben und klassifizieren sowie Strukturen (Zusammenhänge) zwischen Elementen beschreiben.
- Auch Ontologien benötigen eine geeignete *Beschreibungssprache*, z.B. im XML/JSON Format, oder die OWL– The web ontology language
- Ontologien beschreiben u.A. bzw. bestehen aus:
  - **Klassen**
  - **Formale** "ist-ein" Taxonomien, d.h., die Einordnung von Klassen
  - **Attribute** von Klassen
  - **Wertebeschränkungen**
  - **Logische** Randbedingungen

ONTOLOGIEN

- Ontologien können sehr spezielle Beschreibungen von Elementen und Strukturen besitzen oder sehr allgemein sein.

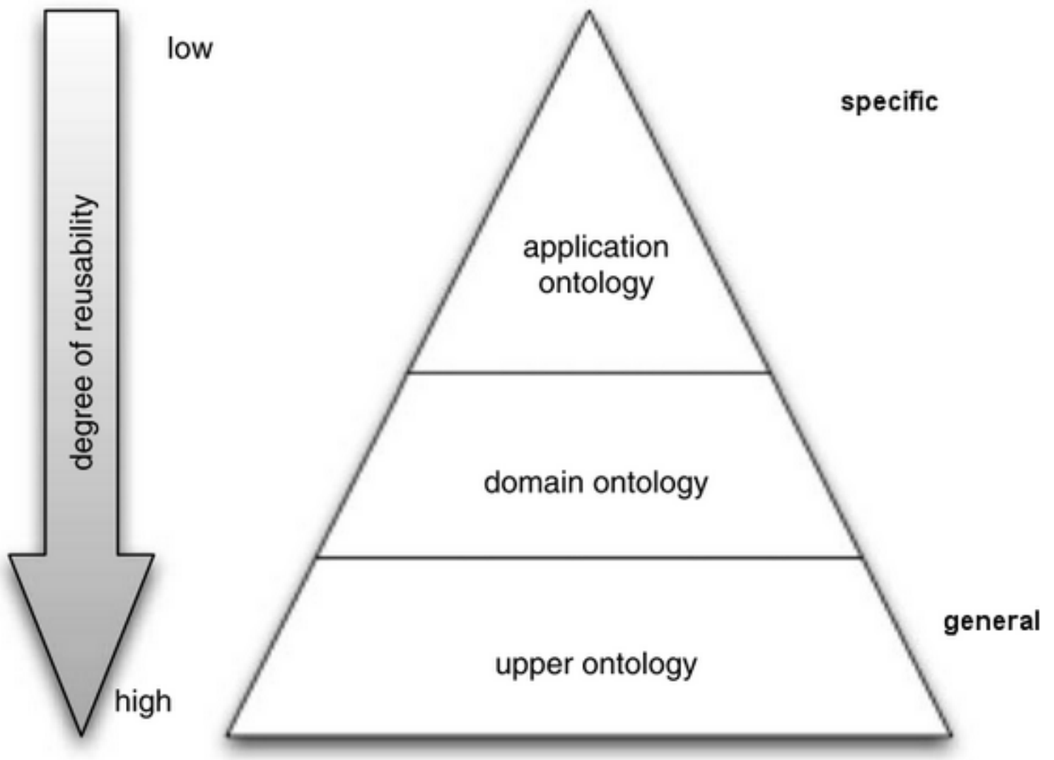


Abb. 34. Die Abstraktion von Ontologien bestimmt ihren Nutzen und eine Hierarchie [B]

ONTOLOGIEN

Applikationsontologie

Eine Anwendungsontologie definiert Konzepte, die von einer bestimmten Anwendung verwendet werden. Es wird typischerweise auf einer Domain-Ontologie und wiederum auf einer oberen Ontologie aufbauen. Konzepte aus einer Anwendungsontologie werden normalerweise nicht wiederverwendbar sein: Sie sind typischerweise nur in der Anwendung relevant, für die sie definiert wurden.

Domainontologie

Eine Domain-Ontologie definiert Konzepte, die für eine bestimmte Anwendungsdomäne geeignet sind.
Domain-Ontologien sind in der Regel auf Konzepten einer übergeordneten Ontologie aufbaut → Wiederverwendung von Ontologien ist sehr wichtig, da je mehr Anwendungen eine bestimmte Ontologie verwenden, desto mehr Übereinstimmung herrscht über Terme.

ONTOLOGIEN

- Ontologien besitzen unterschiedliche Ausdrucksstärke, je nachdem ob sie eher informal oder formal sind.

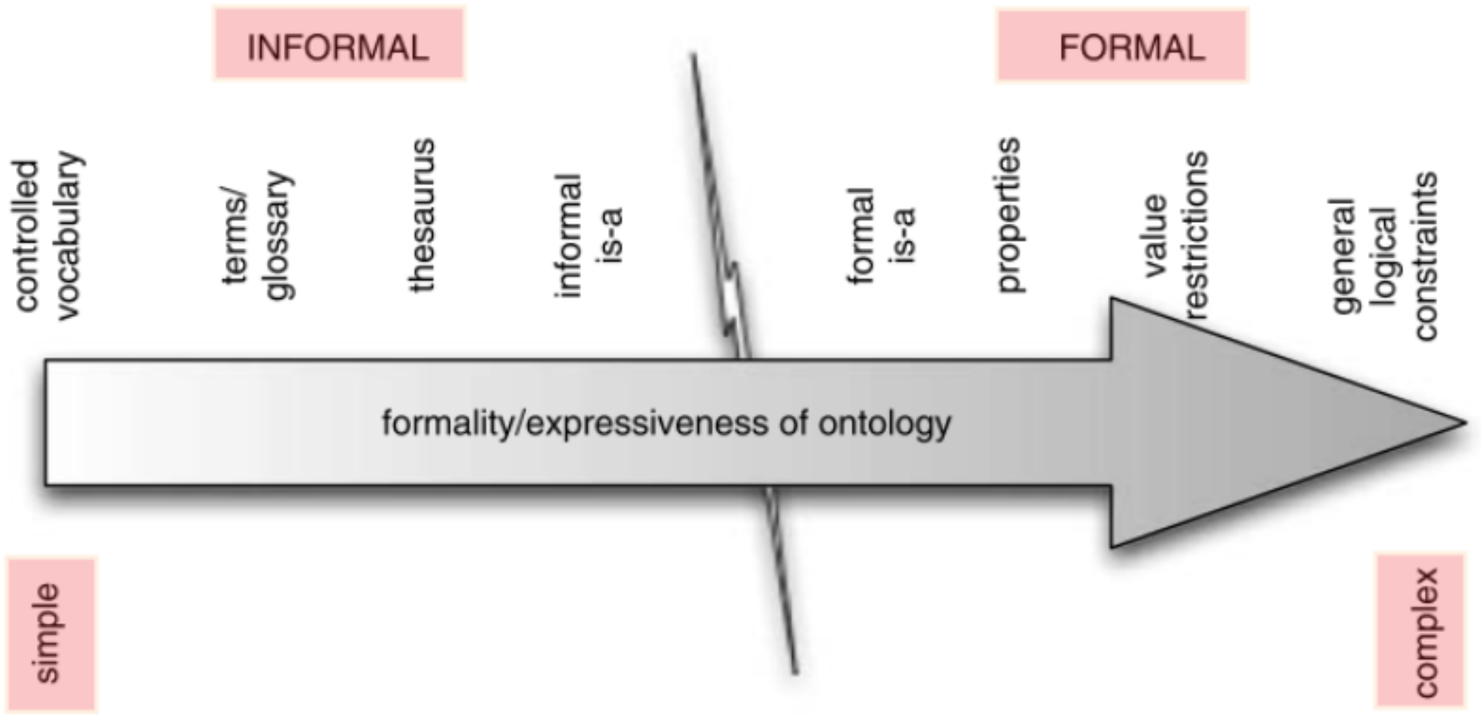


Abb. 35. Spektrum der Ontologien [B]

PROGRAMMIERMODELLE UND PROGRAMMIERSPRACHEN

MODELLIERUNG VON AGENTEN MIT PROGRAMMIERSPRACHEN

- Es gibt eine Vielzahl von prozeduralen und deklarativen Programmiersprachen um
  - Perzeption, Berechnung, Aktion, Planung und
  - Interaktion

zu beschreiben.

ATG MODELL

Zustandsbasierter Agent

- Besteht aus: (1) Körpervariablen (2) Kontrollzustand und Verhalten

Aktivität und Zustand

- Das Verhalten eines aktivitätsbasierten Agenten ist durch einen Agentenzustand gekennzeichnet, der durch Aktivitäten verändert wird.
- Aktivitäten verarbeiten Wahrnehmungen, planen Aktionen und führen Aktionen aus, die den Steuerungs- und Datenzustand des Agenten ändern.
- Aktivitäten und Übergänge zwischen Aktivitäten werden durch einen Aktivitätsübergangsgraphen (Activity Transition Graph, ATG) dargestellt.
- Die Übergänge starten Aktivitäten in der Regel abhängig von der Auswertung von Agentendaten (Körpervariablen), die den Datenzustand des Agenten repräsentieren.

ATG MODELL

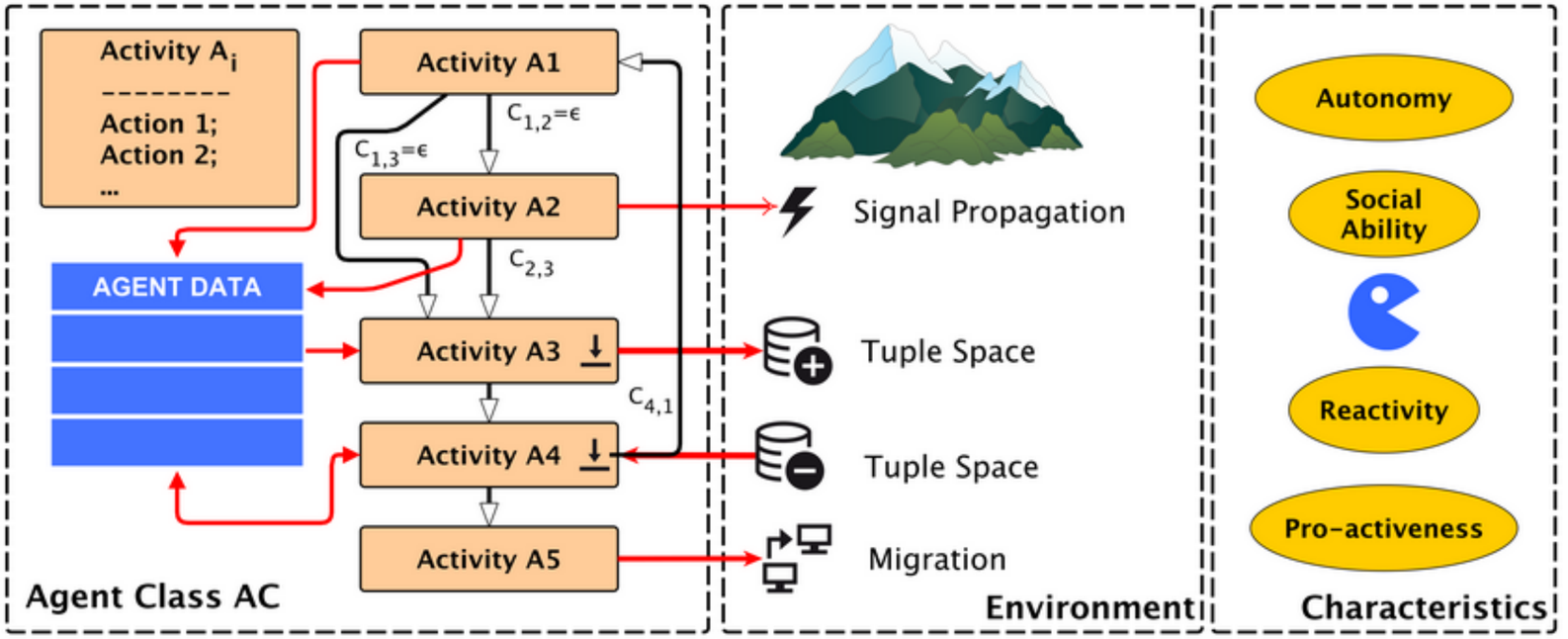


Abb. 36. (Links) Agentenverhalten, das von einem Aktivitätsübergangsgraphen vorgegeben ist, und die Interaktion mit der Umgebung, die durch Aktionen erfolgt, die in Aktivitäten ausgeführt werden (Rechts) Agentenmerkmale

ATG MODELL

- Ein Aktivitätsübergangsgraph, der mit Agentenklassen assoziiert ist, besteht aus einer Menge von Aktivitäten $\mathbb{A} = \{A_1, A_2, ..\}$ und einer Menge von Übergängen $\mathbb{T} = \{T_1\ (C_1), T_2\ (C_2), ..\}$, die die Kanten des gerichteten Graphen darstellen.
- Die Ausführung einer Aktivität, die selbst aus einer Folge von Aktionen und Berechnungen besteht, ist mit dem Erreichen eines Unterziels oder das Erfüllen einer Voraussetzung verknüpft, um ein bestimmtes Ziel zu erreichen, z. B. Sensordatenverarbeitung und Verteilung von Informationen.
- Normalerweise werden Agenten verwendet, um komplexe Aufgaben zu zerlegen basierend auf der Zerlegung durch MAS.
- Agenten können ihr Verhalten basierend auf Lern- und Umgebungsänderungen oder durch Ausführen einer bestimmten Unteraufgabe mit nur einer *Untermenge* des ursprünglichen Agentenverhaltens ändern → **Dynamische ATG**.

ATG MODELL

- Das ATG-Verhaltensmodell ist eng mit der Interaktion von Agenten mit deren Umgebung verbunden, hier hauptsächlich durch
  - Den Austausch von Daten unter Verwendung einer Tupelraum-Datenbank;
  - Durch Migration; und durch
  - Die Weitergabe von Nachrichten zwischen Agenten mittels Signalen.
  - Replikation und Instantiierung von Agenten

DATG MODELL

- Ein ATG beschreibt das vollständige Agentenverhalten.
- Jedes Unterdiagramm und jeder Teil des ATG kann einem Unterklassever-halten eines Agenten zugeordnet werden.
- Daher führt das Modifizieren der Menge von Aktivitäten $\mathbb{A}$ und Übergän-gen $\mathbb{T}$ des ursprünglichen ATG zu mehreren Unter- und Oberverhaltens-weisen, die Algorithmen implementieren, um verschiedene unterschiedli-che Ziele zu erfüllen.
- Die Rekonfiguration der Aktivitäten führt zu einer Menge von Aktivitäts-mengen $\mathbb{A}^* = \{\mathbb{A}_i \subset \mathbb{A}, \mathbb{A}_j \subset \mathbb{A}, \mathbb{A}_k \supset \mathbb{A}, ..\}$, die von der ursprünglichen Menge $\mathbb{A}$ abgeleitet sind, und die Modifikation oder Rekonfiguration von Über-gängen $\mathbb{T}^* = \{\mathbb{T}_1 \subset \mathbb{T}, \mathbb{T}_2 \subset \mathbb{T}, ..\}$ ermöglicht die dynamische ATG-Zusam-mensetzung (Komposition) und Agentenunterklassifizierung zur Laufzeit,

DATG MODELL

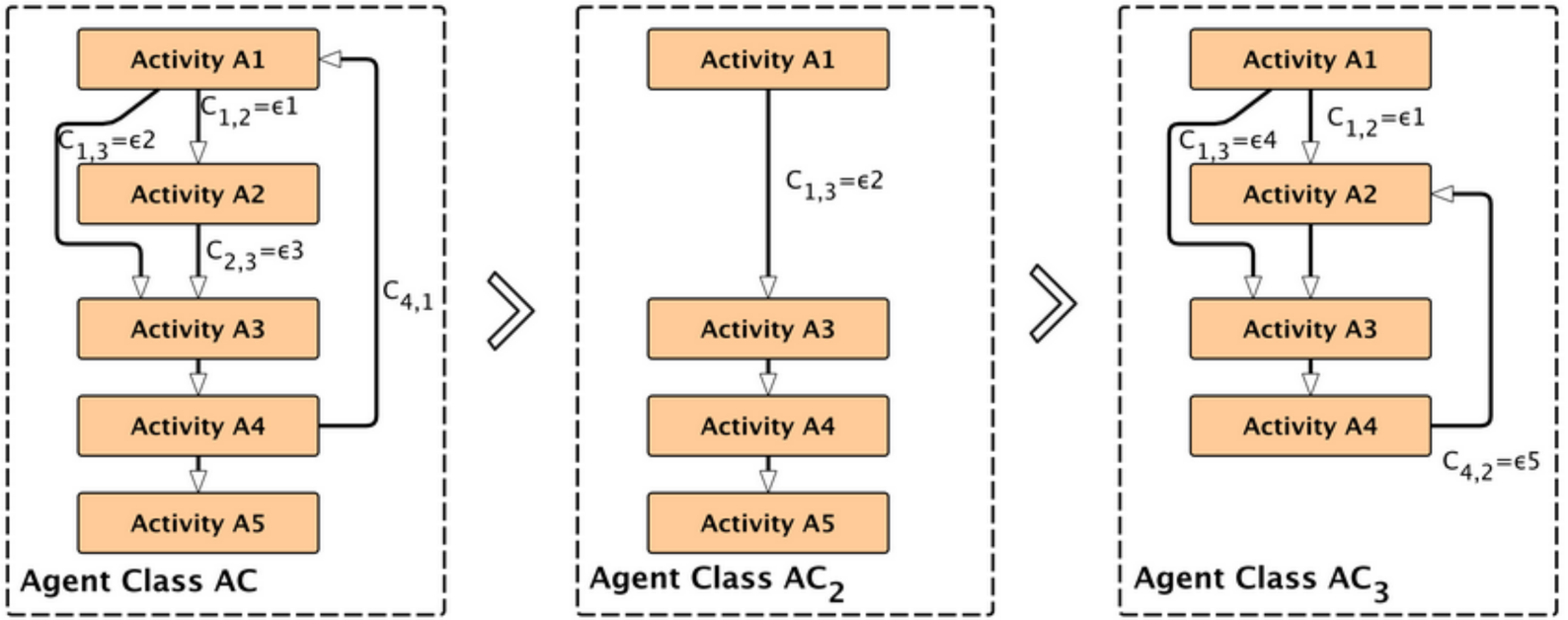


Abb. 37. Dynmaischer ATG: Transformation und Komposition

AGENTENKLASSEN

Eine Agentenklasse beschreibt ein bestimmtes Verhalten eines Agenten mittels eines ATG und einer Menge von Körpervariablen.

- Von einer Klasse können zur Laufzeit Agenten instantiiert (erzeugt) werden.

Verhalten

Eine bestimmte Agentenklasse AC_i bezieht sich auf das zuvor eingeführte ATG Modell, dass das Laufzeitverhalten und die von Agenten ausgeführten Aktion definiert.

Wahrnehmung

Ein Agent interagiert mit seiner Umgebung, indem er eine Datenübertragung unter Verwendung eines einheitlichen Tupelraums mit einer koordinierten datenbankähnlichen Schnittstelle durchführt.
Daten aus der Umgebung beeinflussen das folgende Verhalten und die Aktion eines Agenten. Daten, die an die Umgebung (z.B. die Datenbank) weitergegeben werden, beeinflussen das Verhalten anderer Agenten.

AGENTENKLASSEN

Speicher

Zustandsbasierte Agenten führen Berechnungen durch, indem sie Daten ändern. Da Agenten als autonome Datenverarbeitungseinheiten betrachtet werden können, werden sie hauptsächlich private Daten modifizieren, und ein Berechnungsergebnis, dass diese Daten verwendet, in die Umgebung übertragen.

Daher enthält jeder Agent und jede Agentenklasse eine Menge von Körpervariablen $\mathbb{V} = \{v_1{:}typ_1, v_2{:}\ typ_2, ..\}$, die durch Aktionen in Aktivitäten modifiziert und in Aktivitäten und Übergangsausdrücken gelesen werden.

Parameter

Agenten können zur Laufzeit von einer bestimmten Agentenklasse instantiiert werden, die Agenten mit gleichen anfänglichen Steuerungs- und Datenzuständen erstellt.

Um einzelne Agenten zu unterscheiden (Individuen zu erzeugen), wird eine externe sichtbare Parametermenge $\mathbb{P} = \{p_1{:}typ_1, p_2{:}\ typ_2, ..)$ hinzugefügt, die die Erstellung verschiedener Agenten bezüglich des Datenzustands ermöglicht. Innerhalb einer Agentenklasse werden Parameter wie Variablen behandelt.

AGENTENKLASSEN

Eine Agentenklasse AC_i ist daher zunächst definiert durch das folgende Mengentupel:

$$AC_i = \langle \mathbb{A}_i, \mathbb{T}_i, \mathbb{V}_i, \mathbb{P}_i \rangle$$
$$\mathbb{A} = \{a_1, a_2, .., a_n\}$$
$$\mathbb{T} = \{t_{ij} = t_{ij}(a_i, a_j, cond) | a_i \xrightarrow{cond} a_j; i, j \in \{1, 2, .., n\}\}$$
$$a_i = \{i_1, i_2, ..| i_u \in ST\}$$
$$\mathbb{V} = \{v_1, v_2, .., v_m\}$$
$$\mathbb{P} = \{p_1, p_2, .., p_i\}$$

AGENTENKLASSEN

Multiagentensysteme

Definition 10.
Es gibt ein Multiagentensystem (MAS), das aus einer Reihe einzelner Agenten besteht (ag_1, ag_2, ..). Es gibt verschiedene Verhaltensweisen für Agenten, die als Klassen **AC** = {AC_1, AC_2, ..} bezeichnet werden. Ein Agent gehört zu einer dieser Klassen.

Jede Agentenklasse wird dann durch das erweiterte Tupel $AC = <\mathbb{A}, \mathbb{T}, \mathbb{F}, \mathbb{S}, \mathbb{H}, \mathbb{V}, \mathbb{P}>$ angegeben.

- $\mathbb{A}$ ist der Satz von Aktivitäten (Graphenknoten), $\mathbb{T}$ ist der Satz von Übergängen, die Aktivitäten (Beziehungen, Graphenkanten) verbinden,
- $\mathbb{F}$ ist der Satz von Rechenfunktionen,
- $\mathbb{S}$ ist der Satz von Signalen, $\mathbb{H}$ ist der Satz von Signalhandlern,
- $\mathbb{V}$ ist die Menge der Körpervariablen und $\mathbb{P}$ die Menge der Parameter, die von der Agentenklasse verwendet werden.

AGENTENKLASSEN

Definition 11.
In einer spezifischen Situation ist ein Agent ag_i an einen Netzwerkknoten $N_{m,n,o,..}$ (z.B. Mikrochip, Computer, virtueller Simulationsknoten) an einem eindeutigen räumlichen Ort (m, n, o, ..) gebunden und wird dort verarbeitet.

Es gibt eine Menge verschiedener Knoten $\mathbf{N} = \{N_1, N_2, ..\}$, die z.B. in einem maschenartigen Netzwerk mit einer Nachbarverbindung (z.B. vier in einem zweidimensionalen Gitter) angeordnet sind. Die Knotenverbindung kann dynamisch sein und sich im Laufe der Zeit ändern. Die Knotennachbarn sind unterscheidbar.

Jeder Knoten ist in der Lage, eine Anzahl von Agenten $n_i(AC_i)$ zu verarbeiten, die zu einer Agentenverhaltensklasse AC_i gehören, und mindestens eine Teilmenge von $\mathbf{AC}' \subset \mathbf{AC}$ zu unterstützen.

Ein Agent (oder zumindest sein Zustand) kann zu einem Nachbarknoten migrieren wo er weiter ausgeführt wird.

AAPL

AAPL: Activity-based Agent Programming Language

- Das AAPL-Programmiermodell sollte optimal den Anforderungen von MAS entsprechen, die in unzuverlässigen Sensor- und generischen verteilten Netzwerken mit low-resource Plattformen eingesetzt werden,
- Auf der einen Seite sollte AAPL die Kernkonzepte von Agenten widerspiegeln, auf der anderen Seite sollte AAPL Kernkonzepte der traditionellen Programmiersprache bereitstellen, um die Programmierung von weit verbreiteten Algorithmen zu erleichtern.

AAPL

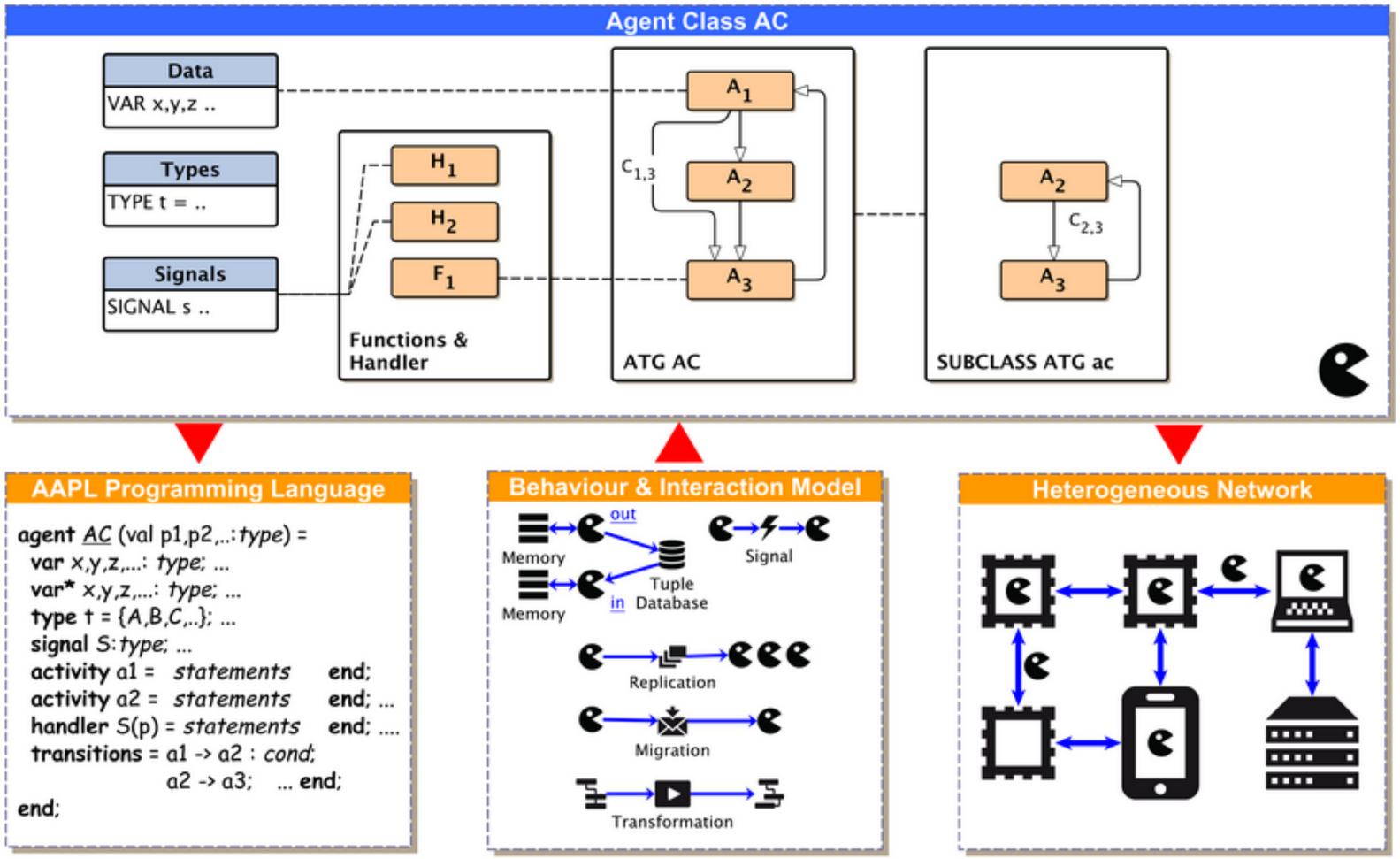


Abb. 38. Programmierebene des Agentenverhaltens mit Aktivitäten und Übergängen (AAPL, links); Agentenklassenmodell und Aktivitätsübergangsgraphen (oben); Agenteninstantiierung, -verarbeitung und -interaktion auf der Netzwerkebene (rechts)

AAPL

Agentenklassen

- Das AAPL-Agentenverhaltensmodell ist eng mit dem DATG-Modell verwandt.
- Eine Agentenverhaltensbeschreibung ist in einer Agentenklasse gekapselt und besteht zunächst aus einer Menge von Aktivitäten $\mathbb{A} = \{a_1, a_2, ..\}$, die Aktionen ausführen (Berechnung und Interaktion mit der Umgebung) und einer Menge von Übergängen $\mathbb{T} = \{t_1, t_2, ..\}$ zwischen den Aktivitäten.
- Ein Agent ag_i befindet sich immer in einem der Aktivitäten $a_i \rightarrow$ Kontrollzustand
- Die Übergänge definieren die Aktivierung von Aktivitäten basierend auf der bereits ausgeführten vorherigen Aktivität und dem internen Datenzustand
- Eine Agentenklasse hat *eine Menge von Aktivitäten*, kann aber *verschiedene Mengen von Übergängen* besitzen die zur Laufzeit ausgewählt und umgeschaltet werden können.

AAPL

Definition und Aufbau einer Agentenklasse

Definition of an Agent Class

```
agent AC [[(p1:DT,p2:DT,..)]] =
begin
  definitions
    body variables   var  v1:T; ..
    body variables*  var* v1:T; ..
    [[ signals ]]        signal s:T; ..
    [[ exceptions ]]     exception e;
    [[ types ]]          type T =  ..;
  activities       activity a = .. end;
  transitions      transitions = .. end;
  [[ functions ]]  function f(..) = .. end;
  [[ handler ]]    handler s(..) = .. end;
  [[ subclasses ]] subclass sc = .. end;
 end;
```

Short Notation

```
Ψ AC:(p1,p2,..) →
{

  Σ : { .. }
  σ : { .. }
  ξ : { .. }
  ε : { .. }
  κ: { .. }
  α a: { .. }
  Π : { .. }
  f : (p1,..) → { .. }
  ξ s:(p) → { .. }
  φ ac: { .. }
}
```

Creation of Agents

```
id := new AC(x1,x2,..);
id := fork (x1,x2,..);
id := fork sc(x1,x2,..);
id := new AC;
.. ATG composition ..
run(id,x1,x2,..);
```

```
id ← Θ+ AC(x1,x2,..)
id ← Θ→(x1,x2,..)
id ← Θ→ sc(x1,x2,..)
```

AAPL

Activity Definition

```
activity ac_i =
  definitions     var x:DT; var* l:DT;
  statement;
  statement;
  ..
end;
```

Short Notation

```
α ac_i { ..
  Σ: {x_1,..} σ:{l_1,..}
  statement;
  statement;
  ..
}
```

Transitions Definition

```
transitions =
  a_i → a_j ⟦ : cond ⟧ ;
  ..
end;
```

```
Π {
  a_i → a_j ⟦ : cond ⟧
  ..
}
```

Subclass Definition

```
subclass sc =
  definitions     var x:DT; var* l:DT;
  .. imports      use x;
  activities      activity a_j = .. end;
  .. imports      use a_j;
  transitions     transitions = .. end;
end;
transitions sc = .. end;
```

```
φ sc: { .. }

↓ x

↓ a_j
π { .. }
```

Verteilte Netzwerke und Mobilität

- In einer bestimmten Situation befindet sich ein Agent ag_i auf einem Netzwerkknoten $N_{m,n,o,..}$ (z.B. Mikrochip, Computer, virtueller Simulationsknoten) an einer einzigartigen räumlichen Stelle (m, n, o, ..).
- Es gibt eine Menge verschiedener Knoten $\mathbf{N} = \{nd_1, nd_2, ..\}$ die in einem verteiltes Netzwerk mit Peer-to-Peer-Nachbarschaftskonnektivität (z.B. zweidimensionales Gitter) oder Peer-to-N Konnektivität (Internet) angeordnet sind.
- Jeder Netzwerkknoten N_i stellte eine Agentenverarbeitungsplattform $AP\text{-}P_i$ zur Verfügung
- Jede Agentenplattform kann eine Anzahl von Agenten unabhängig voneinander ausführen.
- Die Agenten werden entweder lokal von der Plattform, anderen Agenten, oder durch Empfang von Prozessschnappschüssen durch die Plattform erzeugt.

AAPL

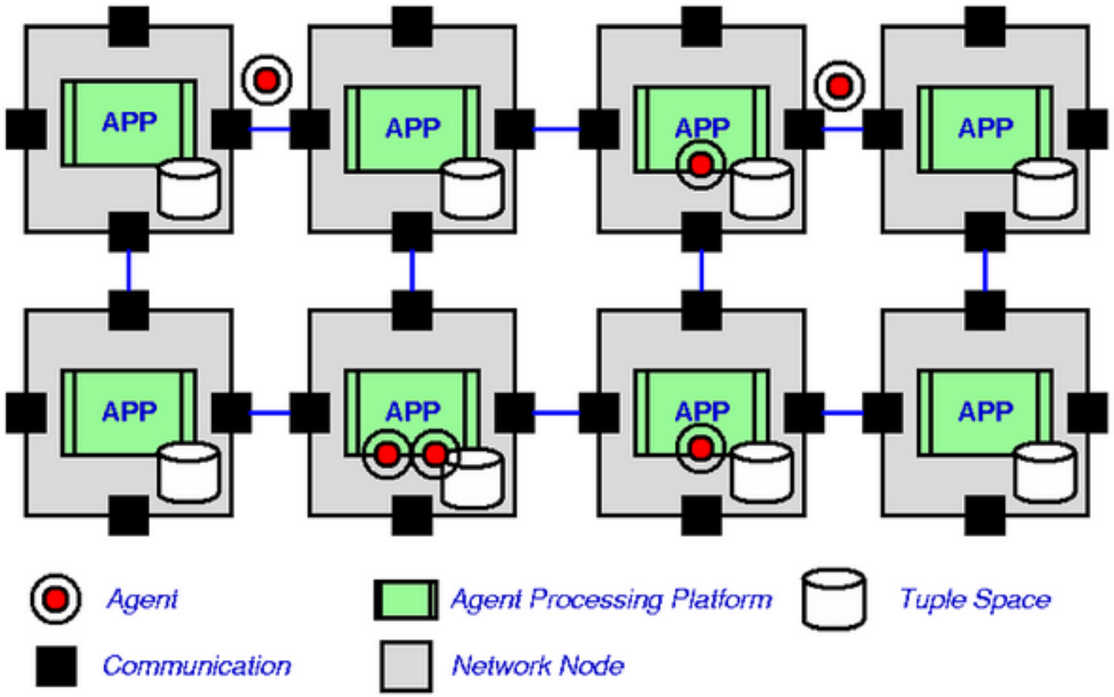


Abb. 39. Maschennetzwerk aus Netzwerkknoten mit APP und Agenten

AAPL

- Agenten können von einer Plattform APP_a zu einer anderen APP_b unter der Angabe der Verbindungsrichtung (NORTH, WEST, ..) migrieren.

moveto(*DIR*)

Der Agent wird zu der durch die Richtung *DIR* angegebenen Plattform transferiert und dort weiter ausgeführt. Die Migration erfolgt durch einen Prozessschnappschuss der den gesamten Daten und Kontrollzustand sowie den ATG (Code) beinhaltet. Die Übertragung kann fehlschlagen (z.B. keine oder gestörte Verbindung)

link?(*DIR*) → Boolean

Prüft eine APP Verbindung in die angegebene Richtung.

DIR

Die Menge aller möglichen Richtungen: *DIR* = { *NORTH*, *SOUTH*, *WEST*, *EAST*, *UP*, *DOWN*, *NE*, *NW*, *SE*, *SW*, .. }

AAPL

Instantiierung von Agenten

- Parametrisierbare neue Agenten einer bestimmten Klasse *AC* können zur Laufzeit von Agenten erzeugt (instantiiert) werden
- Es gibt verschiedene Methoden, um zur Laufzeit neue Agenten zu erstellen:

1. Agenten können von einer ursprünglichen Root-Agentenklasse instantiiert werden und durch Parameter individualisiert werden.
   - » Wenn in der Agentenklassendefinition mehrere Übergangssätze vorhanden sind, kann für den neu erstellten Agenten ein bestimmter Satz ausgewählt und aktiviert werden.
2. Agenten können aus einer *Unterklasse* einer ursprünglichen Root-Agentenklasse instantiiert werden.
3. Agenten können von einem (übergeordneten) Agenten abgespalten werden. Die untergeordneten Agenten (Kindagenten) erben das Agentenverhalten, einschließlich der aktuellen Übergänge und der Daten (Inhalt von Körpervariablen).

- » Mögliche Plattformeinschränkng: Ein Kindagent muss jedoch denselben Übergangssatz des Elternagenten verwenden. Ein Übergangssatzwechsel ist nicht möglich.

4. Agenten werden aus einem ursprünglichen oder bereits modifizierten Agentenverhalten durch Unter- oder Überklassifizierung neu zusammengesetzt und erzeugt → **Freie Komposition mit Aktivitäten und Übergängen.**
   - » Obwohl Unterklassen aus einer Teilmenge von Aktivitäten und Übergängen erstellt werden, ist eine freie Komposition der ATG zur Laufzeit möglich, kann jedoch durch die zugrunde liegende Verarbeitungsplattform eingeschränkt werden.

AAPL

Operationale Semantik: Instantiierung

new *AC* [.*SC*] (v1,v2,..) → *identifier*

Erzeugt einen neuen Agenten der *AC* Klasse mit Parameterwerten v1, v2, usw. Die Funktion gibt einen (auf Knotenebene) eindeutigen Agenten-identifizierer (Name) zurück.

fork [.*SC*] (v1,v2,..) → *identifier*

Ein Agent kann mehrere lebende Kopien von sich selbst erstellen. Die Kindagenten gehören entweder der gleichen Klasse *AC* oder einer Unterklasse *SC* (mit anderen/kleineren ATG) an. Einzelne Körpervariablen bzw. Parameter können verändert vererbt werden.

kill(*identifier*)

Agenten können mittels der *kill* Anweisung zerstört werden (erzwungene Terminierung). Ohne Angabe eines Agentenidentifiziers führt diese Anweisung zur Terminierung des aufrufenden Agenten.

AAPL

Operationale Semantik: Interaktion

Signal Definition

```
signal S1,S2,.. [[ : DT ]];
```

Signal Raising

```
send(ID,Si,[[arg]]);                          ID={$parent,$self,agent-id}
reply(Si,[[arg]]);
broadcast(AC,DX,DY,S,[[arg]]);
sendto(TO,Si,[[arg]]);                        TO={DIR,PATH,node-id}
```

Signal Handler Definition

```
handler Si [([[val]] p)]] = .. end;
```

Timer Installation and Signal Handler

```
signal Ti [[ : DT ]];
timer+(timeout,T);
timer+(timeout,T,V);
timer-(T);
handler Ti [([[val]] p)]] = .. end;
```

Abb. 40. AAPL Signale

AAPL

Generation of Tuples and Markings	Short Notation
out(v_1,v_2,..);	$\nabla^{+}(v_1,v_2,..)$
mark(*timeout*,v_1,v_2,..);	$\nabla^{T}(v_1,v_2,..)$
Search and Removal of Tuples	
in(v_1,x_1?,..);	$\nabla^{-}(v_1,x_1?,..)$
in(v_1,x_1?,?,?,..);	$\nabla^{-}(v_1,x_1?,?,?,..)$
stat := try_in(*timeout*,..,);	$\nabla^{?-}(tmo,..)$
Search and Reading of Tuples	
rd(v_1,x_1?,..,v_2,..);	$\nabla^{\%}(v_1,x_1?,..)$
rd(v_1,x_1?,?,?,..,v_2,..);	$\nabla^{\%}(v_1,x_1?,?,?,..,v_2,..)$
stat := try_rd(*timeout*,..);	$\nabla^{?\%}(tmo,..)$
Testing of Tuple Existence	
exist?(v_1,?,?,..);	$\nabla^{?}(v_1,?,?,..)$
Removal of Tuples	
rm(v_1,?,?,v_2,..);	$\nabla^{\times}(v_1,?,?,..)$
Generation of Active Tuples	
eval(id,v_1,?,?,v_2,..);	$\nabla^{+-}(id,v_1,?,?,..)$
Distributed Tuple Space/Remote Tuple Access	
store(*to*,v_1,v_2,..);	$\nabla^{+}(v_1,v_2,..) \rightarrow to$
collect(*to*,v_1,x_1?,..);	$\nabla^{-}(v_1,x_1?,..) \rightarrow to$
copyto(*to*,v_1,x_1?,..,v_2,..);	$\nabla^{\%}(v_1,x_1?,..) \rightarrow to$
Multi-Pattern Selector	
res:=alt((v_1,x_1?,..)|(..));	$\nabla^{*-}((v_1,x_1?,..)\|(..))$
res:=try_alt(*tmo*,(v_1,x_1?,..)|(..));	$\nabla^{?*-}(tmo,(v_1,x_1?,..)\|(..))$

Abb. 41. AAPL Tupelraum Operationen

AAPL

Operationale Semantik: Verhaltensänderung und Rekonfiguration

Transitions	Short Notation
transition+ ⟦C⟧ (⟦id,⟧ a1,a2,c);	π^+ ⟦C⟧ (⟦id,⟧ a1,a2,c)
transition* ⟦C⟧ (⟦id,⟧ a1,a2,c);	π^* ⟦C⟧ (⟦id,⟧ a1,a2,c)
transition- ⟦C⟧ (⟦id,⟧ a1,a2);	π^- ⟦C⟧ (⟦id,⟧ a1,a2,c)
Activities	
activity+ ⟦C⟧(⟦id,⟧ a1,a2,...);	α^+ ⟦C⟧(⟦id,⟧ a1,a2,...)
activity- ⟦C⟧(⟦id,⟧ a1,a2,...);	α^- ⟦C⟧(⟦id,⟧ a1,a2,...)

Abb. 42. AAPL Verhaltensänderung und Rekonfiguration durch Veränderung der Aktivitäten- und Übgergangsmenge

AAPL

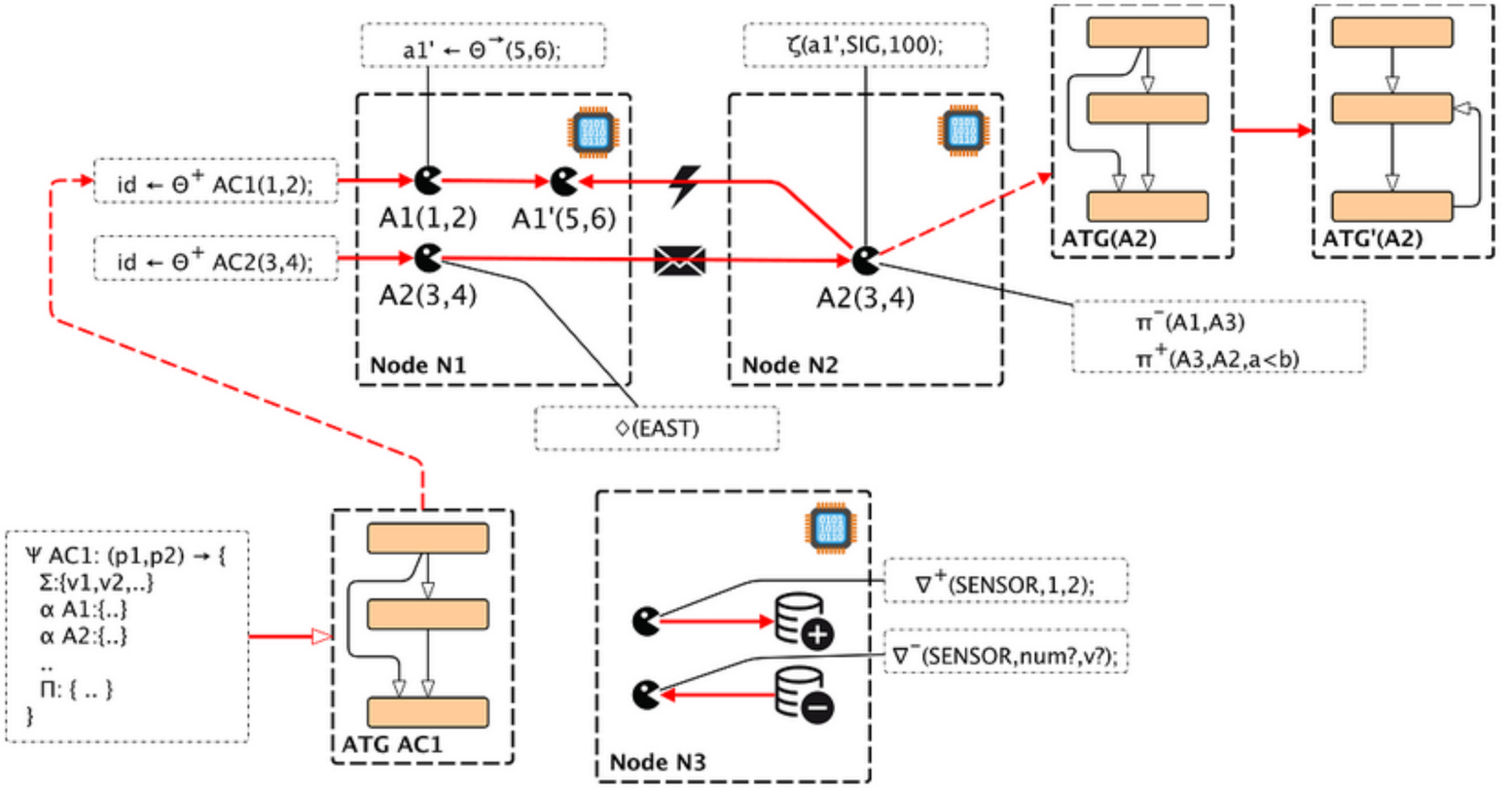
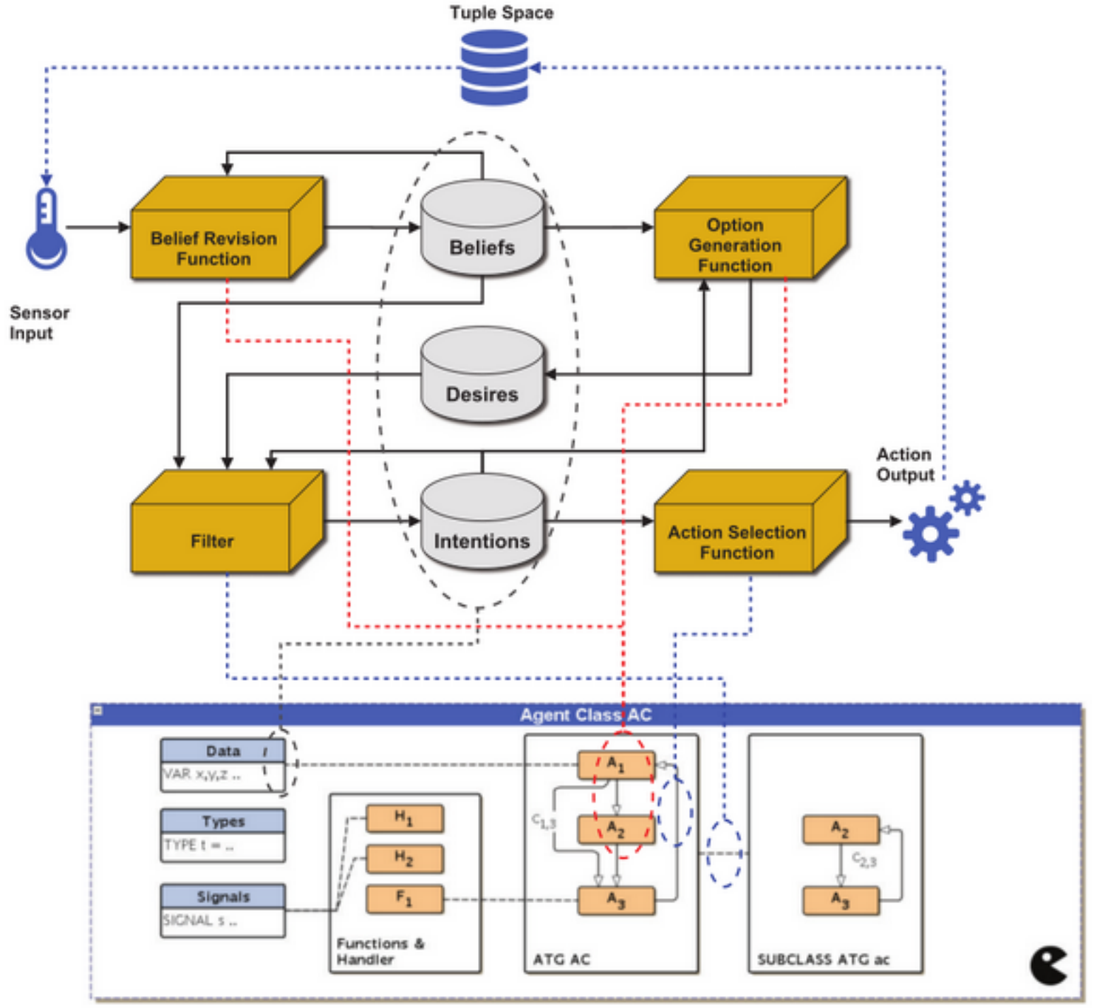


Abb. 43. Effekt von verschiedenen AAPL Anweisungen zur Laufzeit auf Agenten und Plattformen

AAPL

Zusammenhang vom AAPL/DATG Verhaltensmodell mit BDI Architektur



AAPL KURZNOTATION

Tuple Space

∇^{+}(TP) *Add tuple TP to database*

∇^{-}(TP) $\nabla^{?-}$(TMO,TP) *Read and remove tuple TP from database (or try it ?)*

$\nabla^{\%}$(TP) $\nabla^{?\%}$(TP) *Read tuple (or try) TP from database*

$\nabla^{\times}$(TP) *Remove tuple TP from database*

$\nabla^{\top}$(TMO,TP) *Add marking tuple TP to database with lifetime TMO*

?∇(TP) *Test for existence of tuple TP*

x? *Formal Parameter*

Mobility

?Λ(Δ) ?Λ(δ) *Test for connection link in direction Δ*

δ *Set of directions {NORTH,SOUTH,..}*

Δ *Relative position vector and hop vector relative to root node*

∂(δ) *Numeric direction-to-position difference vector*

ϖ *Opposite direction*

⇔(δ) ⇔(Δ) *Migrate to direction δ / Δ*

Agents

ψ AC:(x,y,..)→{..} *Define an agent class AC with parameters x,y,..*

φ ac:{..} *Define a sub-class ac*

Σ: {..} *Definition of body variables,σ:{..} none-persistent*

α *A: {..} Definition of activity A*

F:(x,y,..)→{..} Definition of a function F with parameters x,y,..

ζ *S:(P)→{..} Definition of a signal handler S with parameter P*

Π:{..} Definition of transitions, π:{..} : Define sub-class transitions

$\theta^{\rightarrow}$(v1,v2,..) *Fork agent with arguments*

θ^{X}(A) *Destroy agent A*

θ^{*}AC(v1,v2,..) *Create new agent from agent class AC with arguments*

Timer

τ^{+}(TMO,S) *Add timer with timout TMO and signal S*

τ^{-}(S) *Remove timer for signal S*

Reconfiguration

$\pi^{+}(T_1,T_2,C)$ *Add transition $T_1 \rightarrow T_2$ with condition C*

$\pi^{*}(T_1,T_2,C)$ *Update transitions $T_1 \rightarrow T_2$ with condition C*

$\pi^{-}(T_1,T_2)$ *Remove transitions $T_1 \rightarrow T_2$*

α^{+}(A) *Add activity A*

α^{-}(A) *Remove activity A*

Signals

ζ S(V) ⇒ A *Send a signal S with argument V to agent A*

Values

$V *Agent reference variable (V=self, parent,..)*

ℜ{SET} *Random element selection from a set or in the range {min .. max}*

x←∈ *Change value of variable x*

Symbols

Timer
Reconfiguration
Replication
Move
Tuple Database
Signal
Kill
Blocking Process
Static Transition
Dynamic Transition

Set Iteration

∀{ x∈X | c(x) } do I

∀{ x| c(x) } do I

Repeat the following statement *I(x)* (using x) for each element x of the set X for which the condition *c(x)* is satisfied.

Interval set Iteration

∀ x ∈ { a .. b} do I

∀{ x| x ∈ { a .. b }} do I

Repeat the following statement *I(x)* (using x) for each element x of the interval set {a .. b}

JAVASCRIPT :: DATEN UND VARIABLEN

- Variablen werden mit dem Schlüsselwort `var` definiert → Erzeugung eines Datencontainers!
- Es gibt keine Typendeklaration in JS! Kerntypen:

  T_{core}={`number, boolean, object, array, string, function`}
- Alle Variablen sind **polymorph** und können alle Werttypen aufnehmen.
- Bei der Variabledefinition kann ein Ausdruckswert zugewiesen werden

```
var v = ε,..;   v = ε;
```

JavaScript

CLEAR HELP var assign

JAVASCRIPT :: FUNKTIONEN

- Funktionen können mit einem Namen oder anonym definiert werden
- Funktionen sind Werte 1. Ordnung → Funktionen können Variablen oder Funktionsargumenten zugewiesen werden
- Eine Funktion kann einen Wert mit der `return` Anweisung zurückgeben. Ohne explizite Wertrückgabe → `undefined`
- Es wird nur Call-by-value Aufruf unterstützt - jedoch werden Objekte, Funktionen und Arrays als Referenz übergeben; Parameter p_i sind an Funktionsblock gebunden

```
function name (p1,p2,..) { statements ; return ε } name(ε1,ε2,..)
```

JavaScript

CLEAR HELP fun

JAVASCRIPT :: FUNKTIONEN

- Da in JavaScript Funktionen Werte erster Ordnung sind können
  - » Funktionen an Funktionen übergeben werden und
  - » Funktionen neue Funktionen zurückgeben (als Ergebnis mit `return`)
- Es können daher **anonyme** Funktionen `function (..) {..}` definiert werden die entweder einer Variablen als Wert oder als Funktionsargument übergeben werden.

```
var x = function (p_i) { ε(p_i) }
array.forEach(function(elem,index) { ε(p_i) }
```

JAVASCRIPT :: DATENSTRUKTUREN

In JavaScript sind Objekte universelle Datenstrukturen (sowohl Datenstrukturen als auch Objekte) die mit Hashtabellen implementiert werden. Arrays werden in JavaScript ebenfalls als Hashtabelle implementiert!. D.h. Objekte == Datenstrukturen == Arrays == Hashtabellen.

- Es gibt *kein* nutzerdefinierbares Typensystem in JavaScript.
- Eine Datenstruktur kann jederzeit definiert und verändert werden (d.h. Attribute hinzugefügt werden)

```
var dataobject = {
  a:ε,
  b:ε, ..
  f:function () { .. }
}
..
dataobject.c = ε
```

JAVASCRIPT :: DATENSTRUKTUREN

- Dadurch dass Objekte und Arrays mit Hashtabellen implementiert (d.h. Elemente werden durch eine Textzeichenkette referenziert) werden gibt es verschiedene Möglichkeiten auf Datenstrukturen und Objektattribute zuzugreifen:

```
dataobject.attribute
dataobject["attribute"]
array[index]
array["attribute"]
```

JAVASCRIPT :: OBJEKTE

- Objekte zeichnen sich in der objektorientierten Programmierung durch Methoden aus mit der ein Zugriff auf die privaten Daten (Variablen) eines Objekts möglich wird.
- In JavaScript kann auf Variablen eines Objekts (die Attribute) immer direkt zugegriffen werden.
- Attribute können Funktionen sein - jedoch können die Funktionen nicht wie Methoden direkt auf die Daten des Objektes zugreifen.
- Daher definiert man Methoden über Prototypenerweiterung in JavaScript.
- Die Methoden können über die `this` Variable direkt auf das zugehörige Objekt zugreifen (also auch auf die Variablen/Attribute)
- Es gibt eine Konstruktionsfunktion für solche Objekte mit Prototypendefinition der Methoden
- Objekte werden mit dem `new` Operator und der Konstruktionsfunktion erzeugt.

JAVASCRIPT :: OBJEKTE

```
function constructor (p_i) {
  this.x=ε
  ..
}
constructor.prototype.method_i = function (..) {
  this.x=ε;
  ..
}
...

var obj = new constructor(..);
```

JavaScript

CLEAR HELP obj

AGENTJS

- AgentJS ist die JavaScript Implementierung von AAPL
  - » Die meisten AAPL Operationen und Anweisungen sind in AgentJS verfügbar
  - » Aber: JavaScript unterstützt nicht das Konzept der Prozessblockierung
  - » Daher anderes Scheduling und Ablaufmodell mit Scheduling Blocks

Agentenklasse

- Agentenklassen werden in JS über Konstrukorfunktionen definiert.
- Eine Agentenklasse definiert:
  - » Körpervariablen (nur mobile und werterhaltende)
  - » Aktivitäten (als Funktionsobjekt)
  - » Übergangsbedingungen (als Funktionsobjekt)
  - » Optionale Signalhandler (als Funktionsobjekt)
  - » Ein `next` Attribute initialisiert mit der Startaktivität

AGENTJS

- Aber: Das `this` Objekt als Refernz auf eine Agenteninstanz ist auch in geschachtelten Funktionen gültig, d.h., in allen
  - Aktivitätsfunktionen,
  - Übergangsfunktionen,
  - Signalhandlerfunktionen, und in
  - Callback Funktionen erster Ordnung von eingebauten Funktionen (z.B. `iter(list,function () { this is agent! }`
- Ein Agentenprozess wird in einem Sandkasten gekapselt ausgeführt. Daher:
  - Ein Agent darf **nur** auf Körpervariablen zugreifen und keine freien Variablen oder lokale Variablen verwenden (d.h. welche die außerhalb des Konstruktors definiert wurden oder welche die innerhalb des Konstruktorkörper definiert wurden):

```
function ac (p) {
  var x;       // Wrong!!!
  this.y = p; // Correct!!!
  this.act = { ax: function () { var a; // is correct
}
```

Definition 12.

```
function ac(p1,p2,..) {
  // Body Variables
  this.x=ε; ..
  this.act = {      // Activities
   a1: function () { .. },
   a2: function () { .. },
   ..
   an: function () { .. }
  }
  this.trans = {    // Transitions
   a1: function () { return cond?ai:aj },
   a2: aj,
   ..
  }
  this.on = {       // Signal Handler
    error:    function (err) { .. },
    signal1: function (arg) { .. }
  }
  this.next = a1;
}
```

AGENTJS

Agenteninstantiierung und Terminierung

```
function create(class:string,arguments:{},level?:number) → id:string
function fork(arguments?:{},level?:number) → id:string
function kill(id:string|undefined)
```

- Bei der *create* Operation die einen neuen Agenten der Klasse *ac* instantiiert werden die Klassenparameter der Instanz mit konkreten Werten mit dem Parameterargument `arguments:{p1:v1,p2:v2,..}` initialisiert.
- Bei der *fork* Operation die eine Kopie des aufrufenden Agenten erzeugt werden Klassenattribute (die `this.x` Variablen, nicht die Parameter {p}!) mit neuen Werten überschrieben, d.h. `arguments:{x:v1,y:v2,..}`
- Die *kill* Operation ohne Argument führt zur Terminierung des aufrufenden Agenten (== `kill(me())`)

AGENTJS

Beispiele

```
id = create('explorer',{dir:DIR.NORTH,radius:1});
child = fork({x:10,y:20});
kill(child);
kill(me());
```

Prozessblockierung

- Eine Aktivität wird in einem Durchlauf ausgeführt und kann nicht blockieren, wie dies z.B. bei den Tupeloperationen der Fall sein könnte.
  - » Daher darf sich in *AgentJS* maximal nur **ein** blockierende Operation (die tatsächlich dann die Aktivität blockiert, und nicht den Programmfluss) **am Ende** einer Aktivität befinden.
  - » Bei einer Sequenz von blockierenden Operationen muss in der Aktivität ein Scheduling Block verwendet werden (Mikroaktivitäten).
  - » Jede Mikroaktivität darf eine blockierende Operation enthalten!

AGENTJS

- Es gibt drei verschiedene Blockkonstruktoren:
  - B: Ein sequenzieller Scheduling Block
  - L: Ein iterativer Schleifenblock
  - I: Ein Iteratorblock für Arrays und Objekte (Strukturen)

Definition 13.

```
B([                    L([
   function () { .. },     function init () {..},
   function () { .. },     function cond () {..},
   function () { .. },     function next () {..},
  ..                        [ function () { .. },
])                            function () { .. }, ..
                             ]])

I(obj:[]|{},
  function next (elem:*) { },
  [
   function () {..},
   function () {..}, ..
  ],
   function finalize () { .. }
)
```

AGENTJS

Tupeloperationen

```
function alt (pattern [],callback:function,all?:boolean,tmo?:number)
function collect (to:path,pattern) → number
function copyto (to:path,pattern) → number
function evaluate (pattern,callback:function (tuple)) → tuple
function exists (pattern) → boolean
function inp (pattern,callback:function(tuple|tuple[]|none),all?:boolean,tmo?:number)
function listen (pattern,callback:function (pattern) → tuple)
function out (tuple)
function mark (tuple,tmo:number)
function rd (pattern,callback:function(tuple|tuple[]|none) ,all?:boolean,tmo?:number)
function rm (pattern,all?:boolean)
function store (to:path,tuple) → number
function ts (pattern,callback:function(tuple) → tuple)
function try_alt try_inp try_rd (tmo:number,..)
```

» *collect*, *copyto*, und *store* sind verteilte Tupelraumoperationen und wirken auf entfernten Tupelräumen

AGENTJS

Beispiele

```
out(['MARKING1',1]);
out(['SENSORA',100,true]);
inp(['SENSORA',_,_],function (tuple) {
  if (tuple) this.s =tuple[1];
});
rm(['SENSORA',_,_],true);
try_rd(0,function (tuple) { .. });
ts(['MARKING',_],function (t) { t[1]++ });
alt([
  ['SENSORA',_,_],
  ['SESNORB',_],
  ['EVENT'],
],function (tuple) {
  if (tuple && tuple[0]=='EVENT') {..}
  else ..
});
```

AGENTJS

Signale und Handler

```
function send (to:aid,sig:string|number,arg?:*)
fucntion broadcast (class:string,range,@sig,@arg?)
function sendto (to:dir,sig:string|number,arg?:*)
function sleep (milli:number)
function timer.add (milli:number,sig:string,arg:*,repeat:boolean) → string
function timer.delete (sig:string)
this.on = { SIGNAL : function (arg,from) { .. } }
```

Typen und Muster

```
type aid = string
type range = hops:number|region:{dx:number,dy:number,..}
enum DIR = {NORTH , SOUTH , WEST , EAST , ..
            PATH {tag='DIR.PATH',path:string} ,
            IP {tag='DIR.IP',ip:string} ,
            CAP {cap:string}
} : dir
```

AGENTJS

- Signalhandler werden außerhalb von Aktivitäten ausgeführt.

Code Muster

```
this.child=none;
this.act = {
  a1: function () {
    this.child=fork({child:none});
    timer.add(500,'QUERY',true);
  }
  a2: function () {
   // Raising of signal
   if (this.child) send(this.child,'PARENT',me());
  }
}
// Installation of Signal Handler
this.on : {
   PARENT : function (arg) {
     log('Got signal from my parent '+arg);
   },
   QUERY: function () { .. }
}
```

Mobilität von Agenten

```
enum DIR = {NORTH , SOUTH , WEST , EAST , ..
            PATH {tag='DIR.PATH',path:string} ,
            IP {tag='DIR.IP',ip:string} ,
            CAP {cap:string}
} : dir
function moveto (to:dir)
function opposite (dir) → dir
function link (dir) → boolean|string|[]
```

- Für die IP Richtung existiert eine Typkonstruktionsfunktion `DIR.IP(ip:string|number)` mit der Angabe einer IP Adresse und einer IP Portnummer im Format "IP:PORT", oder bei Verbindungen von JAM Knoten auf den gleichem Hostrechner nur die IP Portnummer.
- Die *opposite* Funktion liefert die entgegengesetzte Richtung, z.B. NORTH -> SOUTH. Bei IP Ports i.A. nicht definiert oder bei bereits migrierten Agenten die IP Adresse des letzten Knotens.

AGENTJS

- Die *link* Funktion testet ob ein Port mit einer anderen Seite verbunden ist (nicht zuverlässig). Bei Multicast IP Ports (Standard) gibt die Funktion alle derzeitig angebundenen anderen Knotenadressen zurück (Aufruf mit Argument `DIR.IP('*')`).
- Nach einer Migration wird die nächste folgende Aktivität (definiert durch die Übergangsbedingungen) ausgeführt. Daher muss die *moveto* Anweisung die letzte in einer Aktivität oder eingebettet in einen Scheduling Block sein!

JAM SHELL

- Die JAM shell *jamsh* integriert eine JAM Plattform.
- Beim Start wird ein JAM Knoten (d.h. einer virtuelle/logische JAM APP) erzeugt.
- Nach dem Start können einzelne Anweisungen ausgeführt werden, wie das Laden von Agentenklassen oder das Herstellen von Verbindungen zu anderen JAM Knoten (physisch oder logisch)

```
# node jamsh
JAM Shell. Version 1.1.14 (C) Dr. Stefan Bosse
[JAM] Created world MUJIVOTO.
[JAM] Created root node mujivoto (0,0).
> help
> port(DIR.IP(10001))
iprouter: add link localhost:10001
[AMP C2:EF:38:1F:F5:EC IP(10001)] Starting 127.0.0.1:10001 [MUL] (proto udp)
[AMP C2:EF:38:1F:F5:EC IP(10001)] IP port 141.26.71.28:10001 (proto udp)
> open('code/hello.js')
> start()
> create('hello',{text:"hello world"})
dicexiro
> stats('node')
```

JAM SHELL

- Skripte (mit JAM Shell Kommandos und eingebetteten Agentenkonstruktorfunktionen) können mittels des `script(file)` Kommandos geladen werden.

Beispiel

```
function hello(msg) {
  this.msg=msg;
  this.act = {
    init: function () {
      log('hello '+this.msg);
    },
    end: function () {}
  }
  this.trans = {
    init:end
  }
  this.next=init;
}
compile(hello);
```

JAM SHELL

Initialisierung

- Die eigentliche JAM APP muss (einmalig) mit dem Kommando `start()` gestartet werden um Agenten auszuführen:

```
> start()
[JAM] Starting JAM loop ..
```

- Schon vor dem Start können Agenten erzeugt werden!
- Mit dem `stop()` Kommando wird die *Ausführung von Agenten* durch JAM gestoppt, jedoch bleiben die Agenten erhalten. Eine nochmalige Verwendung des `start()` Kommandos führt die Ausführung der Agenten weiter.

Netzwerke

- JAM Knoten können z.B. über IP Netzwerke miteinander verbunden werden. Dazu muss
  - » ein Kommunikationsport auf jeder JAM APP erzeugt werden, und
  - » die Ports miteinander verbunden werden. Achtung: Es existiert kein IP Verbindungsaufbau, die Kopplung ist locker!

JAM SHELL

- Mit dem `port(dir)` Kommando kann ein neuer Kommunikationsport für eine JAM APP erzeugt werden, i.A, `port(DIR.IP("IP:IPPORT"))`
- Mit dem `link(dir)` Kommando werden zwei Ports miteinander verbunden, i.A. `port(DIR.IP("IP2:IPPORT2"))`.

```
APP-A > port(DIR.IP('IP1:IPPORT1'))
iprouter: add link localhost:10001
[AMP 63:35:E4:66:FA:43 IP(localhost:10001)] Starting 127.0.0.1:10001 [MUL] (proto udp)
[AMP 63:35:E4:66:FA:43 IP(localhost:10001)] IP port 141.26.71.164:10001 (proto udp)
APP-B > port(DIR.IP('IP2:IPPORT2'))
iprouter: add link localhost:10002
[AMP CE:EE:47:39:61:1A IP(localhost:10002)] Starting 127.0.0.1:10002 [MUL] (proto udp)
[AMP CE:EE:47:39:61:1A IP(localhost:10002)] IP port 141.26.71.164:10002 (proto udp)
APP-A > link(DIR.IP('IP2:IPPORT2'))
[AMP 63:35:E4:66:FA:43 IP(localhost:10001)] Trying link to 127.0.0.1:10002
iprouter: add route 141.26.71.164:10001 -> 127.0.0.1:10002#negodoqe
[AMP 63:35:E4:66:FA:43 IP(localhost:10001)]
  Linked with ad-hoc udp 127.0.0.1:10002, AMP CE:EE:47:39:61:1A, Node negodoqe
APP-B >
iprouter: add route 141.26.71.164:10002 -> 127.0.0.1:10001#quxibumi
[AMP CE:EE:47:39:61:1A IP(localhost:10002)]
  Linked with ad-hoc udp 127.0.0.1:10001, AMP 63:35:E4:66:FA:43, Node quxibumi
```

PLATTFORMEN

ÜBERBLICK

- Plattformen müssen geeignet sein für:
  - » Ausführung von mobilen Agenten
  - » Anbindung an bestehende Software
  - » Einsatz in mobilen und eingebetteten Geräten → Ressourceneffizienz!
  - » Interneteinsatz → WEB Browser!
  - » Skalierbarkeit (> 1000 Agenten)
  - » Simulation!

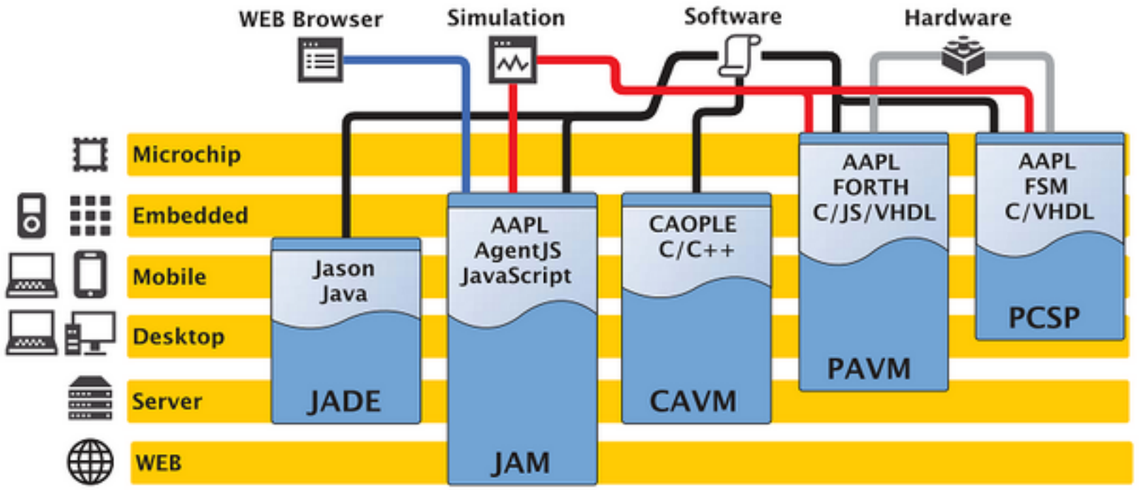


Abb. 44. Vergleich verschiedener Plattformen und ihre Einsatzgebiete

JADE

- JADE implementiert Agenten in Java

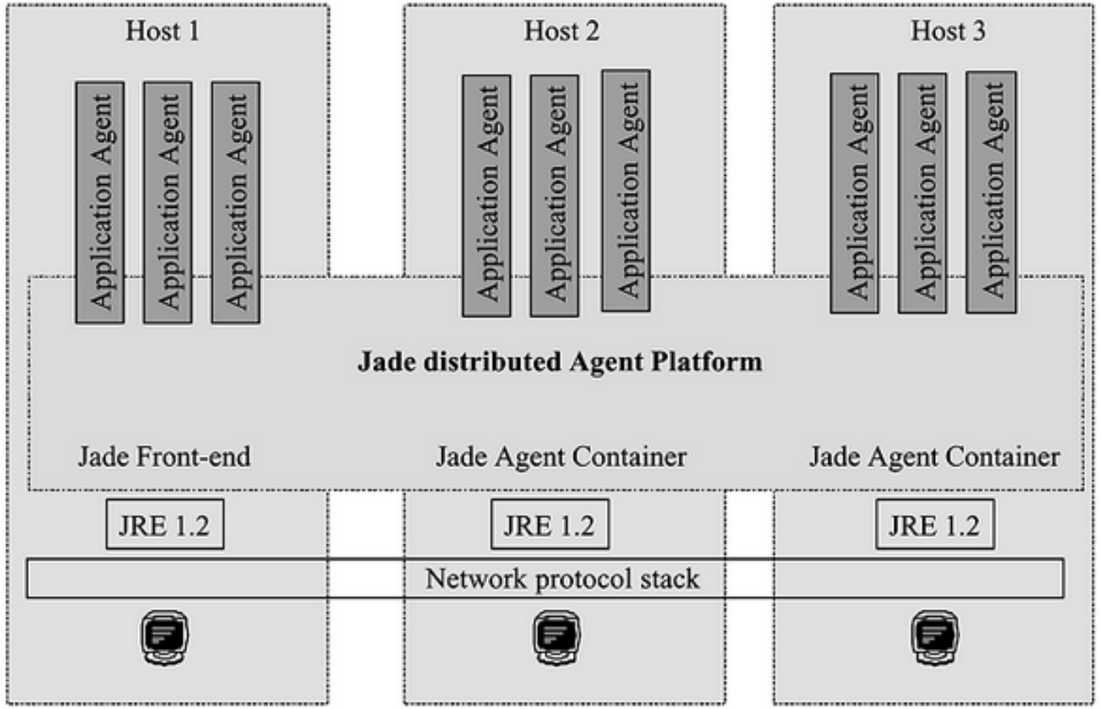


Abb. 45. Softwarearchitektur einer Agentenplattform [4]

JADE

- JADE baut auf FIPA ACL auf und die Kommunikationsarchitektur ist zentraler Bestandteil
- Kommunikation und Agentenmanagement sind eng gekoppelt → Agent Management System & Agent Communication Channel
- Organisationsservices werden benötigt (wo ist wer und wer ist wer?) → Directory Facilitator

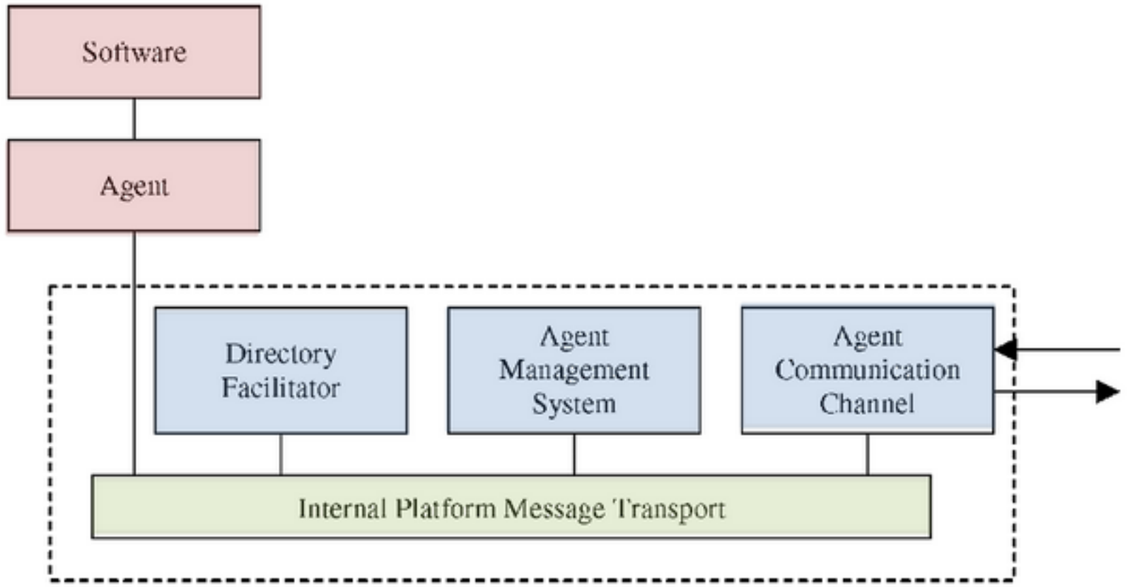


Abb. 46. Allgemeine FIPA-ACL Sofwareplattform [4]

JADE

- Agenten werden durch eine Menge von Verhalten beschrieben die vom Verhaltensscheduler ausgeführt werden
- JADE hat ein Multithreading Ausführungsmodell für grobgranulierte Parallelität
- Ein Thread führt alle Verhalten von einem Agenten aus (Überlappung nicht möglich; nur sequenzielles Scheduling)

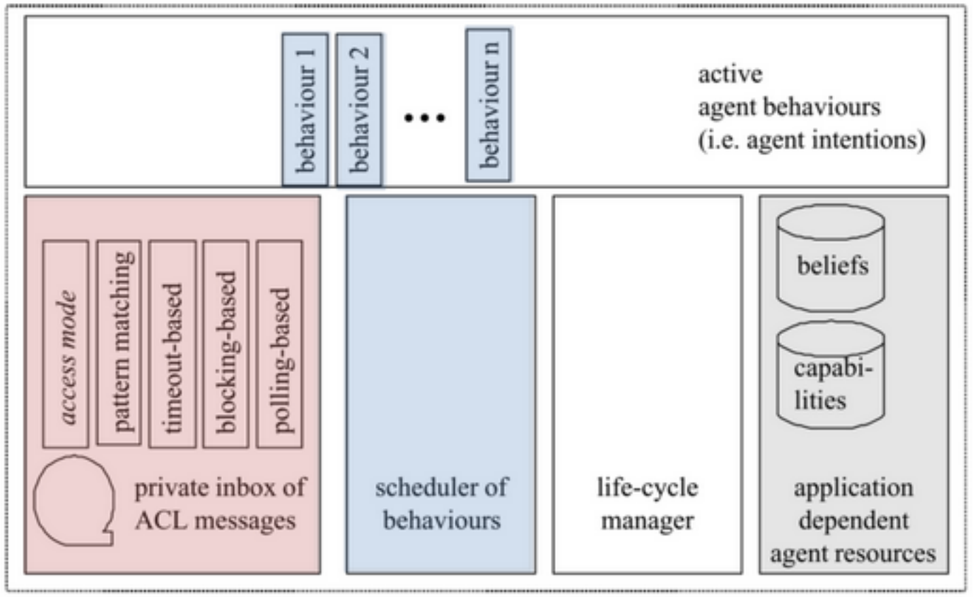


Abb. 47. JADE Agentenarchitektur [4]

JADE

Mikroscheduling

- Das Aufführungsmodell von JADE ist nicht preemptiv.
- D.h. ein Verhalten und dessen Aktionen werden als ganzes (atomar) ausgeführt, was zu großen Latenzen und unfairen Agentenscheduling führen kann
- Daher wird ein Mikroscheduling im Agenten benötigt:

```
public class my3StepBehaviour {
  private int state = 1;
  private boolean finished = false;
  public void action() {
    switch (state) {
     case 1: { op1(); state++; break; }
     case 2: { op2(); state++; break; }
     case 3: { op3(); state = 1; finished = true; break; }
    }
  } } }
```

JAM

- JAM besteht im wesentlichen aus:
  - Einem Agentenscheduler,
  - einem Sandbox Environment für die isolierte Ausführung von Agenten, und
  - dem Agent Input-Output System (AIOS).
- JAM ist vollständig in JavaScript implementiert und wird daher von einer JS VM ausgeführt:
  - Googles V8 mit node.js/jxcore
  - Spidermonkey im WebBrowser oder mit jxcore
  - Samsungs JerryScript mit IoT.js
  - usw.
- AgentJS Code kann direkt ausgeführt werden (nur Sandboxing erforderlich)!
- Eine physische JAM kann eine Vielzahl von virtuellen JAM Knoten ausführen (nur sequenzielles Scheduling)

JAM

AIOS

- Das AIOS kapselt eine Vielzahl von Modulen und stellt eine API zur Verfügung:
  - TUPLE: Tupleraum Datenbank
  - SIGNAL: Signalpropagation zwischen Agenten und JAM Knoten
  - CODE: AgentJS Text ⇔ Code Transformation, Code Morphing, und Sandboxing
  - NODE: Virtueller JAM Knoten
  - WORLD: Bindung von virtuellen JAM Knoten in einem physischen Knoten (Virtualisierung)
  - PROC: Agentenprozesses (Ausführungscontainer für Agenten)
  - MOBI: Agentenmobilität
  - COMM: JAM Kommunikation
  - WATCHDOG: Faires Agentenscheduling durch Laufzeitüberwachung (Time slicing)
  - ML: Machine Learning → Fokus mobile Algorithmen und Modelle
  - SAT: SAT Logic Solver → Knowledge Base

JAM

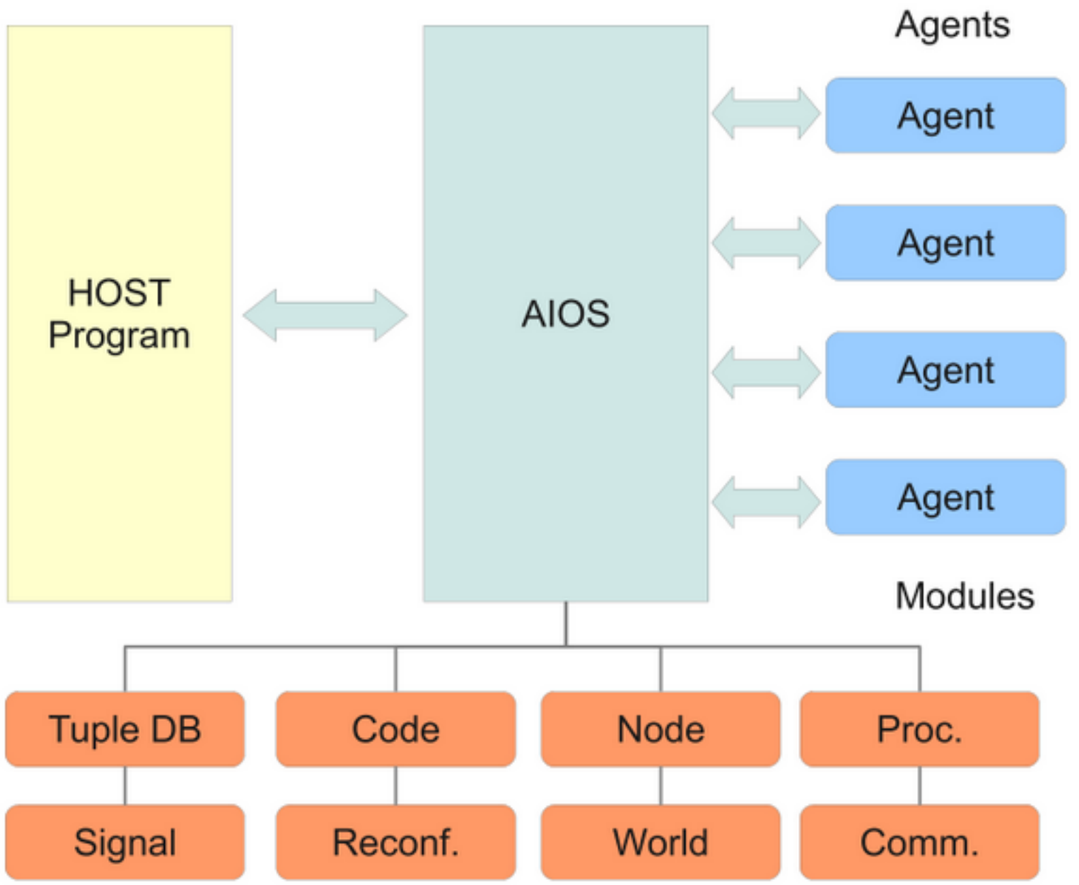


Abb. 48. Das Agent Input-Outut System als Schnittstelle zwischen Agenten und JAM und einer Hostplattform (bzw. Applikation)

JAM

Agentenrollen

- In der realen Welt ist die Anwendungssicherheit ein wichtiges Schlüsselmerkmal einer verteilten Agentenplattform.
- Die Ausführung von Agenten und der Zugriff auf Ressourcen müssen kontrolliert werden, um Denial-of-Service-Angriffe, Agent-Masquerading, Spionage oder anderen Missbrauch zu verhindern.
- Daher haben Agenten unterschiedliche **Zugriffsebenen (Rollen)**:

  0. Gast (nicht vertrauenswürdig, semi-mobil)
  1. Normal (vielleicht vertrauenswürdig, mobil)
  2. Privilegiert (vertrauenswürdig, mobil)
  3. System (sehr vertrauenswürdig, System relevant, lokal, nicht mobil)

- Die unterste Ebene (0) ermöglicht keine Agentenreplikation, Migration oder das Erstellen neuer Agenten.
- Die JAM-Plattform entscheidet über die Sicherheitsstufe für neue empfangene Agenten. Ein Agent kann keine Agenten mit einer höheren Sicherheitsstufe als die eigene erstellen.

JAM

- Die höchste Stufe (3) hat einen erweiterten AIOS mit Host-Plattform-Gerätezugriffsfähigkeiten.
- Agenten können Ressourcen (z.B. CPU-Zeit) aushandeln und ein Level-Raise erreichen, das mit einem Schlüssel gesichert ist, der die erlaubten Upgrades definiert.
  - » Die Systemebene (3) kann nicht ausgehandelt werden.
- Der Schlüssel ist knotenspezifisch. Eine Gruppe von Knoten kann sich einen gemeinsamen Schlüssel teilen (durch einen Server-Port identifiziert der den JAM Knoten identifiziert).
- Ein Schlüssel besteht aus einem Server-Port, einem Rechtefeld und einem verschlüsselten Schutzfeld das das Rechtefeld enthält, das mit einem zufälligen Port generiert wird, der nur dem Server (Knoten) bekannt ist.

JAM

Schlüssel (Capabilities)

- Ein Schlüssel kann verwendet werden um:
  - » Einen Agenten von einer Plattform *A* nach *B* zu migrieren (*B* verlangt den Schlüssel um den Agenten auszuführen);
  - » Eine neue Agentenrolle (Stufe) auszuhandeln;
  - » In einer Agententrolle neue Ressourcen auszuhandeln (CPU, MEM, TS, SCHED, ..);
  - » Um neue Agenten erzeugen und andere terminieren zu können

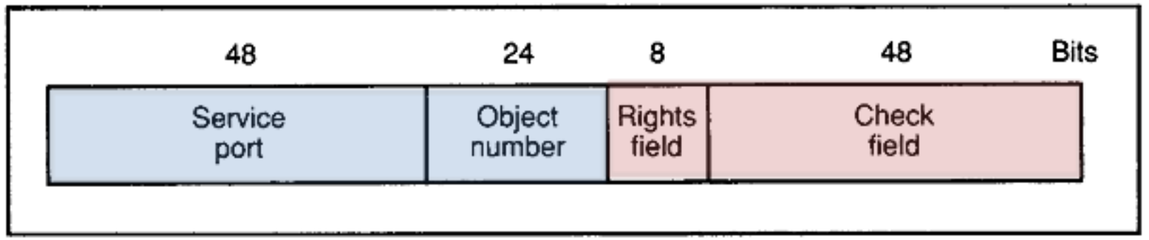


Abb. 49. Aufbau einer Capability: Der Serverport bindet die Capability an einen Service, die Objektnummer ist optional und kann eine Unterklasse des Service oder der zu schützenden Ressource darstellen, das Rechtefeld kodiert die möglichen Operationen der Serviceklasse, und das Schutzfeld (enthält Rechtefeld nochmals verschlüsselt) schützt die Capability gegen Manipulation.

JAM

Schutz von Capabilities

- Das Rechtefeld ist zentral das es die möglichen Operationen mit einer Capability erlaubt.
  - CPU Zeit erhöhen
  - Levelraise
  - Migartion usw.
- Damit das Rechtefeld nicht manipuliert werden kann ohne dass die Capability ungültig wird (und ggf. die Objektnummer) wird ein Schutzfeld erstellt welches die Rechte mit einem privaten Schlüssel (Port) mittels einer One-way Funktion verschlüsslet.
  - Nur der Service / Agentenplattform kennt den privaten Schlüssel

JAM

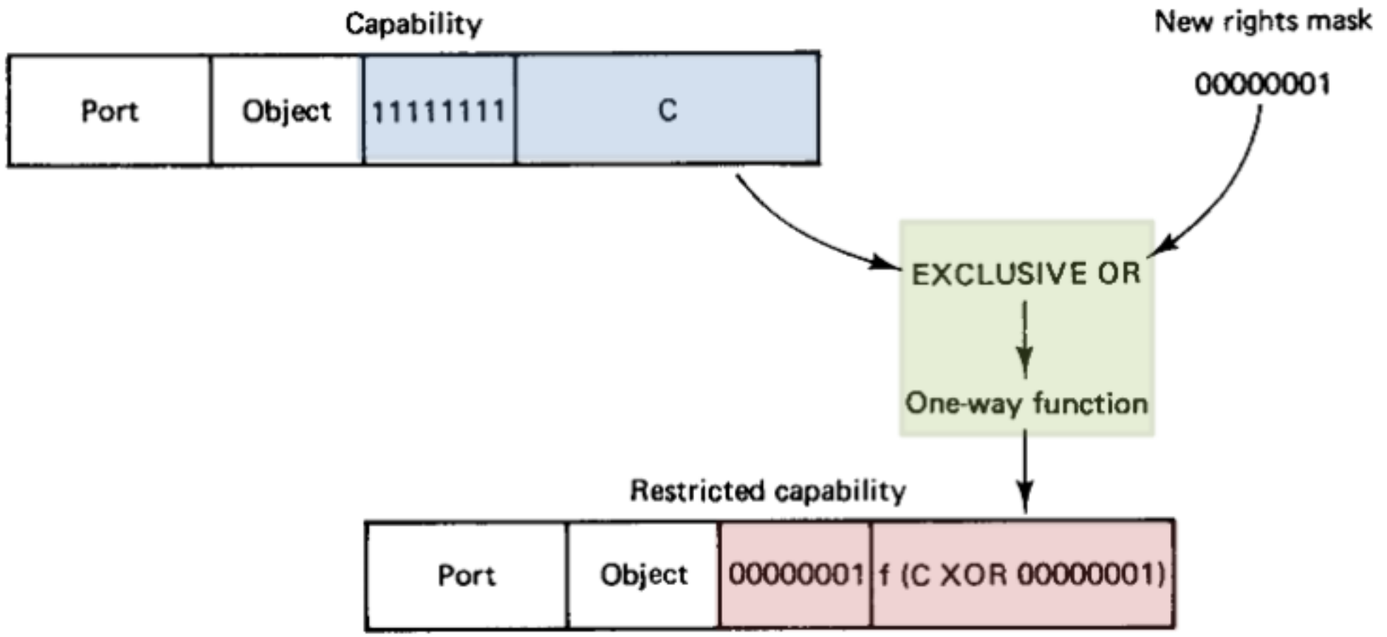


Abb. 50. Erzeugung einer öffentlichen restriktiven Capability aus einer privaten nicht eingeschränkten (enthält privaten Schlüssel *C*) mittels One-way Verschlüsselungsfunktion *f*(*C* xor *R*)

JAM

Kommunikation und Netzwerke

- JAM Plattformen können in beliebigen Netzwerken miteinander verbunden werden.
- Eine Vielzahl von Kommunikationsprotokolle sind verwendbar:
  - RS232
  - UDP
  - TCP
  - HTTP
- Beliebige Netzwerktoplogien können gebildet werden (physisch wie logisch):
  - Stern
  - Gitter (1D/2D/3D)
  - Bus
  - Intranet und Internet (allg. Graphen)
- AMP: Agent Management Port als gemeinsames Protokoll und Interface in heterogenen Systemen

JAM

- AMP definiert eine Menge von Nachrichten die dem Transport von
  - Agenten,
  - Signalen und Tupeln,
  - und Handshakes dienen.

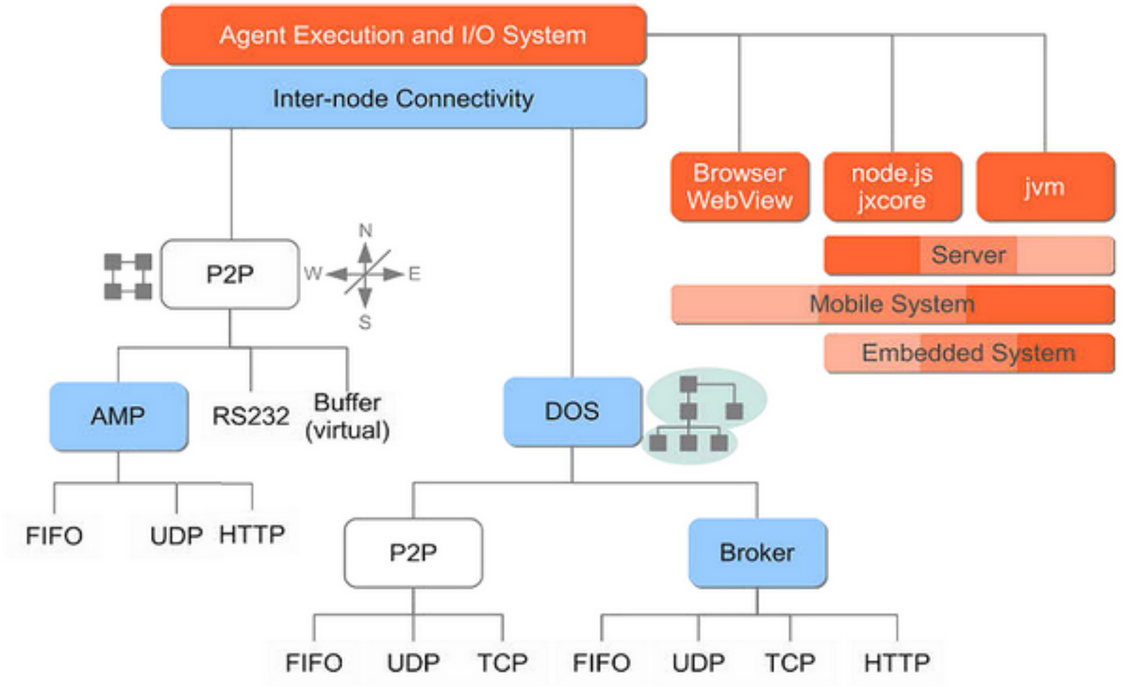


Abb. 51. JAM Konnektivität und eine breites Spektrum an unterstützten JavaScript Host Plattformen

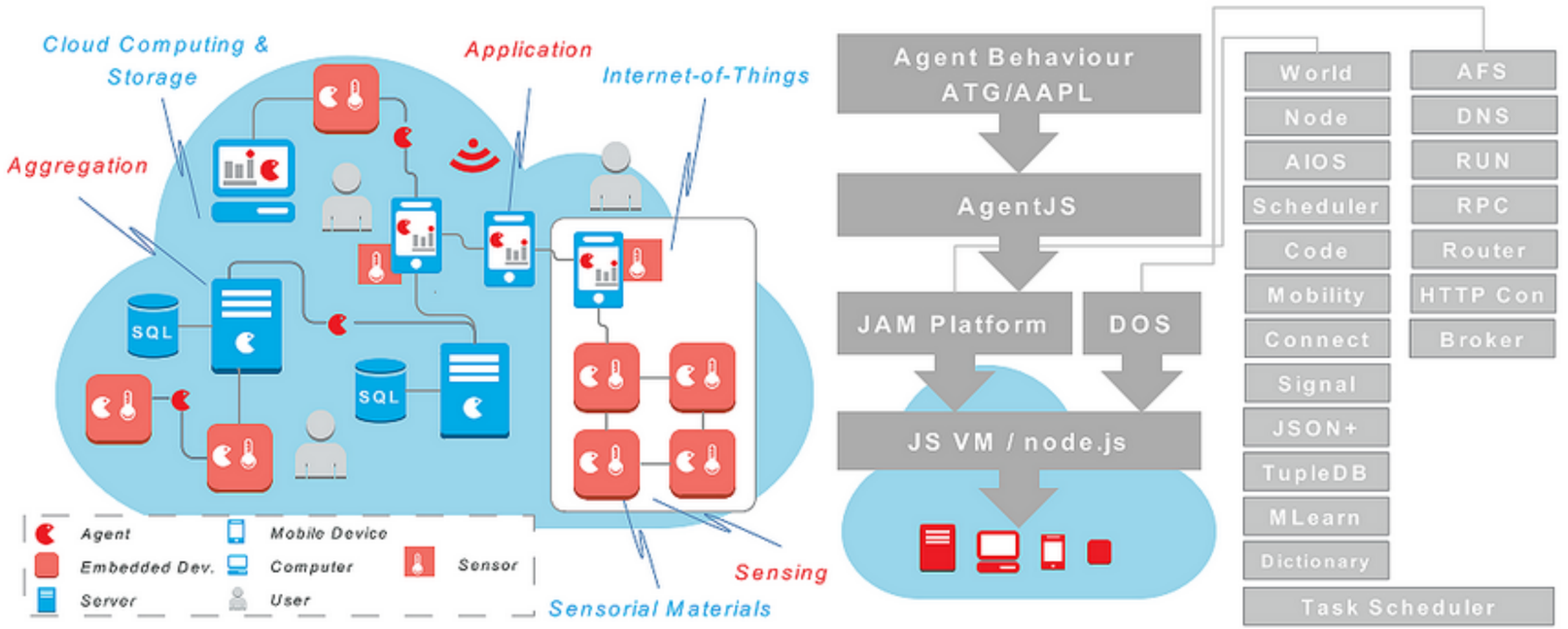
JAM

DOS

- Eine optionale Distributed Organization System (DOS) Schicht implementiert verteilte Systeme via Remote Procedure Calls (RPC)
- DOS stellt Verzeichnis und Namensdienste sowie eine Klienten-Server Architektur zur Verfügung
- AMP wird über DOS über IP/HTTP implementiert
- Ein **Brokerserver** oder ein hierarchisches Netzwerk aus Brokern stellt die Sichtbarkeit und Erreichbarkeit von JAM Platfformen auch in privaten Netzen sicher.

JAM

- JAM kann daher in stark heterogenen Umgebungen und Geräten eingesetzt werden wie dem
  - » Internet (Mobile, Server)
  - » Internet-der-Dinge (IoT) (Mobile und Eingebettete Systeme)
  - » Clouds (Server, WEB Browser)
  - » Materialintegrierten Systemen (Mikrochip)



JAM

Scheduling

- Alle Agenten (>1000 pro JAM) werden über einen Scheduler in einem einzigen Thread (eine JS VM Instanz) verarbeitet.
- Der Scheduler ist nicht preemptiv. D.h. Agentencode in Ausführung kann nicht unterbrochen werden!
  - » Ausnahme: Es gibt einen Watchdog Mechanismus der Agentencode nach Ablauf einer Zeitscheibe abbricht (typisch 20ms)
  - » wie bei JADE: Mikroaktivitäten!
- Dabei sind verschiedene Teilausführungen bei einem Agenten zu unterscheiden:
  - » Aktivitätsfunktion
  - » Transitionfunktion
  - » Signal- oder Exceptionhandler
  - » Mikroaktivitätsblock (Scheduling Block)

JAM

Zwei zentrale Probleme gab es bei JAM und AgentJS zu lösen:

1. Zeitscheibenverfahren
   - » Faires Scheduling
   - » Wird entweder durch eine modifizierte JS VM (Watchdog) oder durch Injektion von Checkpointing Funktionen in den Agentencode erreicht
   - » Der Ablauf der Zeitscheibe führt in beiden Fällen zu einer Auslösung einer Exception
2. Isolation des Agentencodes und Agentenprozesses
   - » Es gibt nur eine JS Programmkontext pro JS VM Instanz!
   - »

JAM

Maschinelles Lernen

- Maschinelles Lernen ist inhärent mit Agenten verknüpft
- Lernen besteht aus zwei “Komponenten”:
  - » Algorithmen die ein Modell aus Daten erzeugen
  - » Modelle und deren Repräsentation (Datenstruktur)
- Aber JAM unterstützt mobile Agenten → Mobile Modelle sind erforderlich die ein Agent transportieren kann!
- Die Algorithmen werden von der Plattform zur Verfügung gestellt, die Daten vom Agenten → Entkopplung von Algorithmen und Modellen!
- Portable Modelle:
  - » Entscheidungsbäume (Knoten und Kanten sind portabel)
  - » Neuronale Netze (Struktur und Gewichte sind portabel)

VERTEILTES VERHALTEN UND GRUPPEN

GRUPPENENTSCHEIDUNG UND VERHANDLUNG

- In Multiagentensystemen gibt es in der Regel Organisationsstrukturen
- In diesen Strukturen sollen gemeinsame Ziele entweder vorgegeben und umgesetzt oder verhandelt werden.

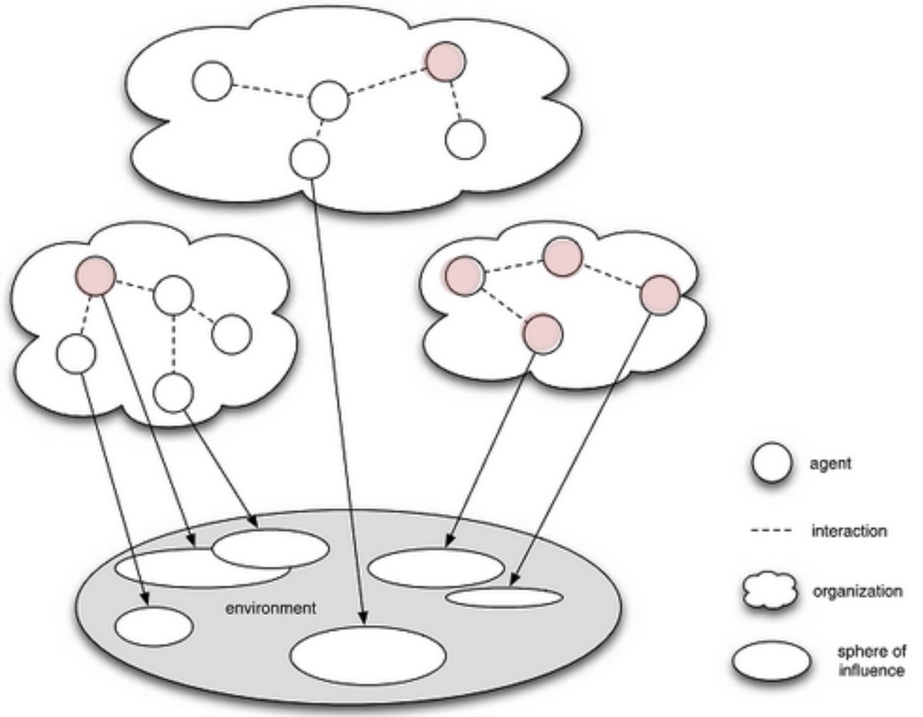


Abb. 52. Typische Gruppenstrukturen in Multiagentensystemen

GRUPPENENTSCHEIDUNG UND VERHANDLUNG

- Dabei können Gruppenentscheidungen und Verhandlungen auf Basis von Nützlichkeitsfunktionen geführt werden
- Es sei eine Gruppe von Agenten (Prozessen) $G=\{ag_1,ag_2,..,ag_m\}$
- Es gibt nun verschiedene Stufen der Verhandlung:
  - Wahl eines Gruppenführers (Leader) oder Mediators
  - Abgabge von Stimmen / Absichten
  - Bestimmung eines gemeinsamenen Ergebnisses mittels Konsensalgorithmus bzw.
  - Wahl: Über Pluralität, z.B. mit einer Mehrheitsentscheidung
  - Benachrichtigung der Gruppenteilnehmer über Ergebnis
- Nachteil des Mehrheitsentscheids: Das Volk ist dumm! D.h. es könnte eine besseres Ergebnis für ein MAS erzielt werden, wenn Fraktionen in den Stimmabgaben berücksichtigt werden würden (differenziertes und gewichtetes Ergebnis) …

VERHANDLUNG UND ABSTIMMUNG

- Verhandlung in Gruppen und Abstimmung über ein gemeinsames Ergebnis (Konsens) ist ein weiteres wichtiges Beispiel für verteiltes Gruppenverhalten → Kommunikation!
- Ein Verhandlungsproblem ist ein Problem, bei dem mehrere Agenten versuchen, zu einer Vereinbarung oder einem Deal zu kommen.
- Es wird angenommen, dass jeder Agent gegenüber allen möglichen Deals eine Präferenz hat.
- Die Agenten senden sich Nachrichten in der Hoffnung, einen Deal zu finden, auf den sich alle Agenten einigen können.
- Diese Agenten stehen vor dem Problem:
  - Sie wollen ihren eigenen Nutzen maximieren, sehen sich aber auch der Gefahr eines Zusammenbruchs der Verhandlungen oder des Ablaufs einer Frist für die Vereinbarung gegenüber.
  - Daher muss jeder Agent sorgfältig verhandeln und jeden Nutzen abwägen, den er aus einem Versuch gegen einen möglicherweise besseren Abschluss oder das Risiko eines Ausfalls bei den Verhandlungen zieht.

VERHANDLUNG UND ABSTIMMUNG

Automatisierte Aushandlung kann in Multiagent-Systemen sehr nützlich sein, da sie eine verteilte Methode zur Aggregation von verteiltem Wissen bietet.

- Verschiedene Protokolle existieren, z.B.,
  - » Monotone Konzession
  - » Monotone Konzession mit zusätzlicher Risikobewertung
  - » Schrittweise Verhandlung
- Häufig in der Form (Monotone Konzession, Vidal,2010)

```
0. δi <- arg maxδ ui(δ)
1. Propse δi
2. Receive δj proposal
3. if ui(δj) > ui(δi)
4.   then Accept δj
5.   else δi <- δi' such that uj(δi') >= e+uj(δi)
6. loop 2.
```

» mit $u_i(\delta)$: Nützlichkeitsfunktion eines Deals δ des i-ten Agenten

VERTEILTER KONSENS

- Ein verteilter Konsensalgorithmus hat das Ziel in einer Gruppe von Prozessen oder Agenten eine gemeinsame Entscheidung zu treffen
- Zentrale Eigenschaften:
  - Zustimmung/Übereinstimmung
  - Terminierung; Lebendigkeit und Deadlockfreiheit
  - Gültigkeit; Robustheit gegenüber Störungen wie fehlerhaften Nachrichten oder Ausfälle von Gruppenteilnehmern
- Beim Konsens kann ein Master-Slave Konzept oder ein Gruppenkonzept mit Leader/Commander und Workern verwendet werden.
  - Beim Master-Slave Konzept kommunizieren nur Slaves mit dem Master
  - Bei Gruppenkonzept (i.A. mit einem Leader) kommunizieren auch alle Gruppenteilnehmer untereinander
- Durch Störung (Fehler oder Absicht) kann es zu fehlerhaften bis hin zu fehlgeschlagenen Konsens kommen.

VERTEILTER KONSENS

- Bedingungen für Interaktive Konsistenz:
  - » IC1: Jeder Worker empfängt die *gleiche* Anweisung vom Leader!
  - » IC2: Wenn der Leader *fehlerfrei* arbeitet, dann empfängt jeder *fehlerfreie* Worker die Anweisung die der Leader sendete!

Byzantinisches Generalproblem

- Beispiel: In einer Gruppe aus drei Prozessen/Agenten ist einer fehlerhaft bzw. versendet fehlerhafte Nachrichten (durch Störung oder Absicht) mit Anweisungen 0/1 (schließlich ein Konsensergebnis) [E]

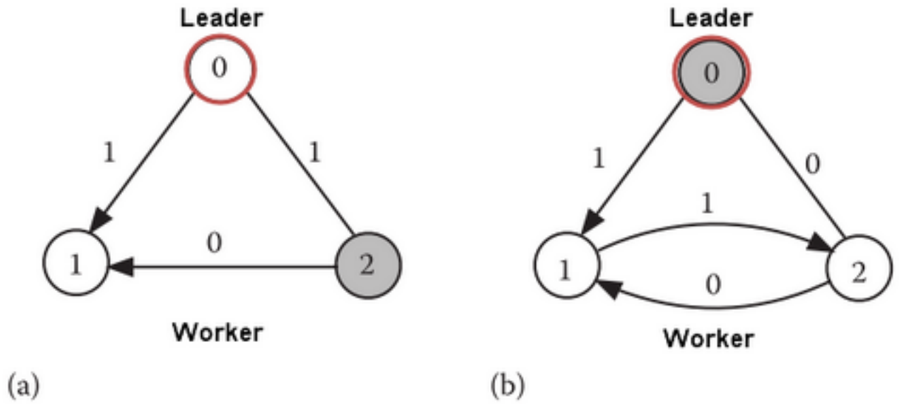


Abb. 53. Byzantinisches Generalproblem: (a) Leader 0 ist fehlerfrei, Worker 2 ist fehlerhaft (b) Leader 0 ist fehlerhaft, Worker 1 und 2 sind fehlerfrei [E]

VERTEILTER KONSENS

- Jeder Worker der Nachrichten empfängt ordnet diese nach direkten und indirekten (von Nachbarn)
- **Fall (a)**: Prozess 2 versendet fehlerhafte Nachricht mit Anweisung 0, Prozess 1 empfängt eine direkte Nachricht mit Anweisung 1 und eine indirekte mit (falschen) Inhalt Anweisung 0
  - » Bedingung IC1 ist erfüllt. Um Bedingung IC2 zu erfüllen wird Worker 1 die direkte Anweisung 1 von Prozess 0 (Leader) auswählen → Konsens wurde gefunden
- **Fall (b)**: Prozess 0 (Leader) versendet an Prozess 1 richtige Nachricht mit Anweisung 1 und falsche Nachricht mit Anweisung 0 an Prozess 1
  - » Würde Prozess 1 wieder zur Erfüllung von IC2 eine Entscheidung treffen (Anweisung 1 auswählen), dann wäre IC1 verletzt. Wie auch immer Prozess 1 entscheidet ist entweder IC1 oder IC2 verletzt → **Unentscheidbarkeit** → Kein Konsens möglich

VERTEILTER KONSENS

Das nicht-signierte Nachrichtenmodell erfüllt die Bedingungen:

1. Nachrichten werden während der Übertragung nicht verändert (aber keine harte Bedingung).
2. Nachrichten können verloren gehen, aber die Abwesenheit von Nachrichten kann erkannt werden.
3. Wenn eine Nachricht empfangen wird (oder ihre Abwesenheit erkannt wird), kennt der Empfänger die Identität des Absenders (oder des vermeintlichen Absenders bei Verlust).

- Algorithmen zur Lösung des Konsensproblems müssen *m* fehlerhafte Prozesse annehmen (bzw. fehlerhafte Nachrichten)

Der OM(m) Algorithmus

- Ein Algorithmus der einen Konsens erreicht bei Erfüllung der Bedingungen IC1 und IC2 mit bis zu *m* fehlerhaften Prozesse bei insgesamt $n \geq 3m+1$ Prozessen mit nicht signierten ("mündlichen") Nachrichten.

  i. Leader *i* sendet einen Wert $v \in \{0, 1\}$ an jeden Worker $j \neq i$.
  ii. Jeder Worker *j* akzeptiert den Wert von *i* als Befehl vom Leader *i*.

VERTEILTER KONSENS

Definition 14.
Algorithmus OM(m)

1. Leader *i* sendet einen Wert $v \in \{0, 1\}$ an jeden Worker $j \neq i$.
2. Wenn $m > 0$, dann beginnt jeder Worker *j*, der einen Wert vom Leader erhält, eine neue Phase, indem er ihn mit OM(m-1) an die verbleibenden Worker sendet.
   - In dieser Phase fungiert *j* als Leader.
   - Jeder Arbeiter erhält somit (n-1) Werte: (a) einen Wert, der direkt von dem Leader *i* von OM (m) empfangen wird und (b) (n-2) Werte, die indirekt von den (n-2) Workern erhalten werden, die aus ihrem Broadcast OM(m-1) resultieren.
   - Wird ein Wert nicht empfangen wird er durch einen Standardwert ersetzt.
3. Jeder Worker wählt die Mehrheit der (n-1) Werte, die er erhält, als Anweisung vom Leader *i*.

VERTEILTER KONSENS

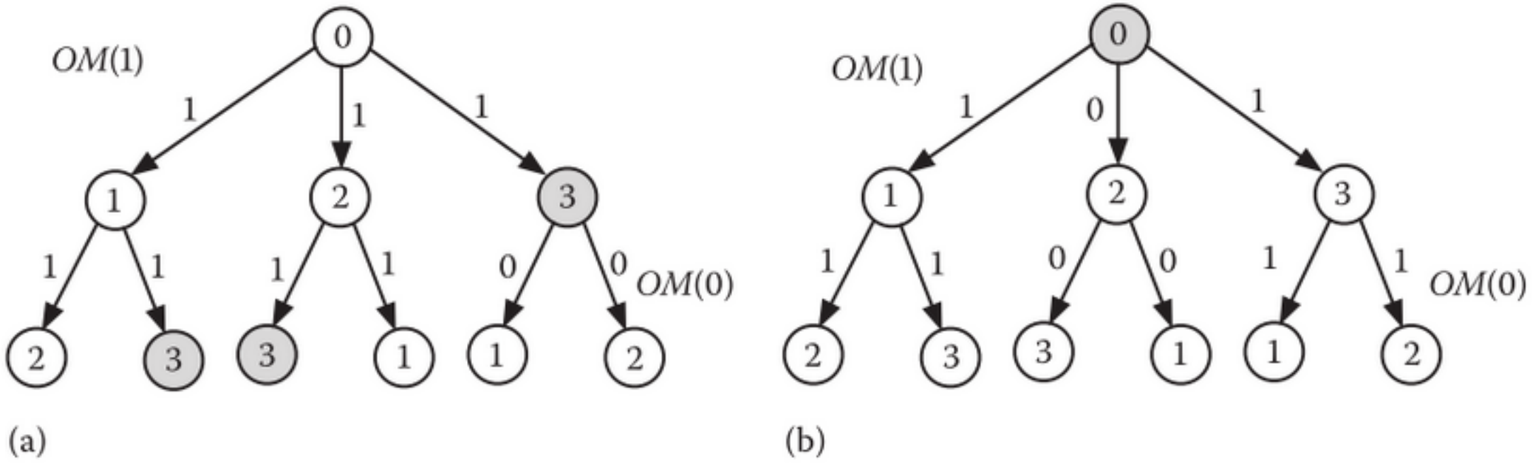


Abb. 54. Eine Illustration von OM(1) mit vier Prozessen und einem fehlerhaften Prozess: die Nachrichten auf der oberen Ebene spiegeln die Eröffnungsnachrichten von OM(1) wider und die auf der unteren Ebene spiegeln die OM(0)-Meldungen wider, die von den Mitteilungen der oberen Ebene ausgelöst werden. (a) Prozess 3 ist fehlerhaft. (b) Prozess 0 (Leader) ist fehlerhaft. [E]

VERTEILTER KONSENS

Der Paxos Algorithmus

- Paxos ist ein Algorithmus zur Implementierung von fehlertoleranten Konsensfindungen.
- Er läuft auf einem *vollständig verbundenen Netzwerk* von n Prozessen und toleriert bis zu m Ausfälle, wobei $n \geq 2m+1$ ist.
- Prozesse können abstürzen und Nachrichten können verloren gehen, byzantinische Ausfälle (absichtliche Verfälschung) sind jedoch zumindest in der aktuellen Version ausgeschlossen.
- Der Algorithmus löst das Konsensproblem bei Vorhandensein dieser Fehler auf einem *asynchronen System von Prozessen*.

» Obwohl die Konsensbedingungen Zustimmung, Gültigkeit und Terminierung sind, garantiert Paxos in erster Linie die Übereinstimmung und Gültigkeit und nicht die Beendigung - es ermöglicht die Möglichkeit der Beendigung nur dann, wenn es ein ausreichend langes Intervall gibt, in dem kein Prozess das Protokoll neu startet.

VERTEILTER KONSENS

- Ein Prozess kann drei verschiedene Rollen wahrnehmen:
  - Antragsteller,
  - Akzeptor und
  - Lerner.

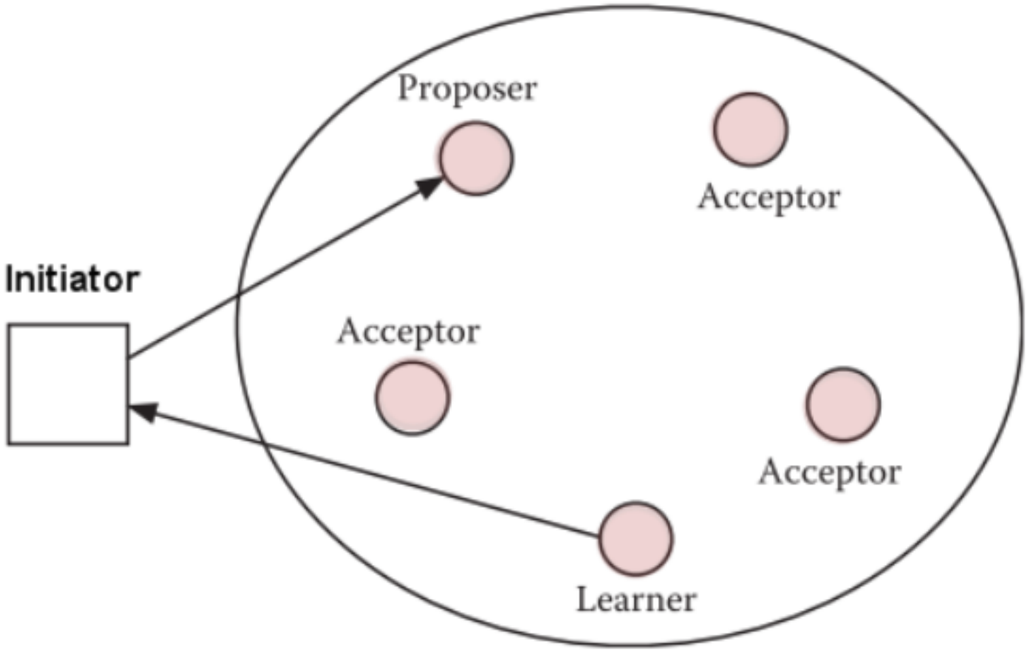


Abb. 55. Typische Rollenverteilung beim Paxos Algorithmus

VERTEILTER KONSENS

- Die **Antragsteller** reichen die vorgeschlagenen Werte im Namen eines Initiators ein,
- die **Akzeptoren** entscheiden über die Kandidatenwerte für die endgültige Entscheidung und
- die **Lernenden** sammeln diese Informationen von den Akzeptoren und melden die endgültige Entscheidung dem Initiator zurück.

- Ein Vorschlag, der von einem Antragsteller gesendet wird, ist ein Tupel (*v*, *n*), wobei *v* ein Wert und *n* eine Sequenznummer ist.
- Wenn es nur einen Akzeptor gibt, der entscheidet, welcher Wert als Konsenswert gewählt wird, dann wäre diese Situation zu einfach. Was passiert, wenn der Akzeptor abstürzt? Um damit umzugehen, gibt es mehrere Akzeptoren.
- Ein Vorschlag muss von mindestens einem Akzeptor bestätigt werden, bevor er für die endgültige Entscheidung in Frage kommt.

VERTEILTER KONSENS

- Die Sequenznummer wird verwendet, um zwischen aufeinander folgenden Versuchen der Protokollanwendung zu unterscheiden.
- Nach Empfang eines Vorschlags mit einer größeren Sequenznummer von einem gegebenen Prozess, verwerfen Akzeptoren die Vorschläge mit niedrigeren Sequenznummern.
- Schließlich akzeptiert ein Akzeptor die Entscheidung der Mehrheit.

Phasen des Paxos Algorithmus

1. **Die Vorbereitungsphase**
   - » Jeder Antragsteller sendet einen Vorschlag (v, n) an jeden Akzeptor
   - » Wenn n die größte Sequenznummer eines von einem Akzeptor empfangenen Vorschlags ist, dann sendet er ein *ack* $(n, \perp, \perp)$ an seinen Vorschlager
   - » Hat der Akzeptor einen Vorschlag mit einer Sequenznummer $n' < n$ und einem vorgeschlagenen Wert v akzeptiert, antwortet er mit $ack(n, v, n')$.

VERTEILTER KONSENS

2. **Aufforderung zur Annahme eines Eingabewertes**
   - » Wenn ein Antragsteller *ack*(n, $\perp$, $\perp$) von einer Mehrheit von Akzeptoren empfängt, sendet er an alle Akzeptoren *accept*(v, n) und fordert sie auf, diesen Wert zu akzeptieren.
   - » Wenn ein Akzeptor in Phase 1 einen *ack*(n, v, n') an den Antragsteller zurücksendet, muss der Antragsteller den Wert v mit der höchsten Sequenznummer in seiner Anfrage an die Akzeptoren einbeziehen.
   - » Ein Akzeptor akzeptiert einen Vorschlag (v, n), sofern er nicht bereits zugesagt hat, Vorschläge mit einer Sequenznummer größer als n zu berücksichtigen.
3. **Die endgültige Entscheidung**
   - » Wenn eine Mehrheit der Akzeptoren einen vorgeschlagenen Wert akzeptiert, wird dies der endgültige Entscheidungswert. Die Akzeptoren senden den akzeptierten Wert an die Lernenden, wodurch sie feststellen können, ob ein Vorschlag von einer Mehrheit von Akzeptoren akzeptiert wurde.

DIVIDE-AND-CONQUER

Replikation

- Replikation dient:
  - Der Gruppenbildung mit Gemeinsamkeiten,
  - Der Vererbung von Verhalten und Spezialisierung
  - Der räumlichen Exploration
  - ..

Beim Divide-and-Conquer" (Teile und herrsche) Ansatz wird versucht ein komplexes Problem soweit rekursiv zu zerlegen dass am Ende nur noch triviale Probeleme übrig bleiben!

- Beispiel: Verteiltes Sensornetzwerk und Bestimmung von geometrisch ausgedehnter Sensoraktivität und Sensorfusion

VERTEILTER INFORMATIONSAUSTAUSCH

Problem: Wie können Informationen von der Informationsquelle zu Informationssenken, d.h. Agenten die an den Informationen interessiert sind, zugestellt werden?

- Die Replikation kann verwendet werden, um die Zustellungswahrscheinlichkeit zu erhöhen und die Latenz zu verringern.
- Lernende Agenten können die Pfadsuche verbessern, indem sie ihre Reisegeschichte berücksichtigen.
- Datenzentrierte gerichtete Diffusionsalgorithmen können leicht mit autonomen mobilen Agenten implementiert werden.
- *Problem: Flutung des Netzwerks mit Agenten!*

Ereignisbasierte Verteilung

Eine Ereignisbenachrichtigungsalgorithmus kann verwendet werden um effizient eine Kommunikation zwischen Informationsquellen und Senken herzustellen.

» Dazu können Benachrichtigungsagenten verwendet werden die entlang eines Pfades Ereignisse markieren, z.B. dass ein Sensorwert über einer Schwelle liegt.

VERTEILTER INFORMATIONSAUSTAUSCH

- Suchagenten werden dann benutzt die Ereignisse zu finden die von den Benachrichtigungsagenten hinterlassen wurden.
- Der Ursprung eines Ereignisses kann durch Rückverfolgung des Pfades gefunden werden.

Random Walk

- Eine einfache Möglichkeit besteht darin, dass der Weg zum Empfänger (Datensenke) durch zufällige Richtungswechsel und Migration von datentragenden Agenten erfolgt.
  - Es wird kein geometrisches Modell und Kenntnis des Netzwerkes benötigt

Gerichtete Diffusion

- Durch Replikation der Information bzw. der datentragenden Agenten und grob richtungsorienterte Zustellung mit überlagerten Random Walk und/oder ggf. Backtracking um tote Netzwerkenden (Seitenarme) zu entkommen.
  - Dazu wird ein teilweise geometrisches Modell des Netzwerkes benötigt

VERTEILTER INFORMATIONSAUSTAUSCH

Beispiel eines Event Agentens in AgentJS

```
function event (dir,target) {
  this.sensors; this.delta={x:0,y:0};
  this.dir=dir; this.target=target;
  this.next = init;

  this.act = {
    init : function () {
      this.delta={x:0,y:0}; this.sensors=[] },
    end : function () { kill() },
    sense : function () {
      rd(['SENSOR',_],function(t){
        this.sensors.push(t[1])
      }) },
    deliver: function () {
     ..
     out(['SENSORS',sensmat]);
     broadcast('node',0,'DELIVER') }
```

VERTEILTER INFORMATIONSAUSTAUSCH

```
  move : function () {
    switch (this.dir) {
      case DIR.NORTH:
       if (!link(DIR.NORTH)) // edge reached
         this.dir=DIR.WEST,fork({dir:DIR.EAST,target:this.target});
      case DIR.WEST:
       if (!link(DIR.WEST)) // edge reached
         this.dir=DIR.NORTH,fork({dir:DIR.SOUTH,target:this.target,next:move});
      ..
    }
    switch (this.dir) {
      case DIR.NORTH: this.delta.y--; break;
      case DIR.WEST:  this.delta.x--; break;
      ..
    }
    moveto(this.dir) } }

this.trans = {
  init: sense ,
  sense: function () {
    if ((!exists([this.target])||zero(this.delta))&&this.dir!=DIR.ORIGIN)
      return 'move'; else return 'deliver' },
  move: sense,
  deliver: end }}
```

VERTEILTER INFORMATIONSAUSTAUSCH

Potentialfeldansatz

- Eine weitere Möglichkeit besteht darin dass dateninteressierte Agenten “farbige” Markierungen in ihrer Umgebung mit Gradienten verteilen
  - » Die “Farbe” charakterisiert die Informations/Datenart, z,B. Temperaturssensor, Ortsinformation, usw.
- Ein datentragender Agent wird entlang des Gradienten der Markierungen einen Weg zur Datensenke finden

VERTEILTE MUSTERERKENNUNG IN SENSORNETZWERKEN

Ausgangssituation: Verteiltes Sensornetzwerk mit Knoten in einem Maschengitternetzwerk.

- Fehlerhafte oder verrauschte Sensoren können Datenverarbeitungsalgorithmen erheblich stören.
- Es ist notwendig, fehlerhafte Sensoren von gut arbeitenden Sensoren zu isolieren.
- Üblicherweise werden Sensorwerte innerhalb eines räumlich nahen Bereichs korreliert, beispielsweise in einem räumlich verteilten mechanischen Lastmonitoringnetzwerks unter Verwendung von Dehnungssensoren.
- Das Ziel des folgenden MAS ist es, ausgedehnte korrelierte Bereiche erhöhter Sensorintensität (im Vergleich zur Nachbarschaft) aufgrund mechanischer Verzerrungen zu finden, die von extern angelegten Lastkräften herrühren.

VERTEILTE MUSTERERKENNUNG IN SENSORNETZWERKEN

- Es wird ein verteiltes gerichtetes Diffusionsverhalten und eine Selbstorganisation verwendet, die von einem Bildmerkmalsextraktionsansatz abgeleitet sind.
- Es handelt sich hierbei um einem selbstadaptiven Kantendetektor.
- Eine einzelne sporadische Sensoraktivität, die nicht mit der umgebenden Nachbarschaft korreliert ist, sollte von einer erweiterten korrelierten Region unterschieden werden, die das zu erfassende Merkmal darstellt.

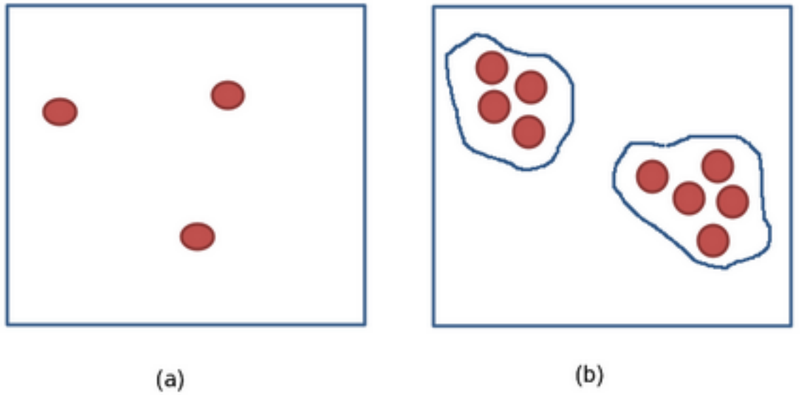


Abb. 56. (a) Nichtkorellierte Sensorstimuli (b) Korrelierte Sensorstimulibereiche

VERTEILTE MUSTERERKENNUNG IN SENSORNETZWERKEN

Der Algorithmus

Die Merkmals-Erkennung wird vom mobilen Explorationsagenten durchgeführt, der zwei verschiedene Verhaltensweisen unterstützt: Diffusion und Reproduktion.

- Das Diffusionsverhalten wird verwendet, um sich in einem ausgedehnten Bereich zu bewegen, der hauptsächlich durch die Lebensdauer des Agenten begrenzt ist, und um das Merkmal zu detektieren;
  - » hier den Bereich mit erhöhter mechanischer Verzerrung (genauer gesagt die Kante eines solchen Bereichs).
- Die Erkennung des Merkmals wird durch das Reproduktionsverhalten verstärkt, das den Agenten veranlasst, am aktuellen Knoten zu bleiben, eine Merkmalsmarkierung zu setzen und mehr Explorationsagenten in der Nachbarschaft auszusenden.

VERTEILTE MUSTERERKENNUNG IN SENSORNETZWERKEN

- Der lokale Stimulus *H(i,j)* für einen Explorationsagenten, der sich an einem spezifischen Knoten mit der Koordinate (*i,j*) befindet, ist gegeben durch:

$$H(i,j) = \sum_{s=-R}^{R} \sum_{t=-R}^{R} \{\|S(i+s,j+t) - S(i,j)\| \leqslant \delta\}$$

S : Sensor Signal Strength

R : Square Region around (i,j)

- Die Berechnung von *H* an der aktuellen Position (*i,j*) des Agenten erfordert die Sensorwerte innerhalb des quadratischen Bereichs (der Region von Interesse ROI) *R* um diesen Ort herum.
- Wenn ein Sensorwert *S(i+s,j+t)* mit $i,j \in \{-R, .., +R\}$ ähnlich dem Wert *S* an der aktuellen Position ist (Differenz ist kleiner als der Parameter *d*), wird *H* um eins erhöht.

VERTEILTE MUSTERERKENNUNG IN SENSORNETZWERKEN

- Wenn der H-Wert innerhalb eines parametrierten Intervalls $D = [e_0, e_1]$ liegt, hat der Explorationsagent das Feature erkannt und verbleibt am aktuellen Knoten, um neue Explorationsagenten zu reproduzieren, die an die Umgebung gesendet werden.
- Wenn H außerhalb dieses Intervalls liegt, wird der Agent zu einem anderen Knoten wechseln und die Exploration (Diffusion) neu starten.
- Die Berechnung von H erfolgt durch eine verteilte Berechnung von Teilsummenausdrücken durch Aussenden von Kind-Agenten an die Nachbarschaft, die selbst mehr Agenten aussenden können, bis die Grenze der Region R erreicht ist.
- Jeder untergeordnete Agent kehrt zu seinem Ursprungsknoten zurück und übergibt den Teilsummenbegriff an seinen übergeordneten Agenten.

VERTEILTE MUSTERERKENNUNG IN SENSORNETZWERKEN

- Da ein Knoten in der Region R von mehr als einem Kind-Agenten besucht werden kann, setzt der erste Agent, der einen Knoten erreicht, eine Markierung MARK.
  - » Wenn ein anderer Agent diese Markierung findet, kehrt er sofort zum übergeordneten Agenten zurück.
- Dieser Mehrwegebesuch hat den Vorteil einer erhöhten Wahrscheinlichkeit, Knoten mit fehlenden (nicht arbeitenden) Kommunikationsverbindungen zu erreichen.
- Ein Eventagent, der von einem Sensingagenten erzeugt wird, liefert schließlich Sensorwerte an Rechenknoten, was hier nicht berücksichtigt wird.

VERTEILTE MUSTERERKENNUNG IN SENSORNETZWERKEN

Das Agentenverhalten

» Definition von Typen, Körpervariablen, und Hauptklasse Explorer mit den Aktivitäten *init* und *percept*

```
1   κ: { SENSORVALUE,FEATURE, H, MARK }     set of key symbols
2   ξ: { TIMEOUT, WAKEUP }                  set of signals
3   δ: { NORTH,SOUTH, WEST, EAST, ORIGIN }  set of directions
4   ε1 =3; ε2 = 6; MAXLIVE = 1;             some constant parameters
5
6   Ψ Explorer: (dir,radius) → {
7     * Body Variables *
8     Σ: { dx, dy, live, h, s0, backdir, group }   global persistent variables
9     σ: { enoughinput, again, die, back, s, v }   local  temporary  variables
10
11    Activities
12    α init: {
13      dx ← 0; dy ← 0; h ← 0; die ← false; group ← ℜ{0..10000};
14      if dir ≠ ORIGIN then
15        ⇔dir; backdir ← ϖ(dir)
16      else
17        live ← MAXLIVE; backdir ← ORIGIN
18      ∇⁺(H,$self,0);
19      ∇%(SENSORVALUE,s0?)
20    }
21    α percept: {
22      enoughinput ← 0;
23      ∀{nextdir∈δ | nextdir ≠ backdir ∧ ?Λ(nextdir)} do
24        enoughinput++;
25        Θ→Explorer.child(nextdir,radius)
26      τ⁺(ATMO,TIMEOUT)
27    }
```

VERTEILTE MUSTERERKENNUNG IN SENSORNETZWERKEN

» Aktivitäten *reproduce* und *diffuse*

```
28    α reproduce: {
29      live--;
30      ∇ˣ(H,$self,?);
31      if ?∇(FEATURE,?) then ∇⁻(FEATURE,n?) else n ← 0;
32      ∇⁺(FEATURE,n+1);
33      if live > 0 then
34       π*(reproduce → init)
35       ∀{nextdir∈δ | nextdir ≠ backdir ∧ ?Λ(nextdir)} do
36         Θ→(nextdir,radius)
37       π*(reproduce → exit)
38    }
39    α diffuse: {
40      live--;
41      ∇ˣ(H,$self,?);
42      if live > 0 then
43       dir ← ℜ{nextdir∈δ | nextdir ≠ backdir ∧ ?Λ(nextdir)}
44      else
45       die ← true
46    }
47    α exit: { ⊗($self) }
48
49    inbound: (nextdir) → {
50      case nextdir of
51      | NORTH →  dy > -radius
52      | SOUTH →  dy < radius
53      | WEST  →  dx > -radius
54      | EAST  →  dx < radius
55    }
56
```

VERTEILTE MUSTERERKENNUNG IN SENSORNETZWERKEN

» Signalhandler und Hauptübergangsnetzwerk

```
57    Signal handler
58    ξ TIMEOUT: {
59      enoughinput ← 0
60    }
61    ξ WAKEUP: {
62      enoughinput--;
63      if ?∇(H,$self,?) then ∇⁻(H,$self,h?);
64      if enoughinput < 1 then τ⁻(TIMEOUT);
65    }
66
67    Main Transitions
68    ∏: {
69      entry → init
70      init → percept
71      percept → reproduce | (h ≥ ε1 ∧ h ≤ ε2) ∧ (enoughinput < 1)
72      percept → diffuse   | (h < ε1 ∨ h > ε2) ∧ (enoughinput < 1)
73      reproduce → exit
74      diffuse → init | die = false
75      diffuse → exit | die = true
76    }
```

» Subklasse Kindexplorer

```
77    Explorer child subclass
78    φ child: {
79      α exit      imported from root class
80      ξ TIMEOUT
81      ξ WAKEUP
82      α percept_neighbour {
83        if not ?∇(MARK,group) then
84          back ← false; enoughinput ← 0; ∇τ(MTMO,MARK,group);  ∇%(SENSORVALUE,s?);
85          h ← (if |s-s0| ≤ DELTA then 1 else 0);
86          ∇+(H,$self,h);
87          π*(percept_neighbour → move)
88          ∀{nextdir∈δ | nextdir ≠ backdir ∧ ?Λ(nextdir) ∧ inbound(nextdir)} do
89            Θ→(nextdir,radius)
90          π*(percept_neighbour → goback | enoughinput < 1)
91          τ+(ATMO,TIMEOUT)
92      }
93      α move: {
94        backdir ← ϖ(dir); (dx,dy) ← (dx,dy) + ∂(dir);
95        ⇔dir;
96      }
97      α goback: {
98        if ?∇(H,$self,?) then ∇-(MARK,$self,h?) else h ← 0;
99        ⇔backdir;
100     }
101     α deliver: {
102       ∇-(H,$parent,v?); ∇+(H,$parent,v+h);
103       ξWAKEUP ⇒ $parent;
104     }
105     π: {
106       entry → move
107       move → percept_neighbour
108       deliver → exit
109       goback → deliver
110     }
```

VERTEILTE MUSTERERKENNUNG UND SENSORDISTRIBUTION

- Sensordistribution in verteilten Sensornetzwerken kann strombasiert oder ereignisbasiert erfolgen.

Strombasierte Sensordistribution

Es gibt einen zentralen oder mehrere dezentrale Netzwerkknoten die in periodischen Intervallen die Sensorwerte aller Knoten abfragen - unabhängig davon ob diese sich zu der letzten Anfrage geändert haben

Ereignisbasierte Sensordistribution

Die Sensorknoten liefern Sensordaten zu einem zentralen oder mehreren dezentralen Knoten wenn sich (1) Der Sensorwert geändert hat und (2) es sich um ein ausgedehntes korreliertes Ereignis handelt → Verteilte Mustererkennung

- Sensorwerte können dann per Randomwalk oder gerichteter Diffusion verteilt werden.

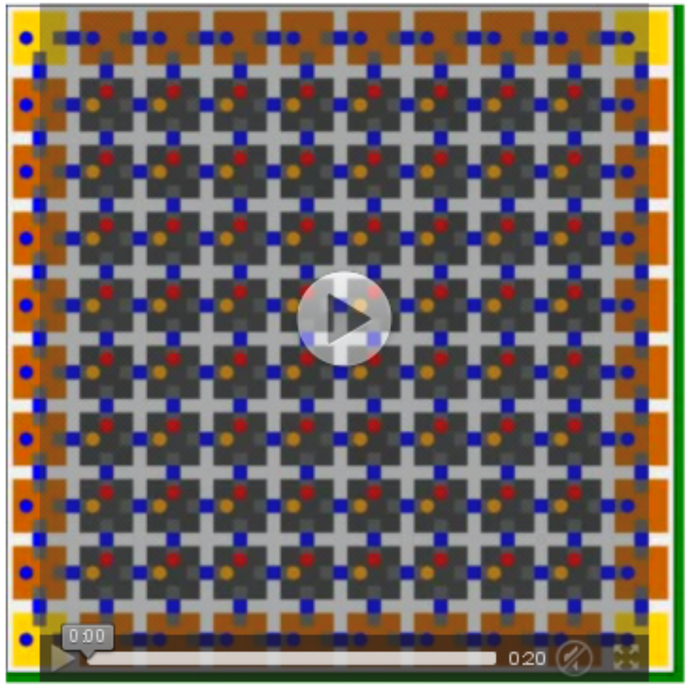
Gerichtete Diffusion

Von einem Quellknoten werden Duplikate von Verteilungsagenten in alle Richtungen ausgesendet, und an den Rändern des Netzwerks zu den Senkeknoten (Rechnern) geleitet.

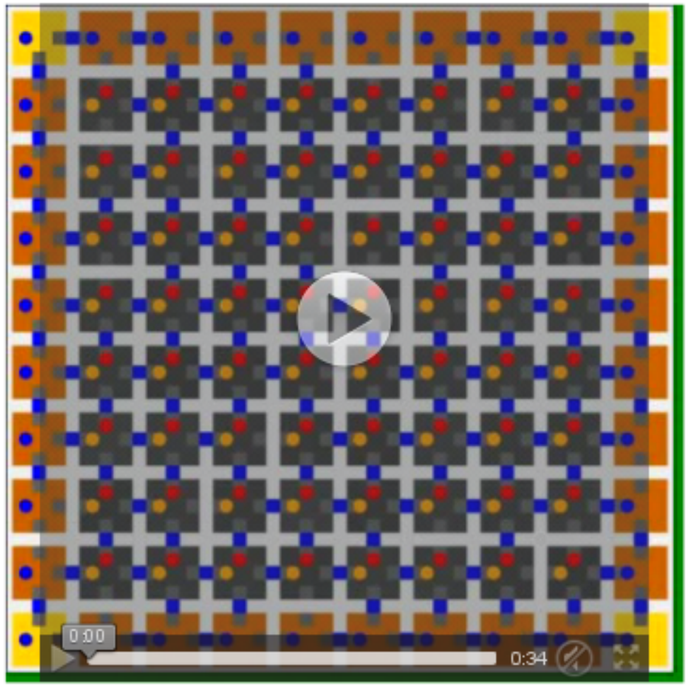
VERTEILTE MUSTERERKENNUNG UND SENSORDISTRIBUTION

Beispiel in Aktion: Positive Ereigniserkennung

Cluster mit 100% Verbindungen



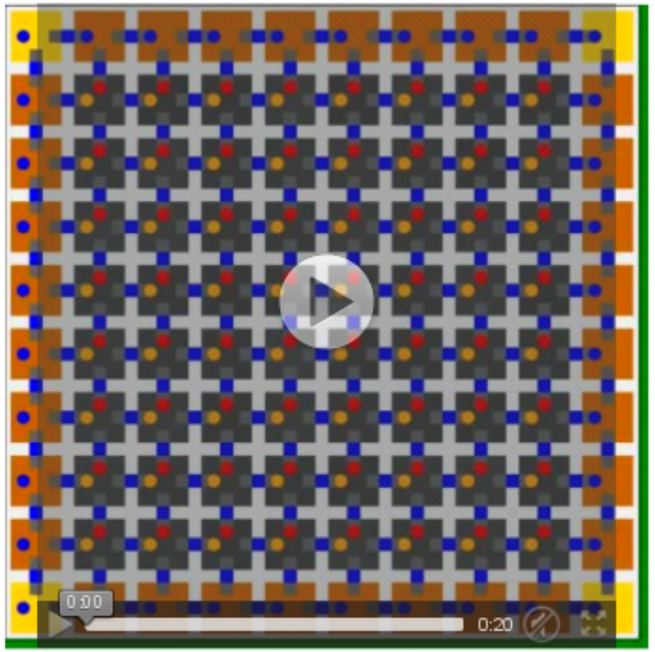
Cluster mit 60% Verbindungen



VERTEILTE MUSTERERKENNUNG UND SENSORDISTRIBUTION

Beispiel in Aktion: Gemischte Situation

Cluster und Störungen



AGENTEN UND LERNEN

MASCHINELLES LERNEN

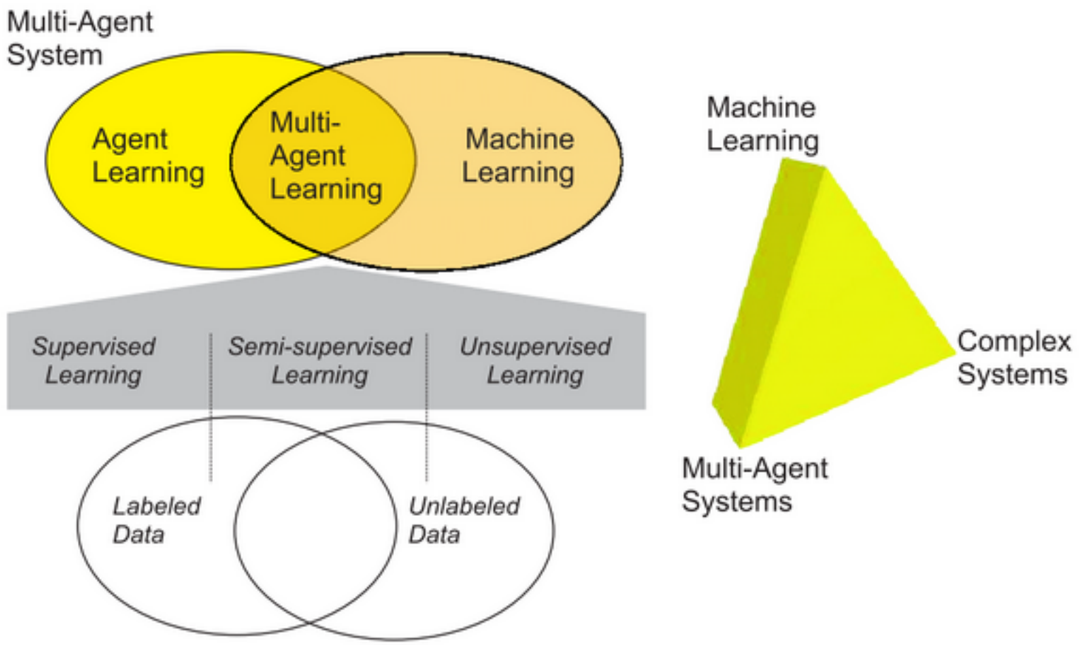


Abb. 57. Maschinelles Lernen lässt sich mit Agenten kombinieren

RÜCKGEKOPPELTES LERNEN

(Belohnungs- oder Verstärkungslernen)

- Eine sehr populäre maschinelle Lerntechnik zur Lösung von Problemen wird Verstärkungslernen genannt (Sutton and Barto, 1998), eine spezifische Art davon ist bekannt als Q-Learning (Watkins und Dayan, 1992).
- Beim Verstärkungslernen wird abgenommen, dass der Agent in einem Markov-Prozess lebt und in bestimmten Zuständen eine Belohnung erhält.
  - » Das Ziel besteht darin, in jedem Zustand die richtigen Maßnahmen zu treffen, um die zukünftige Belohnung des Agenten zu maximieren.
  - » Das heißt, die optimale Strategie/Vorgehensweise finden.
- Formal wird das Problem des Verstärkungslernens durch einen Markov-Entscheidungsprozesses (MDP) definiert, in dem die Belohnungen an den Kanten anstatt in den Zuständen angegeben werden

RÜCKGEKOPPELTES LERNEN

s	$r(s)$
s_1	0
s_2	0
s_3	1
s_4	0

s_i	a	s_j	$T(s_i, a, s_j)$
s_1	a_1	s_1	0.2
s_1	a_1	s_2	0.8
s_1	a_2	s_1	0.2
s_1	a_2	s_4	0.8
s_2	a_2	s_2	0.2
s_2	a_2	s_3	0.8
s_2	a_3	s_2	0.2
s_2	a_3	s_1	0.8
s_3	a_4	s_2	1
s_3	a_3	s_4	1
s_4	a_1	s_4	0.1
s_4	a_1	s_3	0.9
s_4	a_4	s_4	0.2
s_4	a_4	s_1	0.8

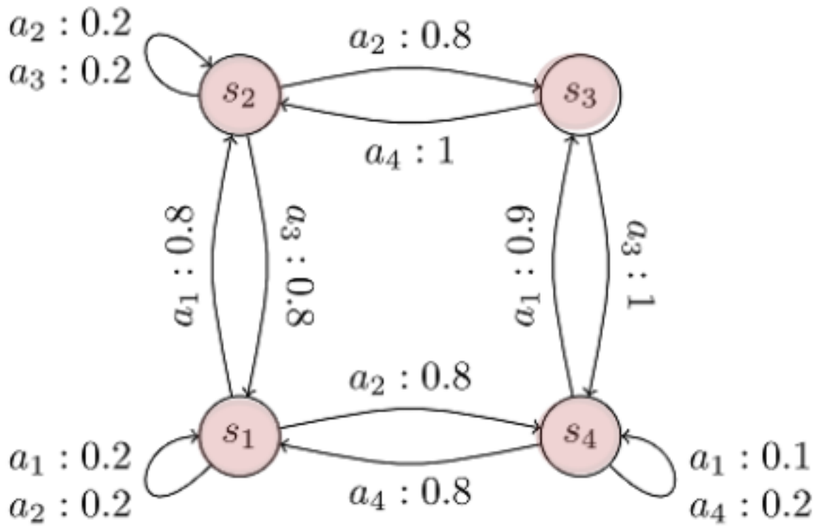
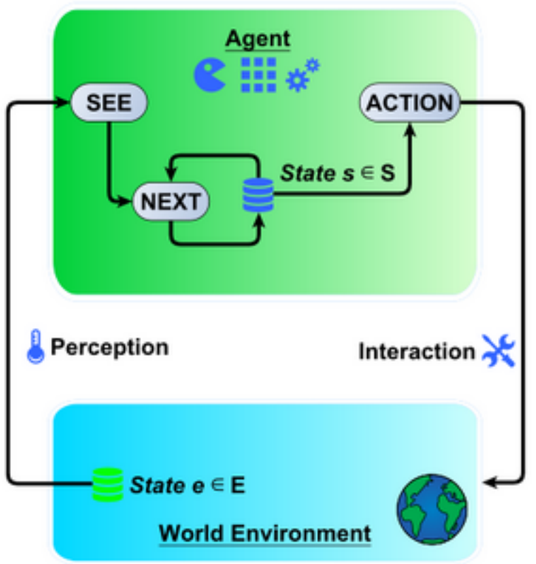


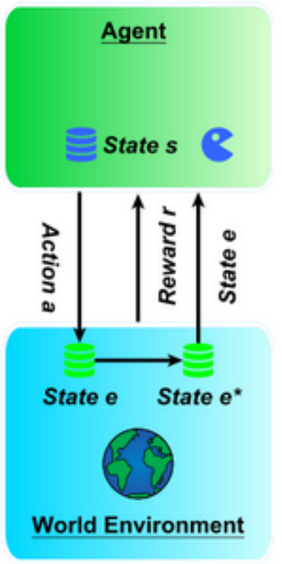
Abb. 58. Grafische Darstellung eines Markov-Entscheidungsprozesses zusammen mit Werten für die Übergangs- und Belohnungsfunktionen. Der Startzustand sei s1, und T die Übergangsfunktion $T(s_i, a, s_j)$ für den Übergang $s_i \rightarrow s_j$.

RÜCKGEKOPPELTES LERNEN

- D.h. die Belohnungsfunktion ist gegeben durch $r(s_t, a_t) \rightarrow \mathbb{R}$, mit s_t als aktuellen Zustand und a_t als Aktion.
- Es gibt eine Menge von Strategien die Zustände auf Aktionen abbilden, d.h. $\pi: S \rightarrow A$
- *Ein verstärkender Lernagent muss die Strategie $\pi^{\star} \in \pi$ finden, die seine zukünftigen Belohnungen maximiert → optimales Erreichen von Zielen!*

$$see : E \rightarrow Per$$
$$next : I \times Per \rightarrow I$$
$$action : I \rightarrow Act$$
$$reward: E \rightarrow A$$





VERTEILTES LERNEN

- Normalerweise werden Lerner zentralisiert, d.h. alle Eingabedaten werden von einem Programm gesammelt und verarbeitet.
- Diese Architektur führt zu einem einzelnen Fehlerpunkt und hohen Datenstromdichten im Netzwerk.
- Es gibt jedoch Ansätze, Lerner mithilfe von Agenten zu verteilen.
- Ein möglicher Ansatz basiert auf der Partitionierung des Lernprozesses in mehreren lokale Lerner, die auf einer räumlichen Datenuntergruppe arbeiten.
- Die lokal gelernten Modelle werden schließlich zu einem globalen Modell fusioniert.
- Dies wird erreicht, indem das (Sensor-) Netzwerk in räumlichen Regionen (Regionen von Interesse ROI) aufgeteilt wird und mehrere Lerner eingesetzt werden, wobei jeder Lerner in einer bestimmten ROI arbeitet.
- Das Lernen von Klassifikationsmodellen und deren Anwendung verwendet daher nur einen lokalen Datensatz, der auch nur eine beschränkte lokale Sicht auf die Welt bietet.

VERTEILTES LERNEN

- Aus globaler Sicht können die Ergebnisse mehrerer lokaler Klassifikationen abweichen.
- Eine geeignete Methode zur Ableitung einer zuverlässigen globalen Klassifikation (Aufbau des globalen Modells) kann durch einen Mehrheitswahl- und einem Wahlprozess umgesetzt werden.
- Jeder lokale Lernende wählt eine Klassifikationsvorhersage.
  - » Die Annahme ist, dass die Mehrheitsentscheidung das wahrscheinlichste Ergebnis liefert.
- Die verteilten Lernersysteme haben im Vergleich zum zentralen Lerner eine sehr gute Skalierungsfähigkeit, und es gibt keinen einzigen Fehlerpunkt.
  - » Defekte Knoten oder fehlende Stimmen verringern nur die globale Vorhersagegenauigkeit.

VERTEILTES LERNEN

$$
\begin{array}{ll}
M : D \rightarrow h(S) & m_{i,j} : d_{i,j} \rightarrow h_{i,j}(s) \\
l \in \mathbf{L} & h_{i,j} : s_{i,j} \rightarrow l_{i,j} \\
h : S \rightarrow l & K : (l_{1,1}, l_{1,2}, \cdots) \rightarrow l \\
D : \{(S^1, l^1), (S^2, l^2), \cdots\} \xrightarrow{\text{Distribution}} & d_{i,j} : \{(s^1_{i,j}, l^1), (s^2_{i,j}, l^2), \cdots\} \\
S : \begin{pmatrix} x_{1,1} & \cdots & x_{n,1} \\ \vdots & \ddots & \vdots \\ x_{1,m} & \cdots & x_{n,m} \end{pmatrix} & s_{i,j} : \begin{pmatrix} x_{i-u,j-v} & \cdots & x_{i+u,j-v} \\ \vdots & \ddots & \vdots \\ x_{i+u,j-v} & \cdots & x_{i+u,j+v} \end{pmatrix}
\end{array}
$$

Mit:

- *M*: Zentraler Lerner
- *D*: Globale Trainingsdatensätze
- *h*: Globales Modell
- *S*: Globale Sensordaten
- *l*: Labels (Klassenattribute)
- *k*: Lokaler Lerner
- *d*: Lokale Trainingsdatensätze
- *m*: Lokales Modell
- *s*: Lokale Sensordaten
- *K*: Globaler Aggregator

VERTEILTES LERNEN

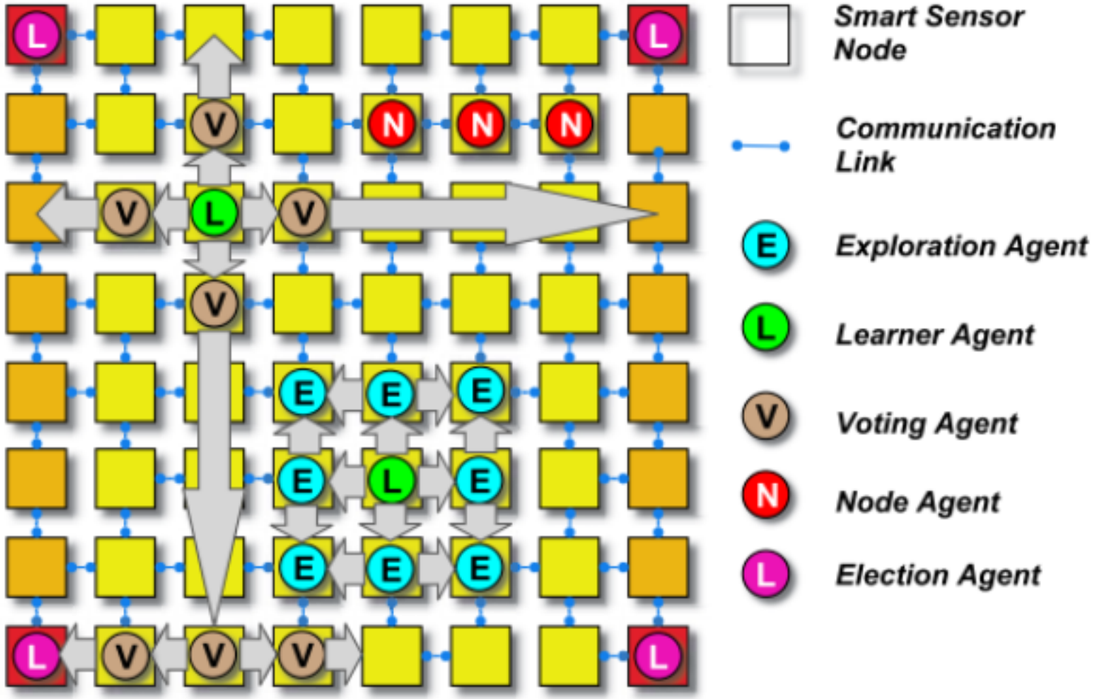


Abb. 59. Logische Netzwerkansicht und Population mit mehreren Agenten, die aus verschiedenen Verhaltensklassen instantiiert wurden.

VERTEILTES LERNEN

Multiagenten System

Knotenagent

- Jeder Sensorknoten ist mit einem nicht mobilen Knotenagenten besetzt, der Sensorakquisition, Sensorvorverarbeitung und Ereignisdetektion durchführt (d.h. einen signifikanten Stimulus erkennt).
- Ein Knotenagent kann neue Agenten für bestimmte Aufgaben instantiieren oder bereits aktive Agenten benachrichtigen.
- Der Knotenagent ist stationär (nicht mobil) und hat erweiterte Rechte (Systemrolle).

VERTEILTES LERNEN

Lerner

- Jeder Sensorknoten hat mindestens einen Lerneragenten, der vom Knotenagenten instantiiert und aktiviert wird, wenn ein Sensorstimulus erkannt wurde.
- Dieser Lerner hat zwei verschiedene Modi: (I) Lernen (II) Klassifizierung mit einem gelernten Modell.
- Die Modusauswahl wird von Benachrichtigungsagenten durchgeführt, die im Netzwerk verteilt sind, und
  I. Die Lerner über eine charakteristische Belastungssituation informieren (und ein Label *l* bereitstellen) und die Lerner veranlassen, einen Trainingsdatensatz mit einem spezifischen Label zu erstellen,
  II. Lerner umschalten in den Anwendungsmodus.
- Die Lerneragenten haben Zugriff auf Sensordaten aus der nahen Umgebung, die von Exploreragenten gesammelt und über die Tuple-Space-Datenbank (einem Plattformdienst) weitergeleitet werden.
- Die einzelnen Lerner erstellen ein lokales Sensor-Last-Vorhersagemodell.

VERTEILTES LERNEN

Explorationsagent

- Dieser Agent liefert Input für die Vorhersage eines signifikanten Sensorstimulus und für das Lernen die Sensordaten in einer räumlich eingeschränkten Region of Interest (ROI).
- Die räumliche Sensorexploration wird mit einem Divide-and-Conquer-Ansatz durch eine Gruppe von Explorationsagenten durchgeführt, die die Sensordaten sammeln und die Daten an die Knoten- oder Lerneragenten liefern.
- Jeder Explorerationsagent, der auf einem bestimmten Knoten in der ROI arbeitet, erstellt Explorationskindagenten, die Daten auf Nachbarknoten untersuchen.

VERTEILTES LERNEN

Wahl- und Abstommungsagent

- » Wenn ein Lerneragent eine Lastsituation aus seiner lokalen Sicht klassifiziert (und das lokale Modell lokale Daten verwendet), sendet er Abstimmungsagenten mit einer Vorhersage der Lastsituation.
- » Die Abstimmungsagenten übermitteln die Stimmen an Wahlagenten, die eine Mehrheitswahl für eine globale und wahrscheinlichste Vorhersage einer Lastsituation durchführen.
- ▸ Die meisten Agenten werden dynamisch von anderen Agenten erstellt, z. B. werden die Explorationsagenten von Knoten, Lernern und anderen Explorer-Agenten erstellt.
- ▸ Die Agenteninteraktion erfolgt über Tuple-Spaces (synchronisierter Datenaustausch basierend auf Patterns). Darüber hinaus werden mobile Signale zur Benachrichtigung anderer Agenten verwendet.

VERTEILTES LERNEN

Use Case: Strukturüberwachung

- Zu Beginn unbekannte äußere Kräfte, die auf eine mechanische Struktur einwirken, führen zu einer Verformung des Materials aufgrund der inneren Kräfte.
- Ein materialintegriertes aktives Sensornetzwerk bestehend aus Sensoren, Elektronik, Datenverarbeitung und Kommunikation kann zusammen mit mobilen Agenten verwendet werden, um relevante Sensoränderungen mit einem ereignisbasierten Informationsverteilundsverhalten zu überwachen.
- Inverse numerische Methoden können schließlich die Materialantwort berechnen. Die Antwort des unbekannten Systems für die extern angelegte Last *1* wird durch die Dehnungssensor-Stimulationsantwort *s'* (eine Funktion von *s*) gemessen, und schließlich berechnen die inversen numerischen Verfahren eine Approximation *1'* der angelegten Last.
- Neben komplexen numerischen Verfahren kann verteiltes Maschinelles Lernen für eine schnelle und effiziente Bestimmung bestimmter ausgewiesener Lastfälle verwendet werden.

VERTEILTES LERNEN

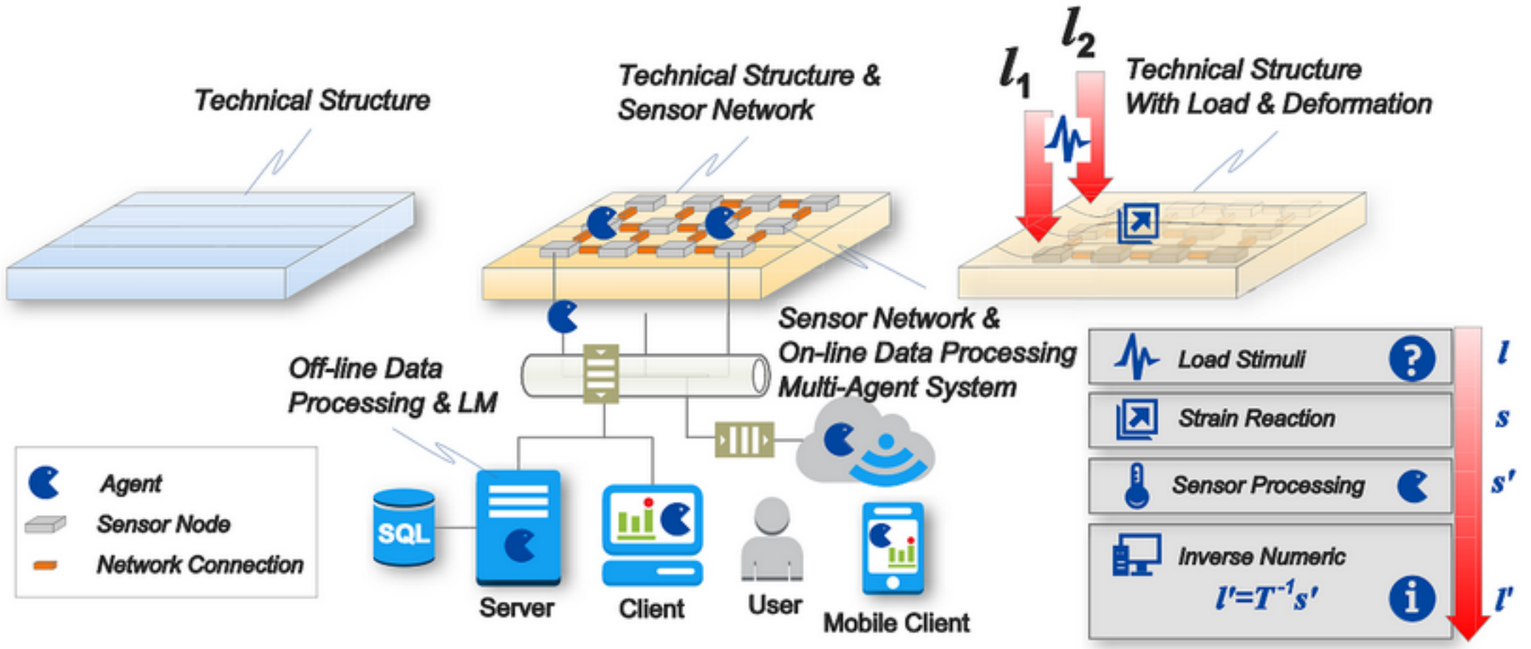


Abb. 60. Vom Material zur intelligenten überwachten Struktur mit mobilen Agenten

VERTEILTES LERNEN

Use Case: Erdbebenüberwachung und Crowd Sensing

- Verteiltes agentenbasiertes Lernen wird verwendet um aus seismischen Sensordaten auf verschiedene Erdbebenereignisse zu schließen

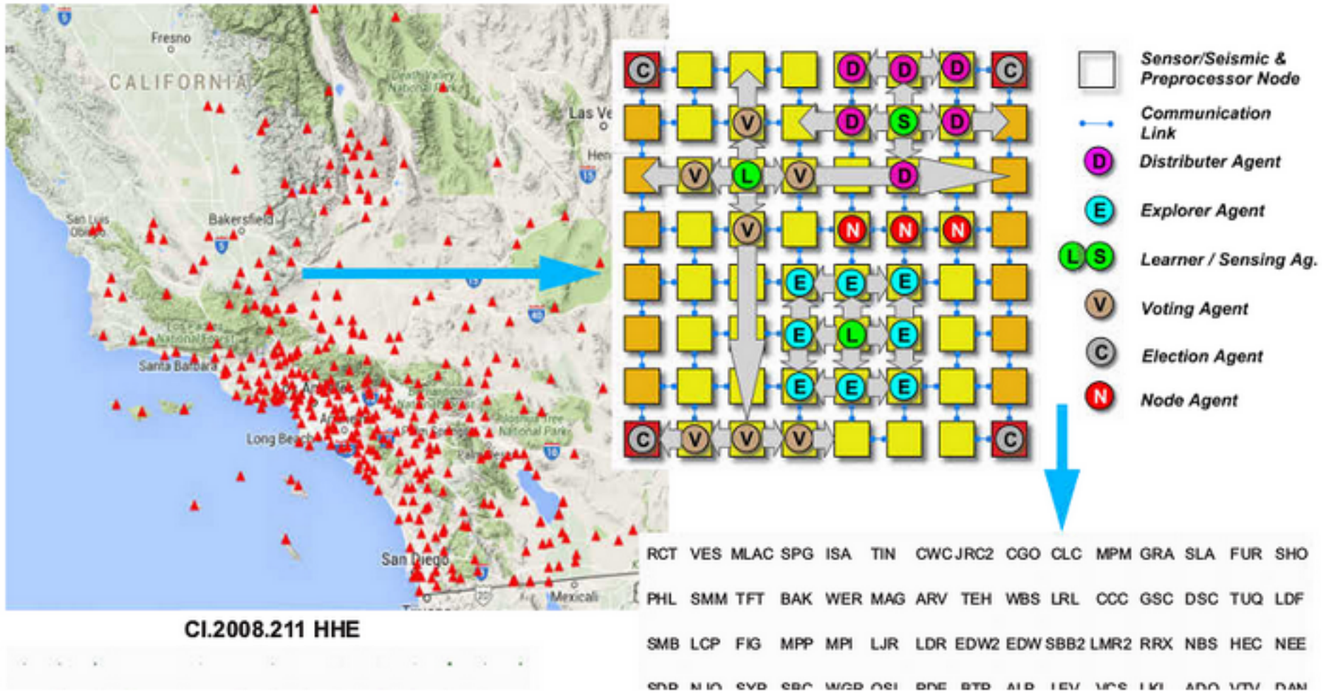


Abb. 61. (Oben, links): Das südkalifornische seismische Sensornetzwerk [Google Maps] (Oben, rechtes) Sensornetzwerk mit Stationen, die auf einer logischen zweidimensionalen Mesh-Grid-Topologie mit räumlicher Nachbarschaftsplazierung abgebildet wurden und Beispielpopulation mit verschiedenen mobilen und immobilen Agenten [1]

VERTEILTES INKREMENTELLES LERNEN

- Neben dem grob granulierten Zyklus **Lernen → Modell → Klassifikation** mit einer Datenmenge (Trainingsdaten) **D**=$\{D_1,D_2,..,D_n\}$ kann das Lernen auch inkrementell erfolgen (strombasiertes Lernen)
- Dabei wird das Modell schrittweise mit neuen Datensätzen aufgebaut → schwierig je nach Algorithmus und Modellklasse (z.B. Entscheidungs-baum)
  - » Entscheidungsbäume bestehen aus Knoten die Eingabevariablen nut-zen um die Klassifikation optimal durch Pfaditeration zu erreichen
  - » Welche Variablen optimal sind (Merkmalsselektion) wird von den Trai-ningsdaten bestimmt
  - » Sind diese nur unvollständig bekannt, kann eine nachträgliche Erwei-terung des Baumes zur Verwendung ungeeigneter (schwacher) Varia-blen führen → Schlechte Klassifikationsergebnisse sind die Folge!

Rückkopplung

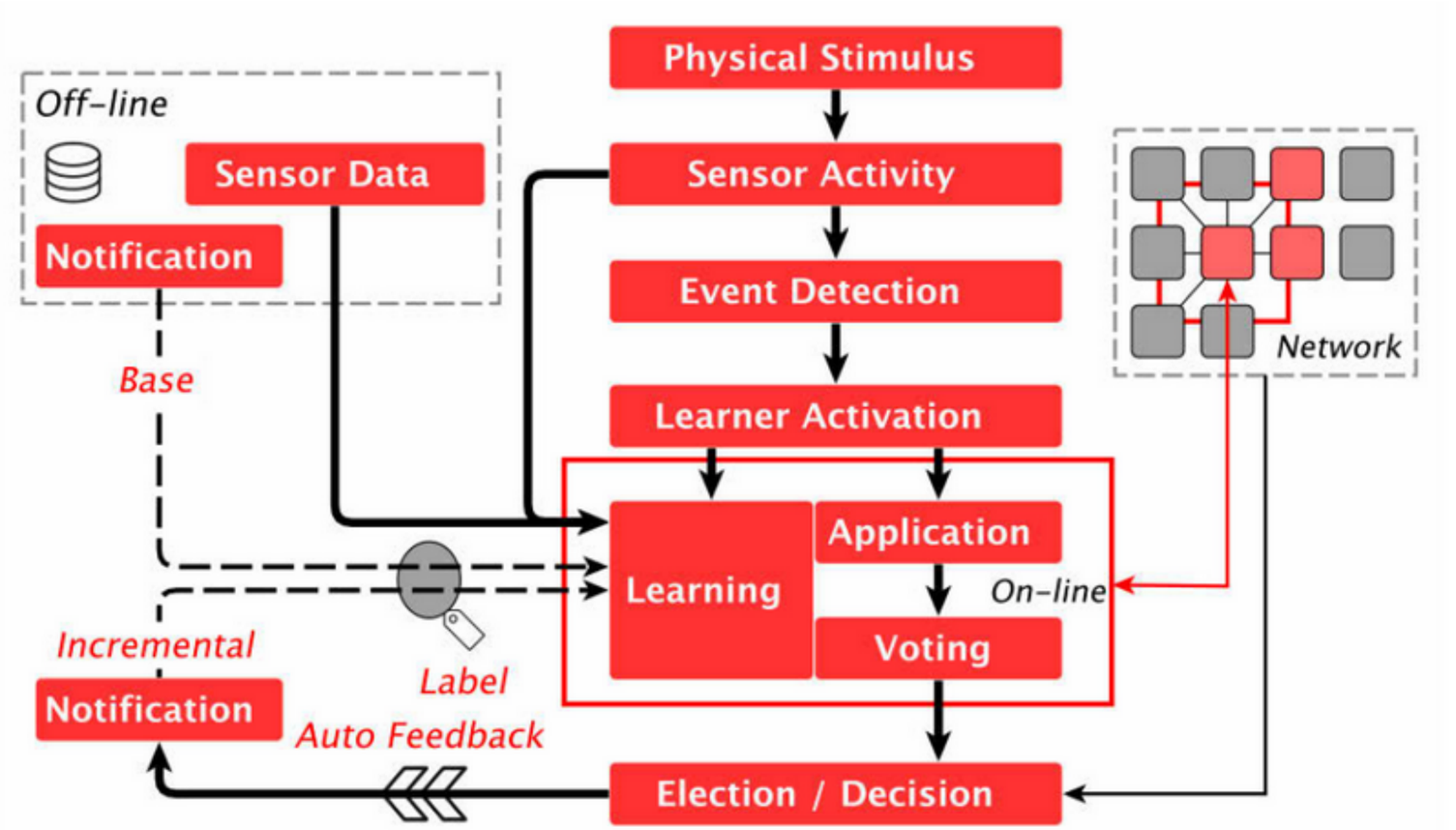


Abb. 62. Das Grundkonzept: Globales Wissen basierend auf Mehrheitsentscheidungen wird an lokale Lerninstanzen zurückgegeben, um das erlernte Modell zu aktualisieren.

REFERENCES

BOOKS

A. Jörg P. Müller, The Design of Intelligen Agents - A Layered Approach, LNAI 1177. Springer Berlin, 1996.

B. M. Wooldridge, An introduction to multiagent systems. John Wiley & Sons, Ltd, 2008/2011.

C. G. Weiss, Ed., Multiagent Sys-
tems - A Modern Ayproach to Distributed Artificial Intelligence. MIT Press, 1999.

D. H. Lin, Architectural Design of Multi-Agent Systems. IGI Global, 2007.

E. S. Ghosh, Distributed Systems - An Algorithmic Approach. Chapman & Hall/CRC, 2015.

PAPERS

1. S. Bosse, Incremental Distributed Learning with JavaScript Agents for Earthquake and Disaster Monitoring, International Journal of Distributed Systems and Technologies (IJDST), (2017), IGI-Global, Vol. 8, Issue 4, DOI: 10.4018/IJDST.2017100103
2. S. Bosse, E. Pournaras, An Ubiquitous Multi-Agent Mobile Platform for Distributed Crowd Sensing and Social Mining, FiCloud 2017: The 5th International Conference on Future Internet of Things and Cloud, Aug 21, 2017 - Aug 23, 2017, Prague, Czech Republic
3. N. J. Carriero, “Implementation of Tuple Space Machines,” 1987.
4. F. Bellifemine, A. Poggi, and G. Rimassa, Developing multi-agent systems with a FIPA-compliant agent framework, SOFTWARE—PRACTICE AND EXPERIENCE, vol. 31, pp. 103–128, 2001.
5. S. Bosse, A. Lechleiter, A hybrid approach for Structural Monitoring with self-organizing multi-agent systems and inverse numerical methods in material-embedded sensor networks, Mechatronics, (2016), DOI:10.1016/j.mechatronics.2015.08.005

VIDEOS AND WEB