Multiagentensysteme: Modelle, Programmierung, Plattformen

PD Stefan Bosse
Universität Bremen, FB Mathematik & Informatik
SS 2020
Version 2020-07-25
sbosse@uni-bremen.de

1. Inhalt

2. Überblick

2.1. Schwerpunkte in diesem Kurs

- Modellierung und Simulation mit Agenten
- 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 integrierten Übungen um obige Techniken konkret anzuwenden

Vorlesung

2 SWS mit Grundlagen und Live Programming (Baukastenprinzip!)

Übung

1 SWS mit Programmierung und angewandter Vertiefung (integriert)

Voraussetzungen

Grundlegende Programmierfähigkeiten (keine spez. Programmiersprache)

2.2. Zielgruppen des Kurses

- ➤ Soziologen
- ➤ Informatiker
- ➤ Wirtschaftswissenschaftler, Ökonomen
- ➤ Biologen
- ➤ Produktionstechniker und Logistiker
- ➤ Systemingenieure (Systems Engineering)
- ➤ Psychologen (?)
- ➤ Geologen (?)

[tiridifilm/istockphoto.com]

2.3. Materialien

1. Die Vorlesungsinhalte (Skript, Folien) werden auf **http://edu-9.de** unter der Rubrik Lehre zusammengestellt und angeboten
2. Weitere Materialien (Tutorials, Übungen, Software) werden ebenfalls auf **http://edu-9.de** bereitgestellt
3. Die Videos sind über **http://edu-9.de** verlinkt und sind auf **http://ag-0.de** verfügbar (*opencast* Server)
4. Interkation der Teilnehmer findet über einen Wiki statt! (*dokuwiki*). Dieser ist über **http://ag-0.de** erreichbar und in den jeweiligen Veranstaltungsseiten auf **http://edu-9.de** verlinkt.
5. Es wird noch einen online Chat geben.
6. Alle weiteren Hinweise und Einführungen (z.B. in Software) nur noch auf dem Wiki!!!

2.4. Methoden und Verfahren mit Agenten

Die drei großen AB*

ABM

Agentenbasiertes Modellieren →
Verhalten und Wechselwirkung von natürlichen oder technischen Systeme werden mit Agenten modelliert

ABC

Agentbasiertes Berechnen (Computing) →
Agenten sind (mobile) Software und Datenverarbeitung →
Programmier- und digitales Kommunikationsmodell

ABS

Agentenbasierte Simulation →
Simulation von komplexen Systemen mit Agentenmodellen (ABM) oder Simulation von Agenten (ABC)

2.5. Leistungen

Folgende Möglichkeiten einer Prüfungsleistung stehen zur Auswahl:

1. Mündliche Prüfung (über mindestens die Hälfte der Modulblöcke)
2. Schriftliche Ausarbeitung zu einer Fragestellung zu dem Thema (Review/Survey)

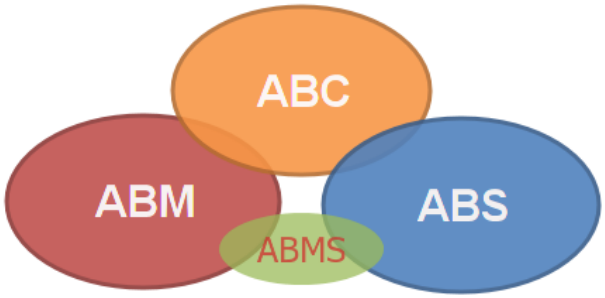
3. Die Bearbeitung einer experimentellen oder simulativen Arbeit (NetLogo/JAM)

2.6. Methoden und Verfahren mit Agenten

Variationen und Kombinationen

ABMS
Kombination und Schnittmenge aus ABM und ABS als eigene Disziplin

ABX
Kombination aus ABM, ABS, und ABC! →
Erweiterte Simulation und Modellierung mit Daten aus Crowd Sensing



Einsatzgebiete von Agenten

1. ABM und ABS werden in folgenden Disziplinen eingesetzt:
   - ➤ Soziologie
   - ➤ Biologie
   - ➤ Physik, Materialwissenschaften (Zelluläre Automaten!)
   - ➤ Ökonomie/Wirtschaftswissenschaften
2. ABC und ABS werden eingesetzt in:
   - ➤ Verteilte und parallele Datenverarbeitung, Informatik
   - ➤ Agentenbasiertes Planen, Produktionstechnik

➤ Künstliche Intelligenz, agentenbasiertes Lernen und Schwärme (Robotik)

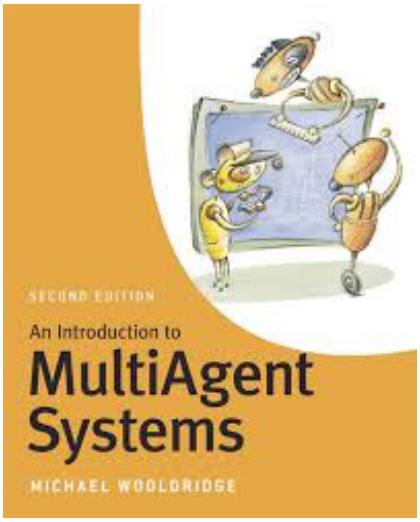
2.7. Literatur

Vorlesungsskript ABM,ABS,ABC

Die Inhalte der Vorlesung werden sukzessive bereitgestellt

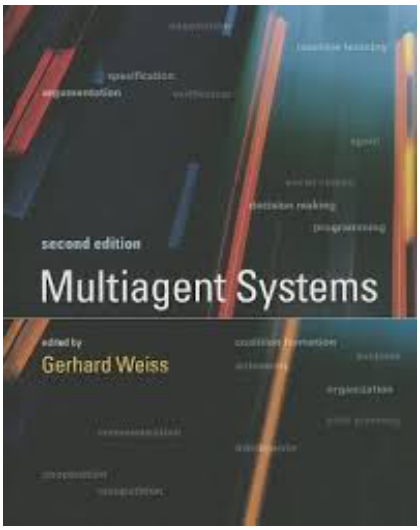
MultiAgent Systems ABC

Michael Wooldridge, John Wiley & Sons, 2002

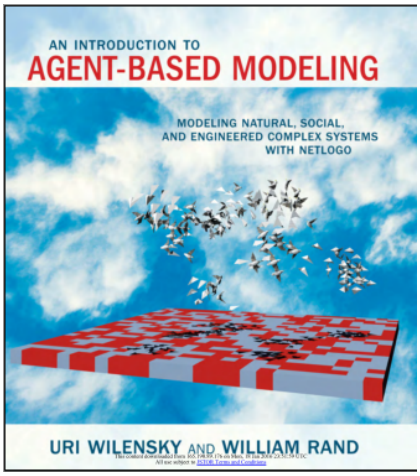


Multiagent Systems - A Modern Approach to Distributed Artificial IntelligenceABC

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



An Introduction to Agent-Based Modeling [ABM,ABS]
Uri Wilensky, William Rand, William, MIT Press, 2015



The JavaScript and nodejs Handbooks [ABC]
Flavio Copes, 2018, https://flaviocopes.com

2.8. Software

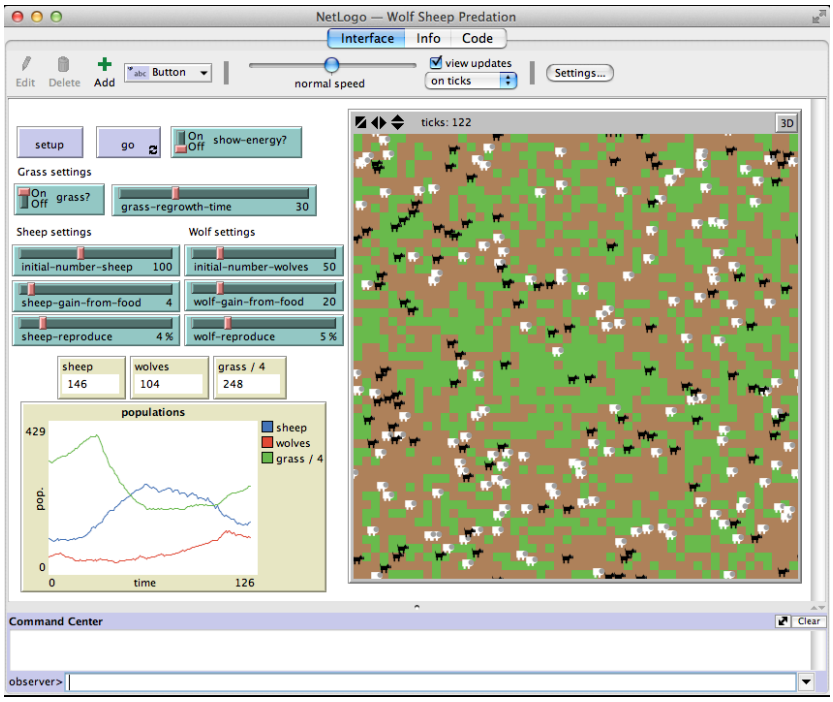
Verwendete Software (Vorlesung und Übung)

NetLogo ABM,ABS

ccl.northwestern.edu/netlogo

agentscript.org

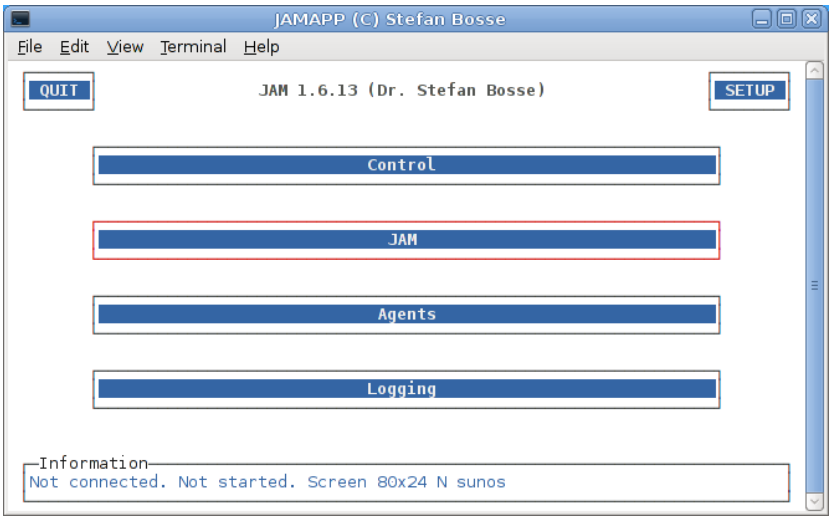
- Simulation und Evaluierung von Multiagentensystemen
- Vollständig in JAVA programmiert
- Einsatz auf verschiedenen Betriebssystemen: Windows, Unix, MacOS, ..
- Bottom-up Modellierung, aber globales Modell
- *AgentScript*: JavaScript Modellierung



JAM ABC

ag-0.de

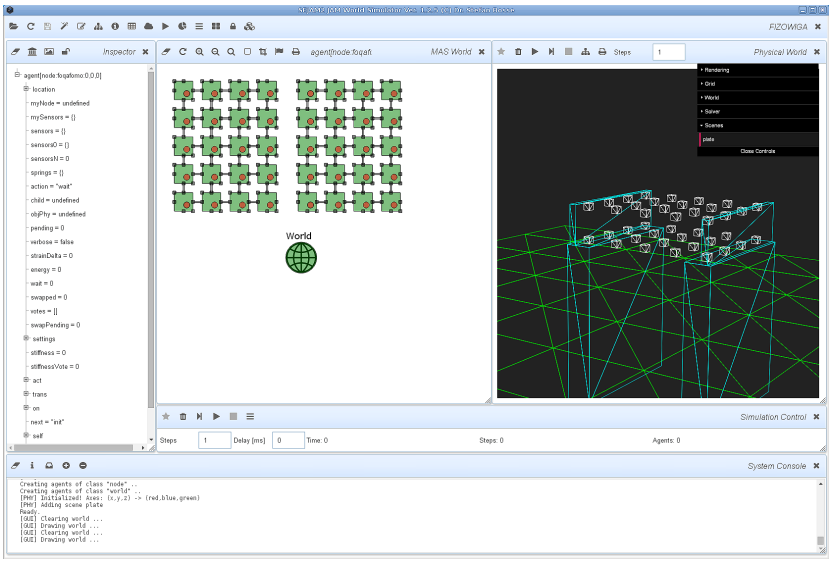
- JAM: JavaScript Agent Machine
- Vollständig in JavaScript programmiert (+Agenten: *AgentJS*)
- Einsatz auf verschiedenen Hostplattformen: PC, Smartphone, Embedded PC, Server, ..



SEJAM2 ABC,ABX

ag-0.de

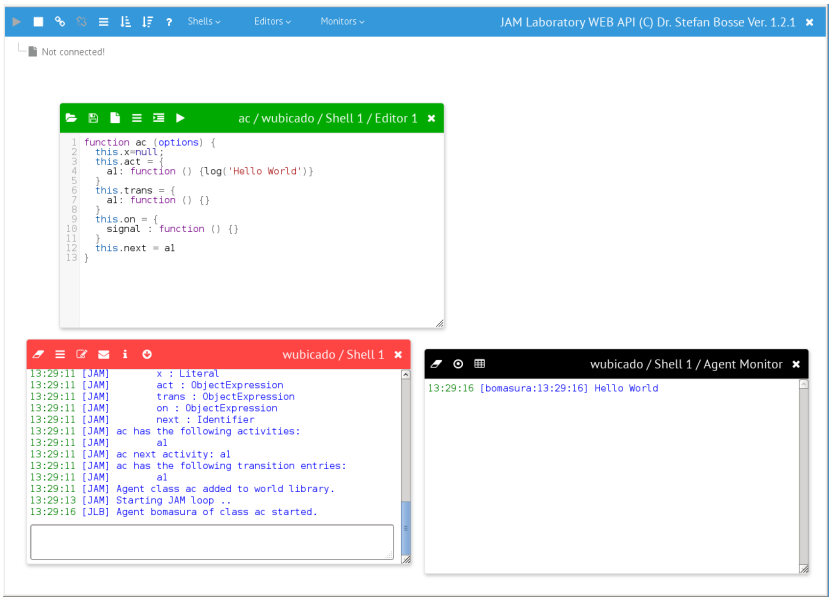
- SEJAM: Simulation Environment for JAM
- Simulationsumgebung und Entwicklungs IDE
- Bottom-up Modellierung, lokales Modell



JAM Laboratory ABC

ag-0.de

- ➤ Vollständige JAM Plattform
- ➤ IDE mit Interpreter Shell, Editor, und Nachrichtenfenstern
- ➤ Kann in jedem WEB Browser ausgeführt werden
- ➤ Internetanbindung an andere JAM Plattformen



JAM Laboratory ABC

2.9. Fragestellungen

1. Wie kann man komplexe dynamische Systeme mit Agenten modellieren und simulieren, die
   - ➤ aus einzelnen einfach Einheiten bestehen, und
   - ➤ die Einheiten miteinander wechselwirken?
2. Wie kann man das globale Verhalten → also Systemverhalten → einer Menge von Agenten studieren?
3. Wie kann robuste Datenverarbeitung in heterogenen Umgebungen (Netzwerken) mit Agenten stattfinden?

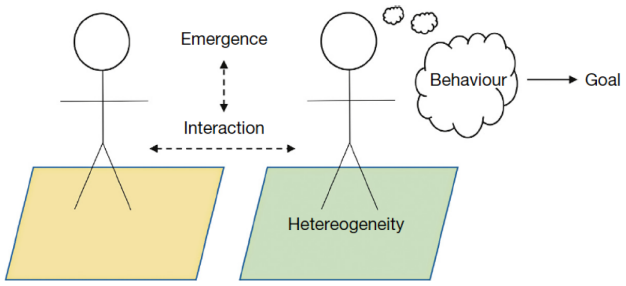


Fig. 3 Schematic illustrating of some of the main components of an agent.

[Crooks et al., 2018]

4. Wie kann in ABM/ABS die reale Welt auf künstliche Welten für Agenten abgebildet und repräsentiert werden?

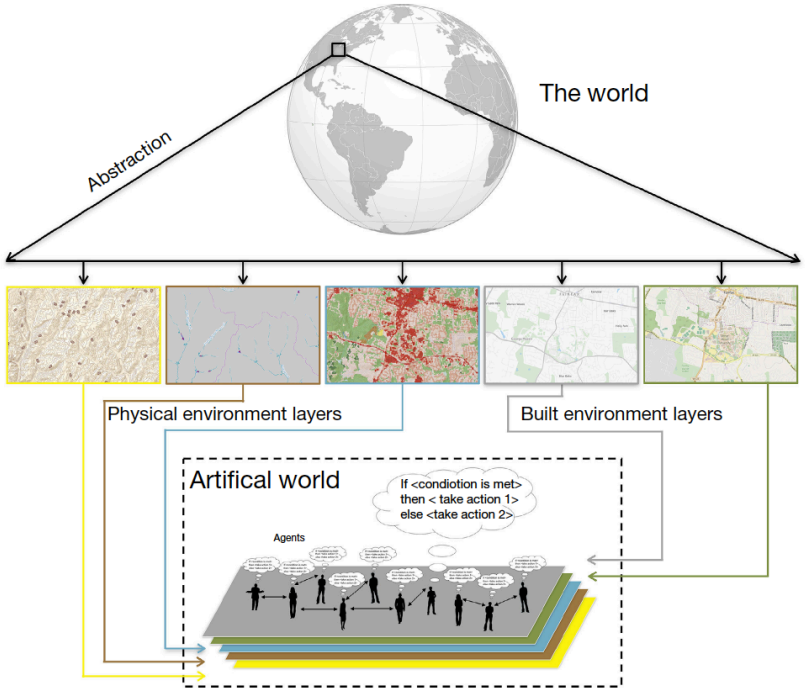


Fig. 7 Abstracting from the "real" world into a series of layers to be used in the artificial world for which to base the agent-based model upon.

[Crooks et al., 2018]

2.10. 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

2.11. 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)

2.12. Programmierung

NetLogo (deklarativ, prozedural)

```
breed [tcells tcell] ; the T cells
breed [DCs DC] ; the dendritic cells (DCs)
to setup
  clear-all
  set-default-shape turtles "circle" ; all turtles will be circles (spheres)
    create-tcells 100 [
      set color green ; make T cell green
      setxyz random-xcor random-ycor random-zcor ; put T cell at random location
    ]
    create-DCs 2 [
      set size 2 ; make DC of size 2 (twice that of T cells)
      set color red ; make DC red
      setxyz random-xcor random-ycor random-zcor
    ]
  reset-ticks
end
```

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, funktional, obj.bas.)

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;
}
```

2.13. Pressestimmen

Was sagen andere?

➤ Dr. Wander Jager, Universität Groningen

3. Einführung in die Agentenwelt

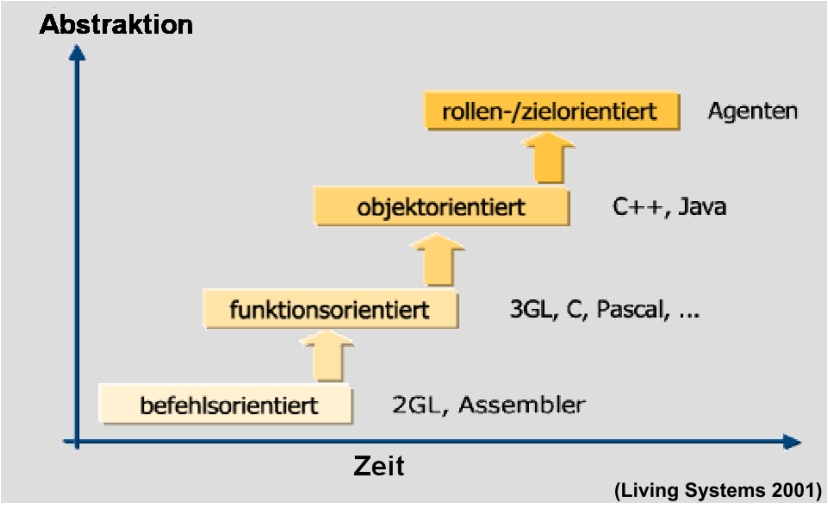
3.1. Herausforderungen

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

3.2. Software im Wandel

- Die drei Paradigmen der Programmierung: Befehlsorientiert → Funktionsorientiert → Objektorientiert
- Zukünftige Softwareentwicklung → Agentenbasiert?



3.3. Vergleich Traditionelle vs. Multiagentensysteme

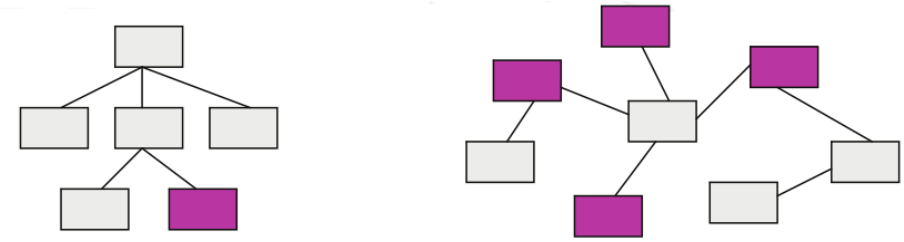
Traditionelle Systeme

- 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



3.4. Endliche Zustandsautomaten

- Das Gegenteil von Agenten?
- Ein endlicher Zustandsautomat (EZA/englisch FSM) verarbeit Eingangsdaten (Variablen x) und gibt Ergebnisse y in Abhängigkeit seines Zustandes $\sigma \in \Sigma$ aus
- Die Menge der Zustände Σ ist von Beginn an festgelegt und konstant!
- Keine Adaptivität, keine Rekonfiguration!
- Einfachstes Modell einer Berechnung (im Sinne der Informatik)
- Folgendes Beispiel zeigt einen EZA der eine Zahl auf Teilbarkeit durch 3 unteruschen soll
- Dabei wird die Zahl im binären Zahlensystem Ziffer für Ziffer links nach rechts in den Automaten gegeben
- Dieser ändert bei jeder neuen Eingabe (Sensor) seinen Zustand σ ∈ Σ={0,1,2}

-

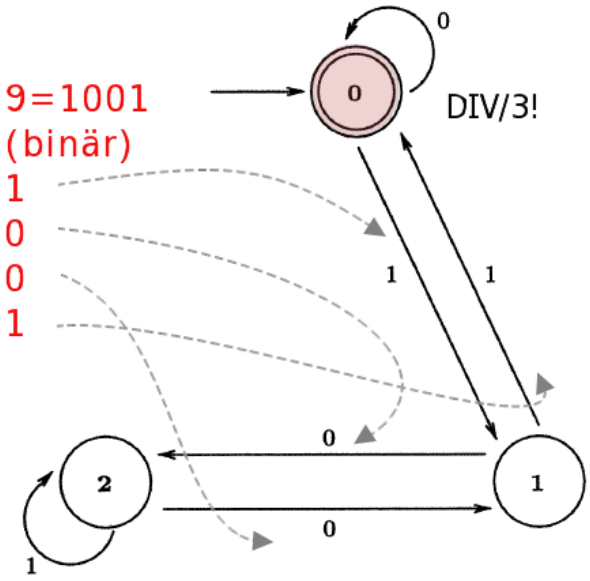
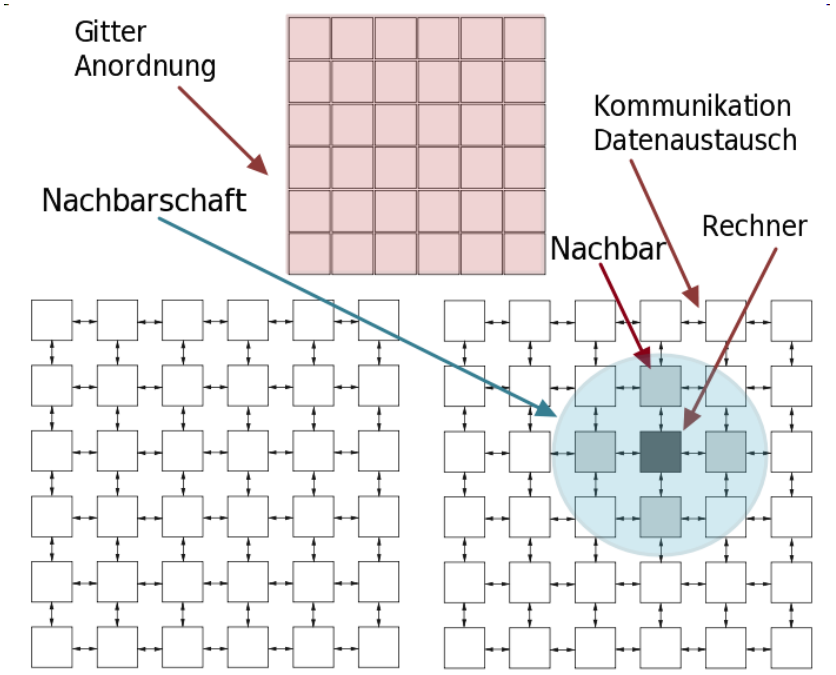


Abb. 1. Beispiel eines EZA

3.5. Zelluläre Automaten

- Die Ursuppe für Agenten?
- Zelluläre Automaten besitzen diskrete Zustände und ändern ihren Zustand nur zu diskreten Zeitpunkten

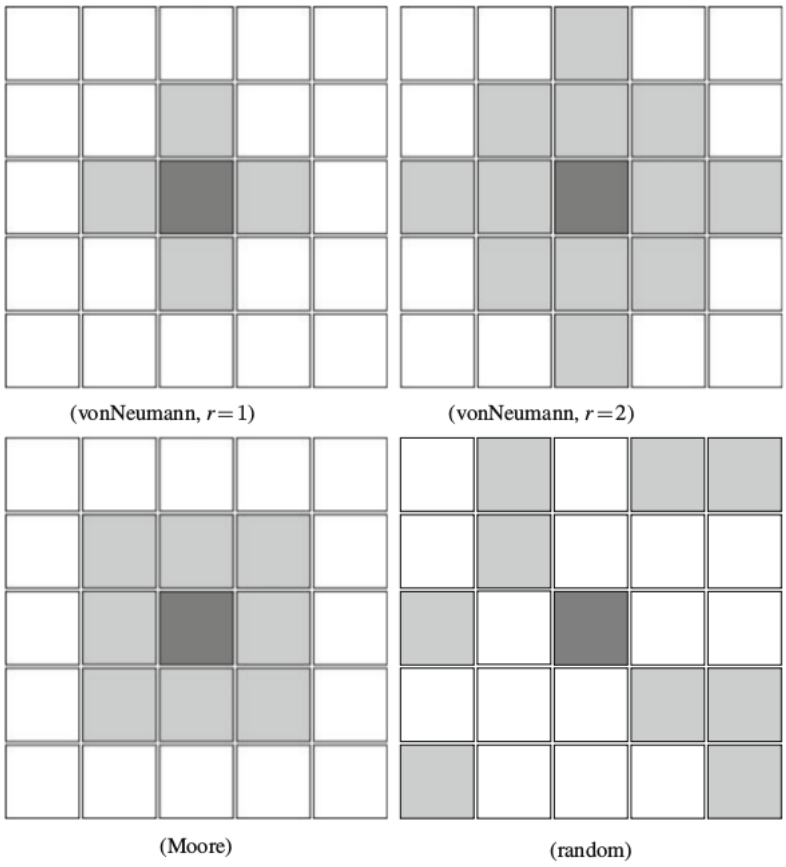


[SCSCA,2010]

Abb. 2. Zelluläre Automaten als Netzwerk aus einfachen kommunizierenden Berechnungseinheiten

- Es gibt immerhin schon Nachbarschaftskommunikation (Datenaustausch) zwischen Knoten des Netzwerks
- Aber: Ein Automat ist noch ein (nicht adaptiver) endlicher Zustandsautomat → ZA = Lokal verbundende EZA!
- Auch die Netzwerkkonfiguration ist statisch - entspricht nicht natürlichen Systemen
- Nachbarschaftsrelationen können durch einen Radius charakterisiert werden:
  - Radius 1: Nur unmittelbare Nachbarn → **von-Neuman Nachbarschaft**
  - Radius ≤ 2: kurzreichweitige Verbindungen (einfache Systeme)
  - Radius > 2: langreichweitige verbindungen (komplexe Interaktion und Dynamik)
- **Kommunikation kann als Austausch von Signalen** zwischen einzelnen Zellen verstanden werden

- Zelluläre Automaten besitzen räumliche und zeitliche Dynamik
- Nachbarschaftsrelationen sind auch bei Agenten wichtige Eigenschaft



[Hoekstra,SCSCA,2010]

Abb. 3. Verschiedene Nachbarschaftsrelationen

- Eine Zelle besitzt eine endliche (kleine) Menge von Zuständen $\sigma = \{ s_1,..,s_n \}$.
- Eine Berechnung mit Eingabe- und internen Daten (Perzeption und innerer Zustand) führt i.A. zu einer Änderung des Zustandes der Zelle → es gibt einen Zustandsübergang
- Übergangsregeln Φ können aus einfachen arithmetischen Operationen (Funktionen) bestehen, und beziehen die Zellen aus der Nachbarschaft N mit ein (durch Kommunikation):

$$\sigma_{i,j}(t+1) = \Phi(\sigma_{k,l}(t) | \sigma_{k,l}(t) \in N)$$

➤ Die Nachbarschaftskonnektivität und die Anzahl der Zustände je Zelle bestimmen die Anzahl der möglichen lokalen Regeln die zu einem Zustandsübergang in der Nachbarschaftsgruppe führen → wird sehr schnell sehr groß!!

Number of states σ	Number of neighbors n	σ^{σ^n}	Number of rules N_r
2	2	2^{2^2}	16
2	3	2^{2^3}	256
2	5	2^{2^5}	4 294 967 296
2	10	$2^{2^{10}}$	$1.797 \cdot 10^{308}$
5	2	5^{5^2}	$2.98 \cdot 10^{17}$
5	3	5^{5^3}	$2.35 \cdot 10^{87}$
5	5	5^{5^5}	$1.91 \cdot 10^{2184}$
10	2	10^{10^2}	10^{100}
10	3	10^{10^3}	10^{1000}
10	5	10^{10^5}	10^{100000}

[Hoekstra,SCSCA,2010]

Abb. 4. Anzahl der Regeln eines ZA in Abhängigkeit der Anzahl der Zustände pro Zelle und des verbinsungsgrades der Zellen

Beispiele für Dynamik und Emergenz

```
var world = new CAWorld({
  width: 96,height: 64, cellSize: 6
});
world.palette = [
  '255, 255, 255, 1', '68, 36, 52, 1'
];
world.registerCellType('wall', {
    getColor: function () {
        return this.open ? 0 : 1;
    },
    process: function (neighbors) {
        var surrounding = this.countSurroundingCellsWithValue(neighbors, 'wasOpen');
        this.open = (this.wasOpen && surrounding >= 4) || surrounding >= 6;
    },
    reset: function () {
        this.wasOpen = this.open;
    }
}, function () { /* init cells */ this.open = Math.random() > 0.40; });
world.initialize([ { name: 'wall', distribution: 100 } ]);
```

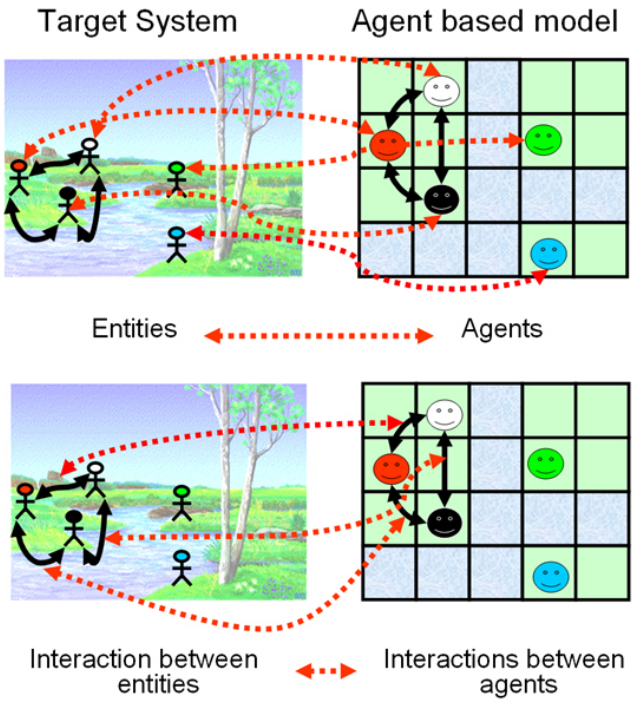
3.6. Agenten

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 Instanz 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**)

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



[Galan,2009]

Abb. 5. Beziehung realer natürlicher Welt mit Lebewesen zu virtueller Welt mit Agenten

3.7. Multiagentensysteme - 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 globalen Aufgabe zu erfüllen,
  - ❏ auch wenn manche dieser Agenten gegensätzliche Interessen haben und ihre eigenen (konkurrierenden) Ziele verfolgen?

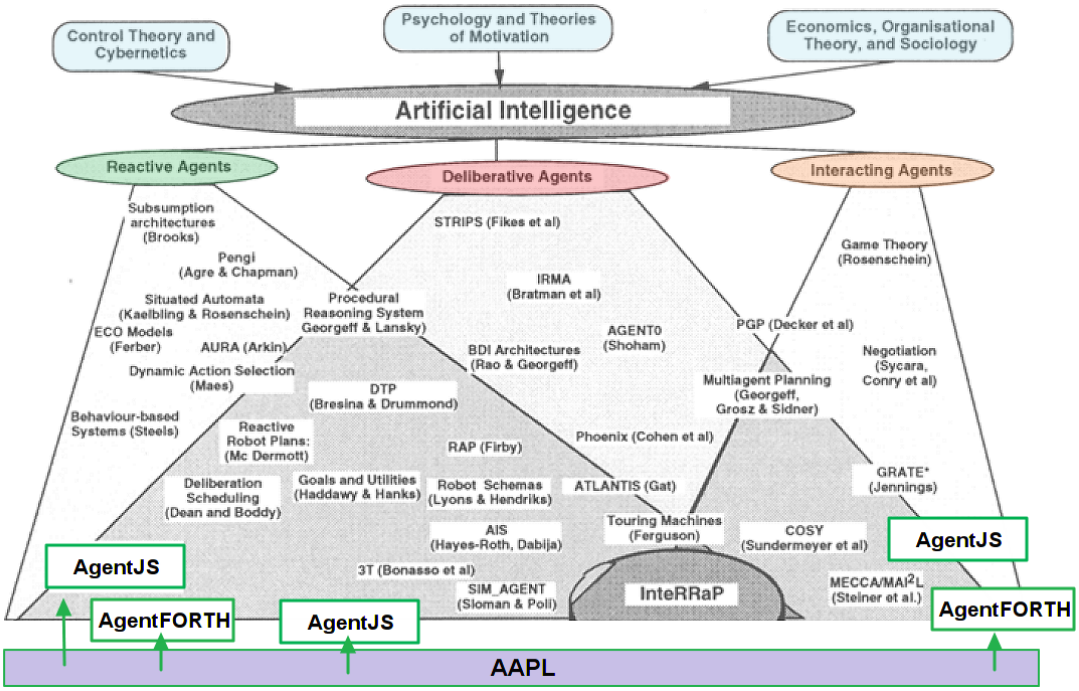


Abb. 6. Entwurf von Intelligenten Agenten: Eine Roadmap [A] und AAPL als Basis

3.8. Multiagentensysteme - Taxonomie

Reaktiver Agent

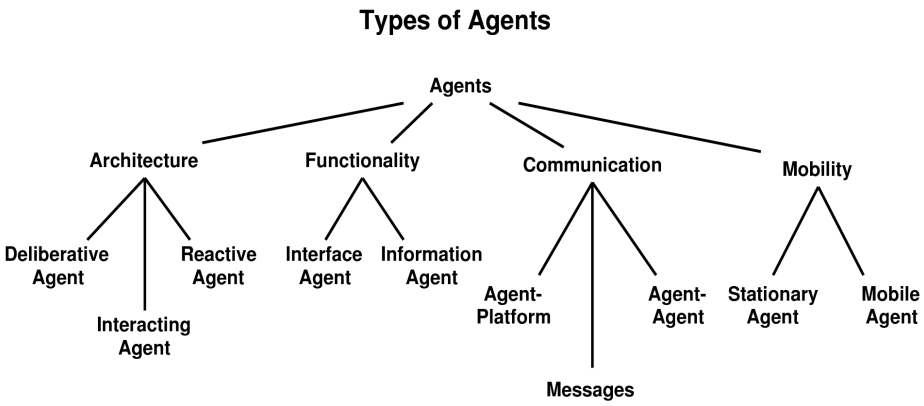
Ein Agent der seine Umgebung wahrnimmt und zeitnah darauf reagiert

Deliberativer Agent

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

Interagierende Agenten

Multiagentensysteme mit kommunizierenden Agenten



3.9. Multiagentensysteme - 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?

➤ **Ontologien**: 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 verhandeln 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?

3.10. Multiagentensysteme - 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???

3.11. Multiagentensysteme - 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**.

3.12. Multiagentensysteme - 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 Ontologie 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.

3.13. Multiagentensysteme - Wofür?

Sichten auf Multi-Agenten Systeme

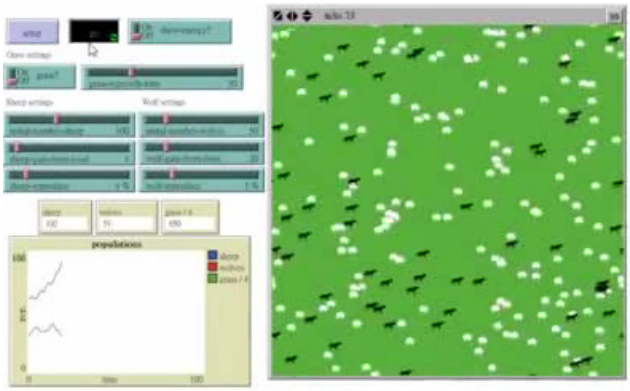
- Agenten als **Softwareentwicklungsparadigma** (agentenorientierte Programmierung)
  - Moderne Softwarearchitekturen setzen sich aus vielen, dynamisch interagierenden Komponenten zusammen
  - Weiterentwicklung der Konzepte der Modularisierung und Objektorientierung
- Agenten als Mittel zur **Modellierung**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

3.14. Multiagentensysteme - Dafür!

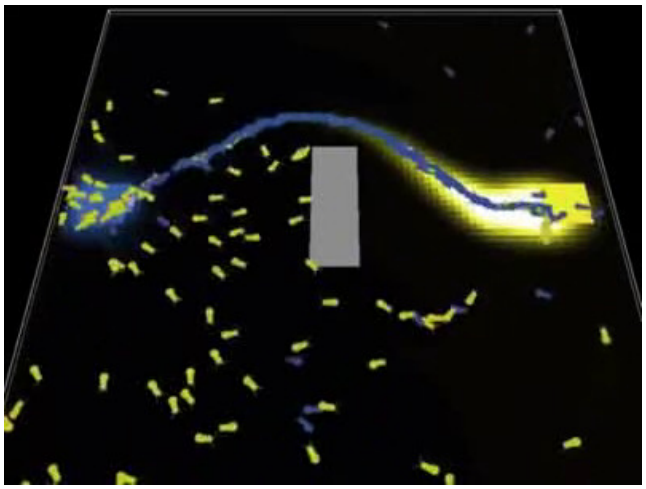
Simulation komplexer Welten

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



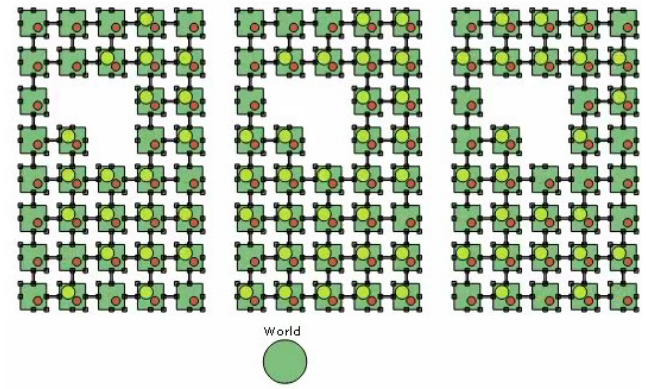
Modellierung komplexer Welten

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

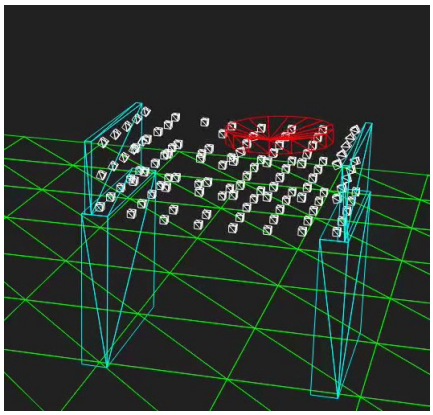


MAS Welt

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



Physikalische Welt



3.15. Multiagentensysteme - Klassifizierung

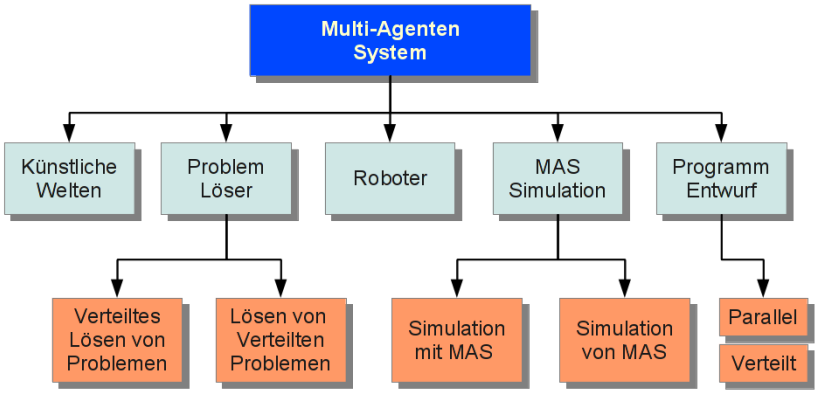


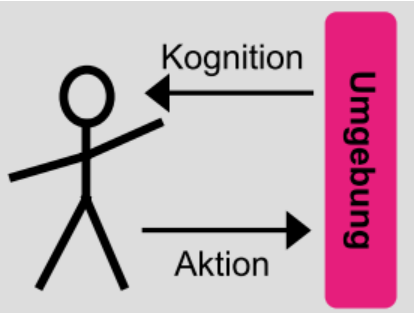
Abb. 7. Klassifizierung und verschiedene Typen der Anwendung von MAS

3.16. Multiagentensysteme - 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

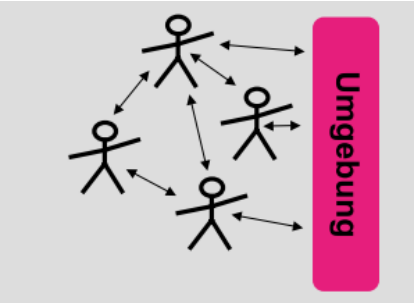
Kommunikation

- ➤ Mit Menschen
- ➤ Mit anderen Agenten

Umgebung • Kognition • Aktion [Wooldrigde 2000]

Multiagentensysteme

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



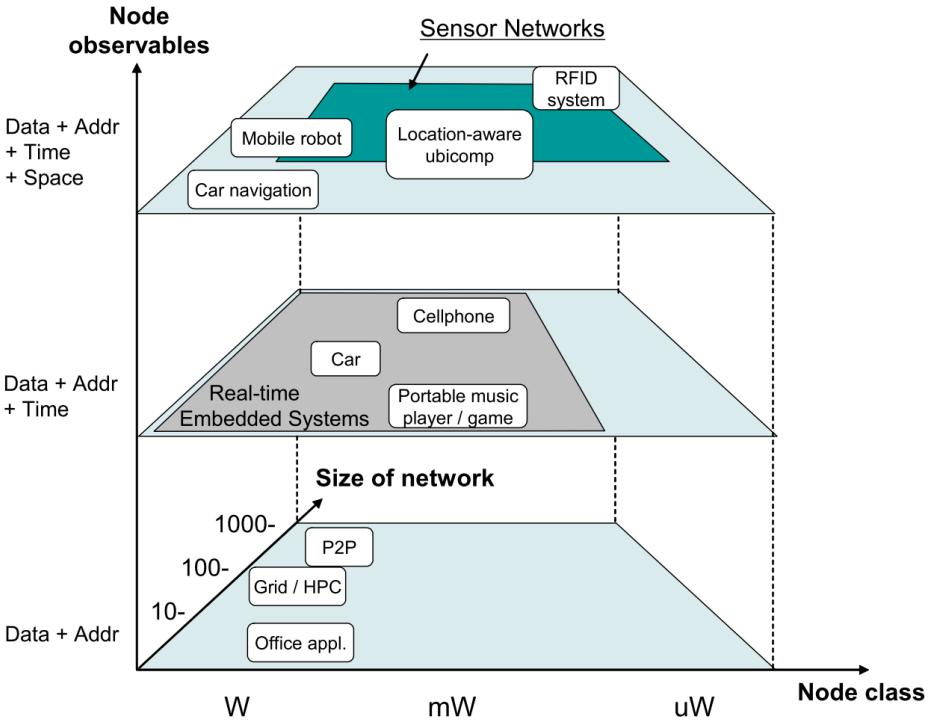
3.17. Netzwerke

Klassifikation von Netzwerken

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

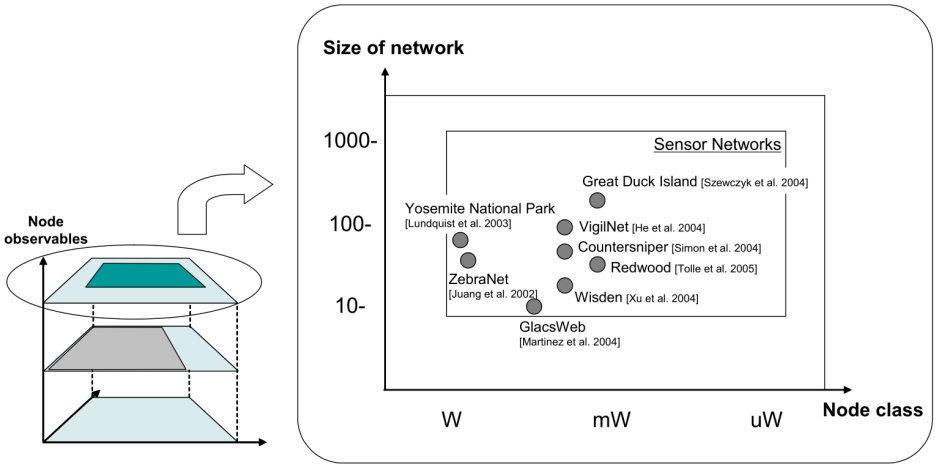
Relevante Observablen von Netzwerkknoten:

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



[Sugihara, 2008]

Sensornetzwerke



[Sugihara, 2008]

Abb. 8. Beispiele von Sensornetzwerken nach Anzahl der Knoten / Knoten Klasse

3.18. 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.

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

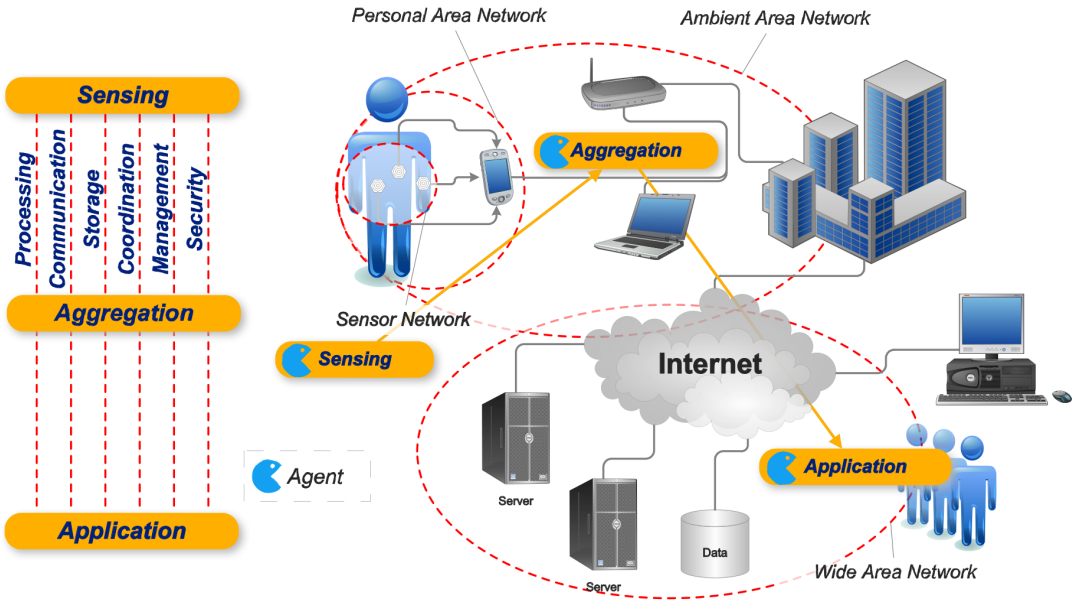
➤ Speicher

➤ Koordination

➤ Management

➤ Sicherheit

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



4. Anwendungsbeispiele, Plattformen, Toolkits

4.1. Entwicklungswerkzeuge

- ➤ Domainen: ABC, ABM (Planung und Koordination)
- ➤ Weit verbreitete Programmiersprache: JAVA

Name	Referenz
AGlobe	Šišlák et al. (2005)
FIPA-OS	Poslad et al. (2000)
JACK	Howden et al. (2001)
JADE	Bellifemine et al. (2005), Bellifemine et al. (2007)
JADEX	Braubach et al. (2005), Jadex (2014)
JIAC	Hirsch et al. (2009), Lützenberger et al. (2013)
Living Systems Technology Suite	Rimassa et al. (2005)
MAdkit	Gutknecht and Ferber (2000)
Multi-Agent System Development Kit	Gorodetsky et al. (2005)
Repast Simphony	North et al. (2013), Repast-S (2014)

Entwicklungswerkzeuge für MAS [Leitao, IAEASAI, Kap. 1, 2015]

4.2. Programmiersprachen

Agentenorientierte Programmiersprachen (AOP)

- ➤ Unterteilung in zwei wesentliche Klassen:
  - ❏ Inherentes Agentenmodell (I) und Kommunikation
  - ❏ Als Erweiterung (E) zu bestehenden Programmiersprachen
- ➤ Weiterhin kann eine AOP mit einer Architektur verknüpft sein (A)

Name	Klasse	Referenz
Agent0	I/A	Shoham (1991)
2APL/3APL	I	Dastani et al. (2005)
AgentSpeak(L)/Jason	E	Kinny et al. (1996), Rao (1996), Bordini et al. (2007)
ASPECS	?	Cossentino et al. (2007)
GOAL	?	Hindriks et al. (2012) GOAL (2011)
Golog	?	Levesque et al. (1997)
MetateM	?	Dennis et al. (2008)
PLACA	?	Thomas (1995)
AgentJS	E/I	Bosse (2014)

Agentenorientierte PL [Leitao, IAEASAI, Kap. 1, 2015]

4.3. Simulationswerkzeuge

Name	APL/Modell	Refe
Agent.GUI	?	Derksen et al. (2011)
AMASON	?	Klügl and Davidsson
MASON	?	Luke et al. (2005), M
MAST	?	Vrba et al. (2008)
NetLogo	Text/Prozedural, Turtle	Wilensky and Rand (
Repast for High Performance Computing (Repast HPC)	?	RepastHPC (2014)
SeSAm	Graph, ATG/ABM	Klügl (2009)
SEJAM2	Text/JS, ATG/ABC/ABM	Bosse (2016)

Agentenbasierte Simulationswerkzeuge [Leitao, IAEASAI, Kap. 1, 2015]

4.4. Modellierung

➤ Agentenbasiertes Modellieren wird vielfältig eingesetzt (eine Auswahl):

Application Area	Model Description
Air Traffic Control	Agent-based model of air traffic control to analyze control policies and performance of an a fic management facility (Conway 2006)
Anthropology	Agent-based model of prehistoric settlement patterns and political consolidation in the Lake caca basin of Peru and Bolivia (Griffin and Stanish 2007)
Biomedical Research	*The Basic Immune Simulator*, an agent-based model to study the interactions between innat adaptive immunity (Folcik, An and Orosz 2007)
Chemistry	An agent-based approach to modeling molecular self-assembly (Troisi, Wong and Ratner 2
Crime Analysis	Agent-based model that uses a realistic virtual urban environment, populated with virtual bu agents (Malleson 2009).
Ecology	Agent-based model of predator-prey relationships between transient killer whales and other rine mammals (Mock and Testa 2007).
Energy Analysis	Agent-based model for scenario development of offshore wind energy (Mast et al. 2007).
Epidemic Modeling	*BioWar*, a scalable citywide multi-agent model, that simulates individuals embedded in soci health, and professional networks and tracks the incidence of background and maliciously i duced diseases (Carley et al. 2006).
Market Analysis	Agent-based simulation that models the possibilities for a future market in sub-orbital space ism (Charania et al. 2006).
Organizational Decision Making	Agent based modeling approach to allow negotiations in order to achieve a global objective cifically for planning the location of intermodal freight hubs (van Dam et al. 2007).

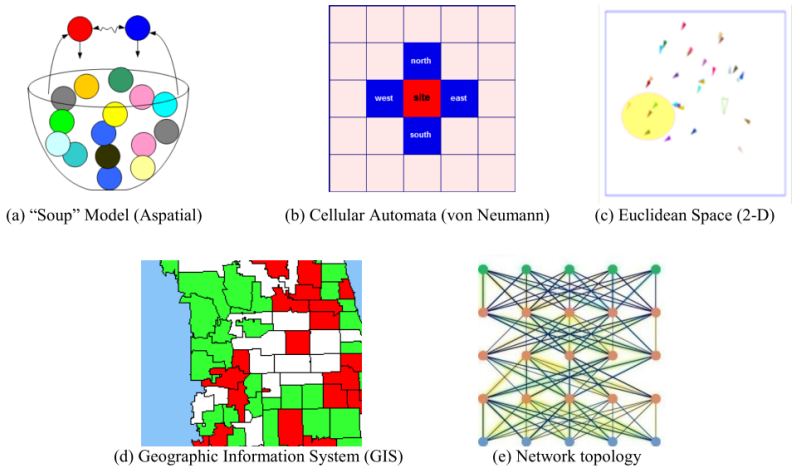
[Macal, WSC, 2009]

Topologien

➤ Die agentenbasierte Modellierung befasst sich ebenso mit der Modellierung von Agentenbeziehungen und Agenteninteraktionen wie mit der Modellierung von Agenten und dem Verhalten von Agenten.

➤ Die Hauptprobleme bei der Modellierung von Agenteninteraktionen sind die Angabe, wer mit wem verbunden ist oder sein könnte, und die Dynamik, die die Mechanismen der Interaktionen steuert.

➤ Beispiel: Ein agentenbasiertes Modell des Internetwachstums würde beispielsweise Mechanismen enthalten, die angeben, wer mit wem, warum und wann eine Verbindung herstellt.



(a) "Soup" Model (Aspatial) (b) Cellular Automata (von Neumann) (c) Euclidean Space (2-D)

(d) Geographic Information System (GIS) (e) Network topology

Abb. 9. Topologien für Agentenbeziehungen und Soziale Interaktion [Macal, WSC, 2009]

➤ Die Mikromodellierung mit Agenten umfasst besonders deren Interaktion untereinander und mit der Weltumgebung (Maschinen, Menschen, usw.)

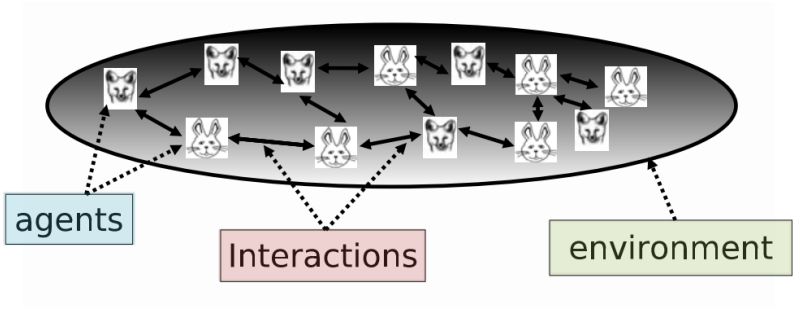
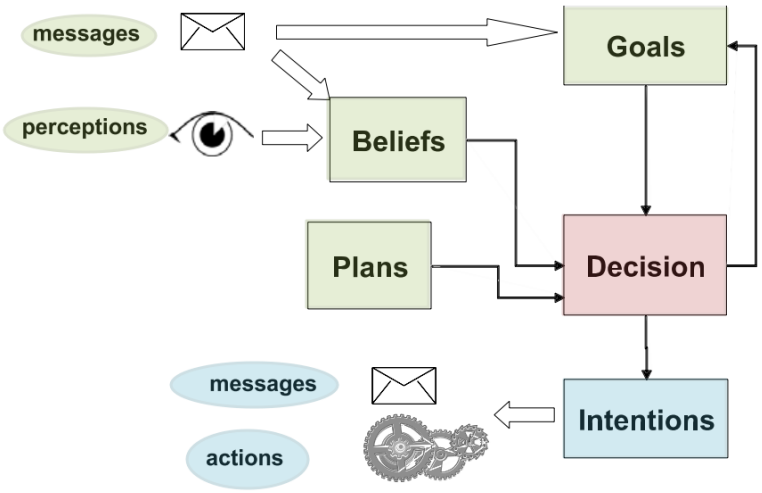


Abb. 10. Das agentenbasierte Modell mit Entitäten auf Mikroebene, ihre Aktionen und Interaktionen sowie die Umgebung [Uhrmacher ed., MASSA, 2009]

Agentenmodell

➤ Zutaten: Perzeption, Kommunikation, Wissen (Beliefs), Regeln (Plan, Goals), Ausführen (Intentions) → BDI Architektur!



[Uhrmacher ed., MASSA, 2009]

4.5. Produktion und Logistik

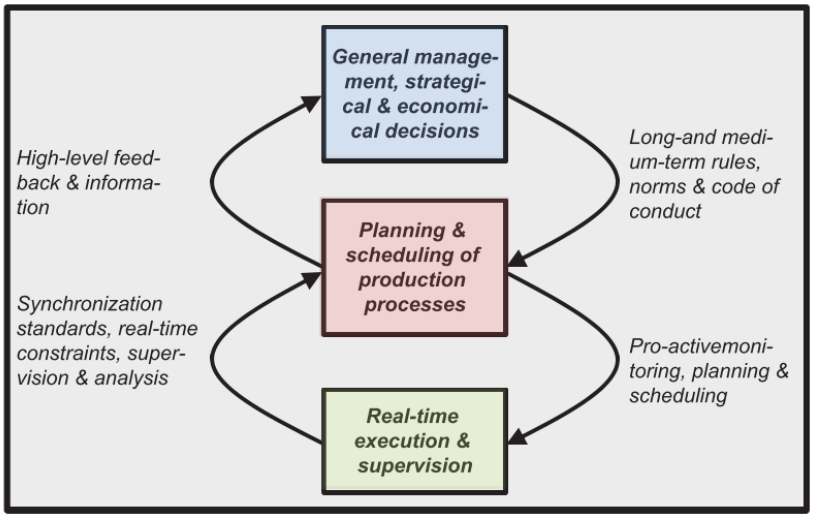
- In der **Produktion** und **Logistik** werden ABM/ABC/ABS für die Planung, Entscheidungsfindung, und Steuerung von Produktionsanlagen und logistischen Systemen eingesetzt
- Dabei können Agenten verschiedene Entitäten in Produktion und Logistik repräsentieren:
  - Materialien und Waren
  - Transportträger (Fahrzeuge usw.)
  - Warenlager
  - Menschen
  - Maschinen (Produktion/Herstellung)
  - Konsumenten
  - Lieferanten usw.
- Man unterscheidet folgende Bereiche in denen Agenten eingesetzt werden können:
  - **Planung**
  - **Terminierung und Ablaufsteuerung** (Scheduling)

- ❏ **Monitoring** (Überwachung und Analyse)
- ❏ **Vorhersage** (von Einflussgrößen wie Bedarf) → Agentenbasiertes Lernen

➤ Die Ebenen der Entscheidungsfindung in Prozessen wie Produktion und Logistik sind miteinander in einem Kreislauf verbunden und basieren auf:

- ❏ Regelsätzen (Rules)
- ❏ Zu erfüllenden Randbedingungen (Constraints)

➤ D.h. Agenten sind Theoremprüfer und Logiklöser

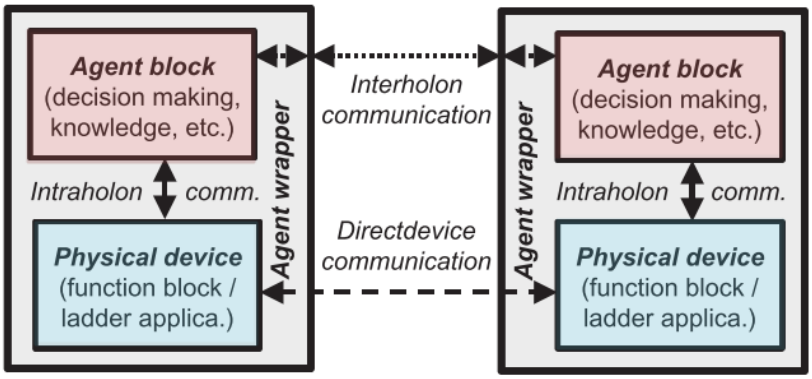


[Leitao, IAEASAI, Kap. 2, 2015]

➤ Dabei finden Aktionen teilweise in Echtzeit (online) und teilweise versetzt/offline statt

➤ Agenten können entkoppelt von physikalischen Plattformen (ABM/ABS) oder gekoppelt an physische Plattformen eingesetzt werden (ABC/Cyber Physical Systems CPS)

➤ Dabei kommunizieren die Plattformen (Maschinen, Fahrzeuge) und die Agenten miteinander:



[Leitao, IAEASAI, Kap. 2, 2015]

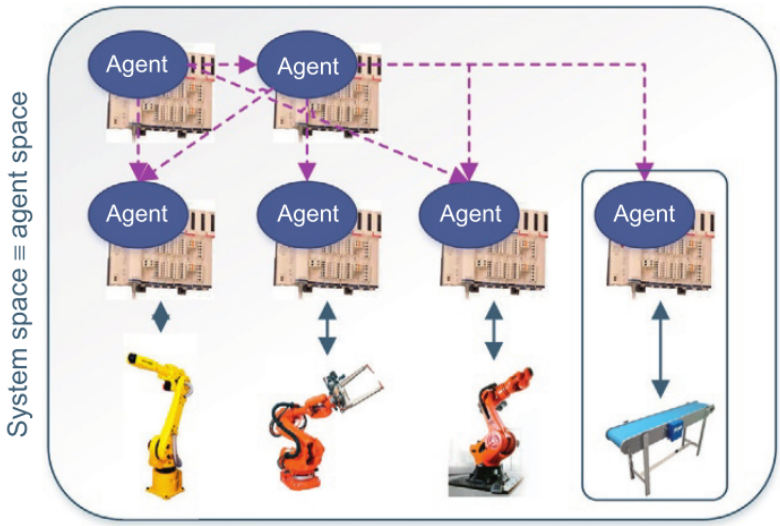
Agentenbasiert Entwurf

➤ Die Kopplung von Agentenmodellen mit dem Entwurfsprozess (z.B. einer Produktionsanlage) ist der Einstiegspunkt für agentenbezogene Prinzipien und Technologien in aktuellen Automatisierungsszenarien:

[Leitao, IAEASAI, Kap. 3, 2015]

Eingebettete Agententechnologien in der Automatisierung

➤ Der nächste Schritt ist die Einbettung von Agententechnologien in die Systeme (z.B. Maschinen):



[Leitao, IAEASAI, Kap. 3, 2015]

4.6. Plattformen

Agentenplattformen

- Domänen: ABC, ABM (Planung und Koordination)
- Weit verbreitete Programmiersprache: JAVA

Simulationswerkzeuge

- Domänen: ABC, ABM, ABMS
- Entweder eigenständig oder als Erweiterung zu bestehenden Plattformen (Beispiel: SEJAM als Visualisierungs- und Steuerungsebene für JAM)

4.7. CAPNET

CAPNET: Component Agent Platform based on .NET^{ABC}

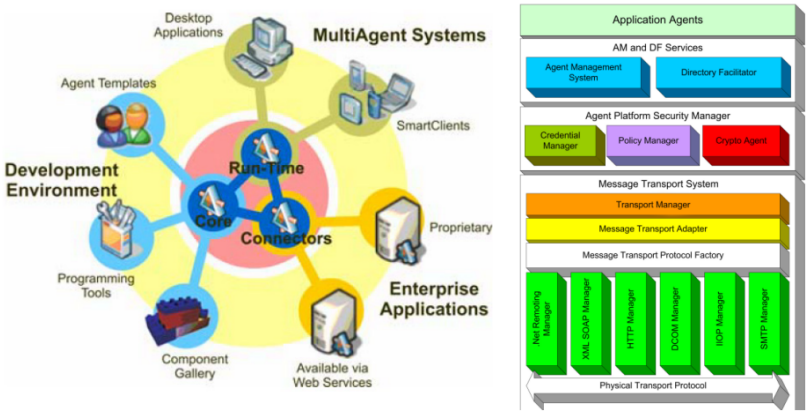


Abb. 11. (Links) Stark heterogene und verteilte Systeme (Rechts) Plattform [Contreras, 2004]

4.8. HERA

Hera: Healthcare and Homecare Services System[ABC]

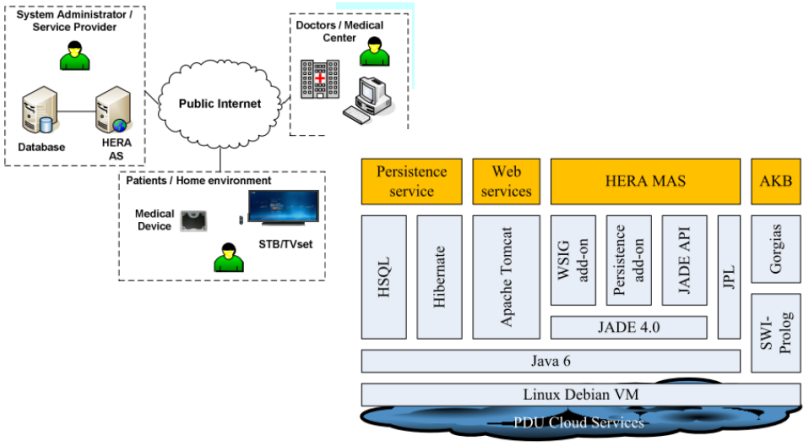


Abb. 12. (Links) HERA Helathcare and Homecare Services System (Rechts) Plattform: JAVA und JADE basierend [Spanoudakis, 2015]

4.9. MAGENTA

MAGENTA: Multi-Agenten Systeme für Logistik[ABC]

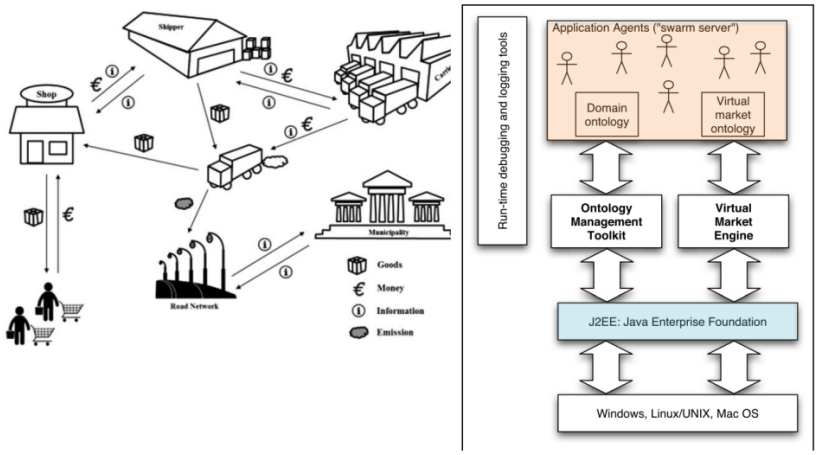
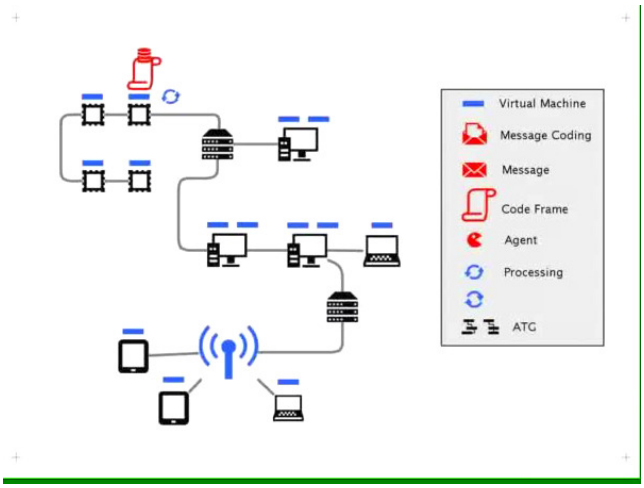


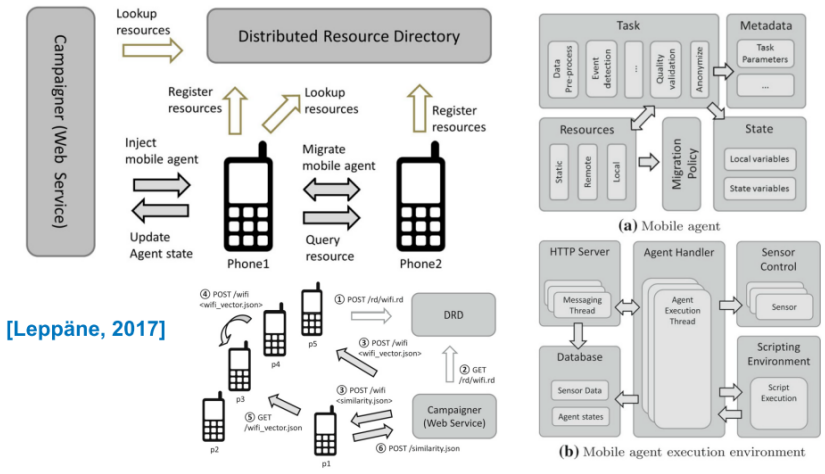
Abb. 13. (Links) Multi-Agenten Systeme für Logistik [Anand, 2014] (Rechts) Plattform [Himoff, 2005]

4.10. Mobile Cloud

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



4.11. Crowd Sensing

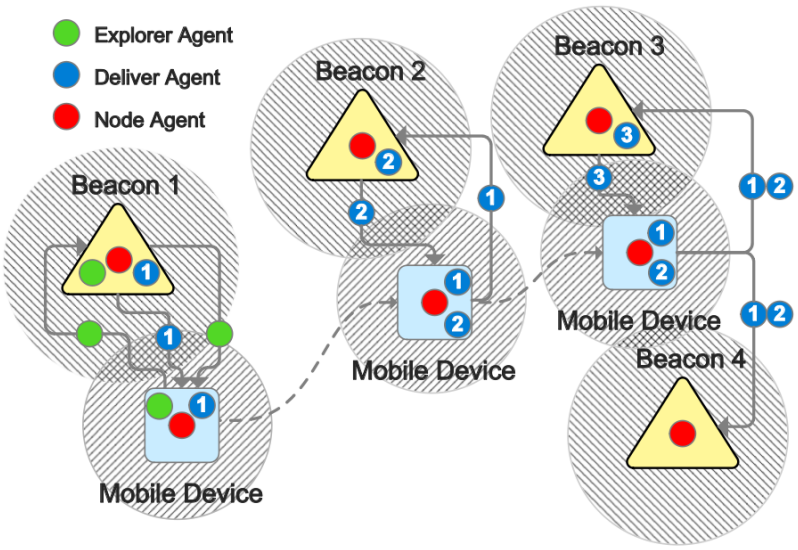


[Leppäne, 2017]

Abb. 14. (Links) Datenaustausch zwischen Smartphones und WEB Service (Rechts) Agent & Plattform

4.12. 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



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

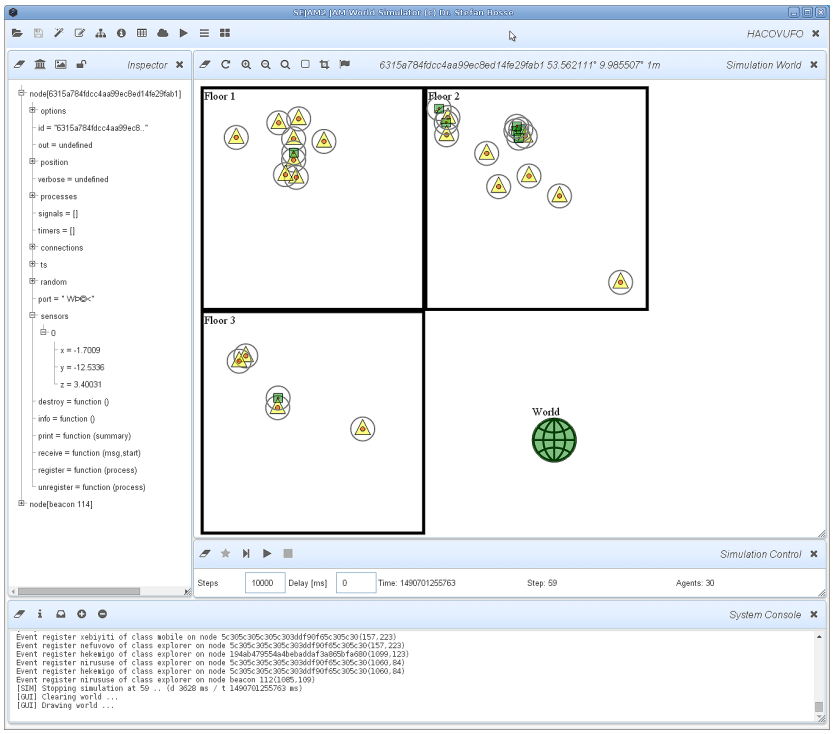


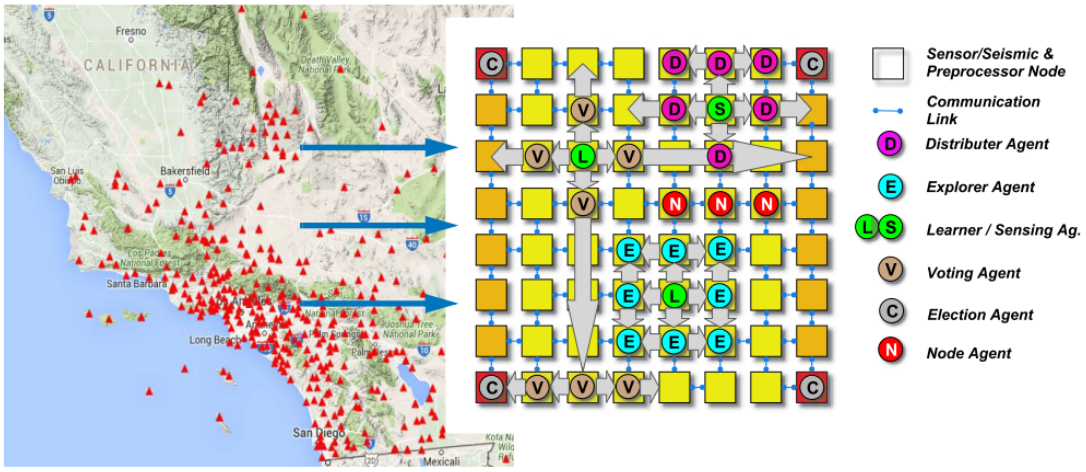
Abb. 15. (Links) Objekt Explorer (Rechts) Simulationswelt mit Beacons, mobilen Geräten (Smartphones) und Agenten

4.13. iSENSNET

iSENSNET: Intelligent Sensor Processing in Sensor Networks

Erdbebenüberwachung und Ereignisklassifizierung

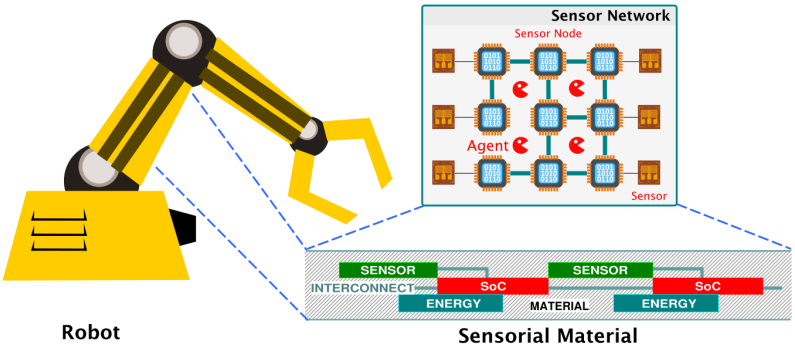
- ➤ Verteiltes seismisches Netzwerk-CI (South California), das auf einem 2D Mesh-Netzwerk abgebildet ist (JAM Agentenplattform)
- ➤ Mobile Agenten werden zum verteilten und inkrementellen Lernen eingesetzt [1]



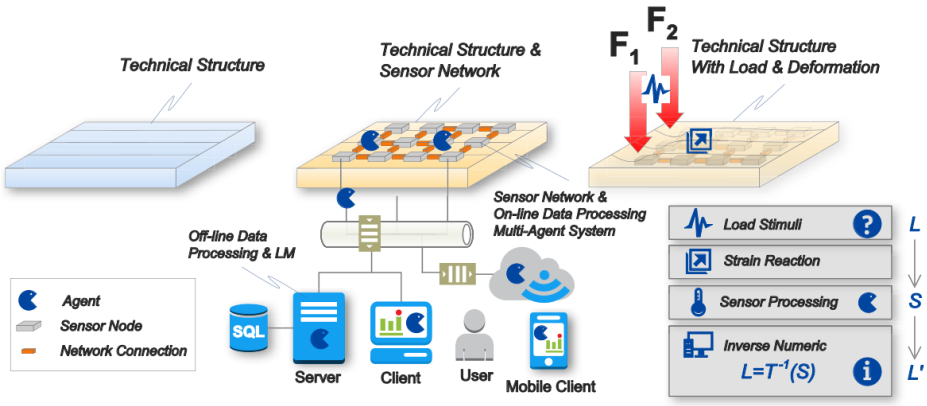
- Erweiterung des seismischen Netzwerkes mit mobile Geräten wie Smartphones
- Smartphones können bei Erdbeben zusätzliche Informationen liefern
- Mobile Agenten können zwischen seismischen und mobilen Netzen/Geräten migrierien

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*



- 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)



4.14. NetLogo

- ➤ Reines ABM/ABS Werkzeug
- ➤ Programmiert in JAVA
- ➤ Simulation + Analyse
- ➤ Klasse: Agenten-basiertes Räumliches Modellieren und Simulieren (ABSS)
- ➤ Einsatzgebiete:
  - ❑ Verkehrssimulation
  - ❑ Biologie (Tierverhalten, Schwärme, Vermehrung, Ausbreitung, ..)
  - ❑ Soziologie (Migration, Netzwerkbildung, Segregation, …)
  - ❑ Logistik (Warentransport und Transportströme)
  - ❑ Medizin (Genetik, Populationsentwicklung, Epidemiologie, usw.)
  - ❑ Psychologie (Spieltheorie, Interaktion)
  - ❑ Allg. Netzwerke (Internet)
  - ❑ Umwelt (Ausbreitung von Schadstoffen usw.(
  - ❑ Chemie (Entstehung von Molekülen, Gasen, usw.)

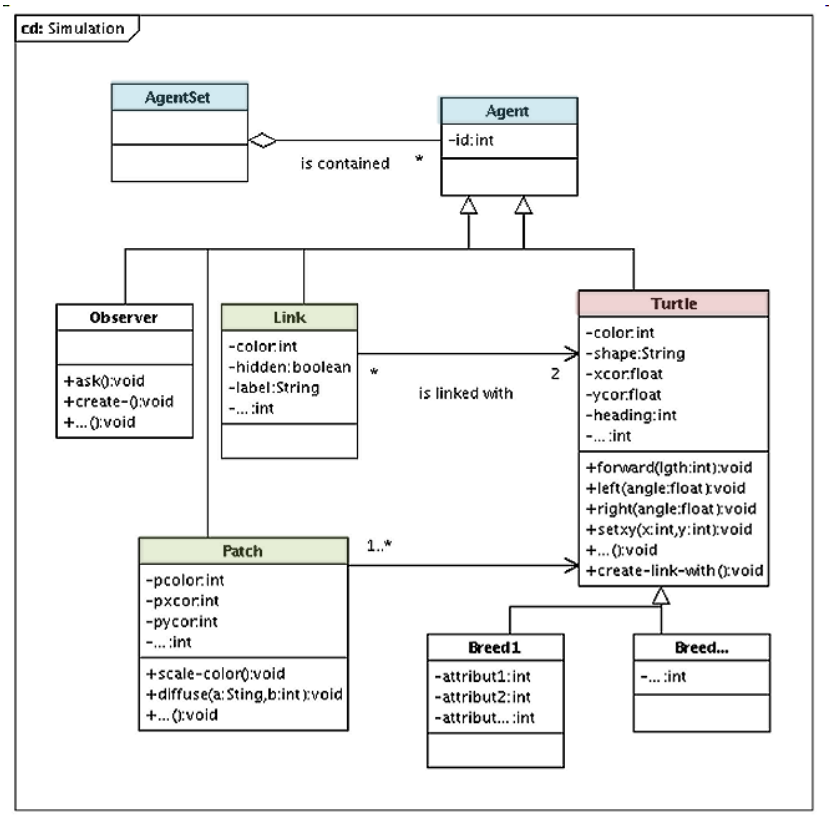
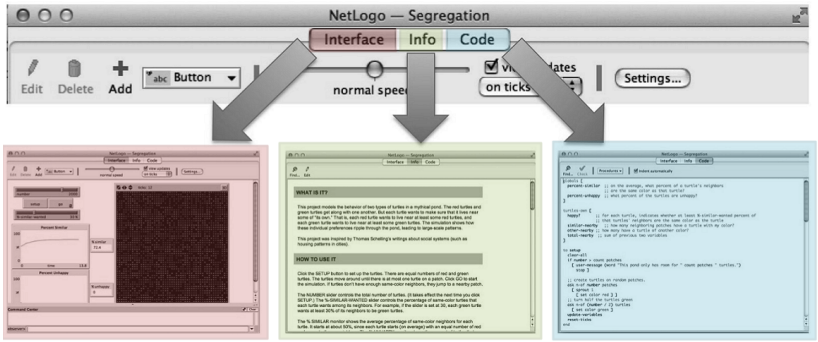


Abb. 16. Das NetLogo Agenten- und Programmiermodell [Banos ed. ABSSN, 2015]

➤ Die grafische Oberfläche (GUI) besteht aus der eigentlichen Simulationsschntittstelle (Simulationswelt, Steuerung, Analyse), Informationen, und einem Modelleditor:



4.15. JavaScript Agent Machine (JAM)

- ➤ Wird in diesem Kurs für ABC eingesetzt!
- ➤ Programmiersprache: JavaScript
- ➤ Modell: Aktivitätsübergangsgraphen (ATG), reaktiv

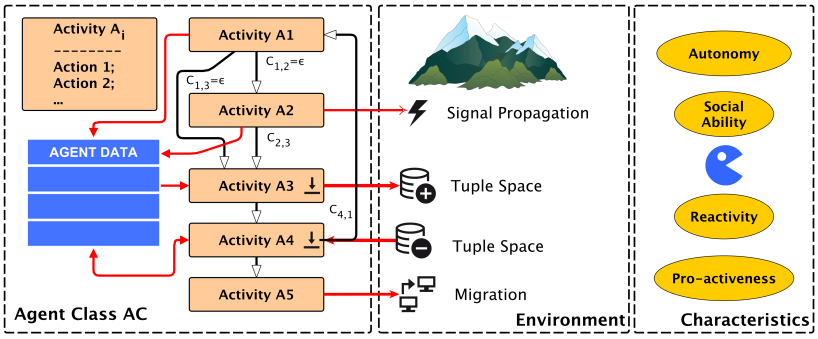


Abb. 17. JAM Agenten unt ATGs

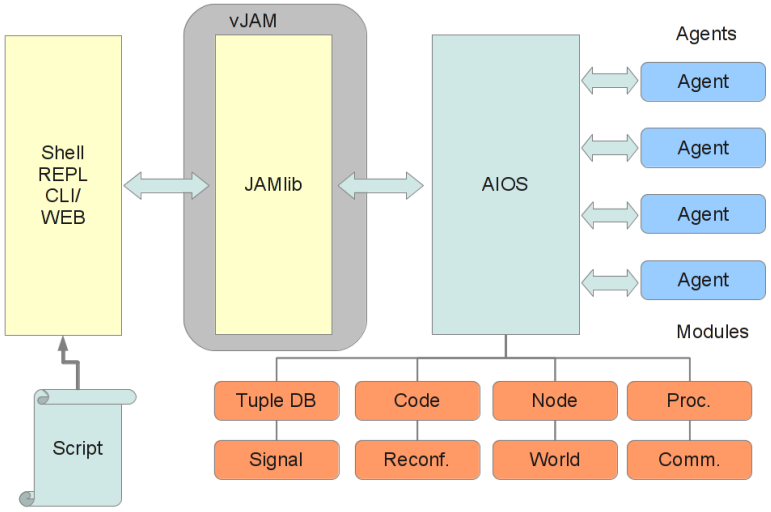


Abb. 18. Softwarearchitektur von JAM: Zentral ist das Agent Input-Output System (AIOS)

4.16. SEJAM

- ➤ Visualiserungserweiterung von JAM (ABC/ABS/ABM)

- Fügt eine Visualisierung einer virtuellen zweidimensionalen Welt hinzu die aus einer Menge von verbundenen virtuellen JAM Knoten besteht
- Neben der Programmierung der Agenten (wie in JAM) gibt es zusätzlich ein Simulationsmodell (Beschreibung und Aufbau der Simulationswelt)
- Es werden zwei verschiedene Agentenklassen unterschieden (wenn auch programatisch identisch):
  - Berechnungsagenten (ABC)
  - Physische Agenten (ABM)

Berechnungsagenten

- Mobiler Code, nicht an eine JAM Plattform gebunden
- Ausführbar in der Simulation und der "realen" Welt (Z.B. auf einem Smartphone)

Physische Agenten

- Nur in der Simulation: Agent und (virtuelle) Plattform bilden unzertrennliche Einheit → ähnlich den NetLogo Agenten
- Mobil durch die räumliche Verschiebung der virtuellen Plattform (der "Körper")

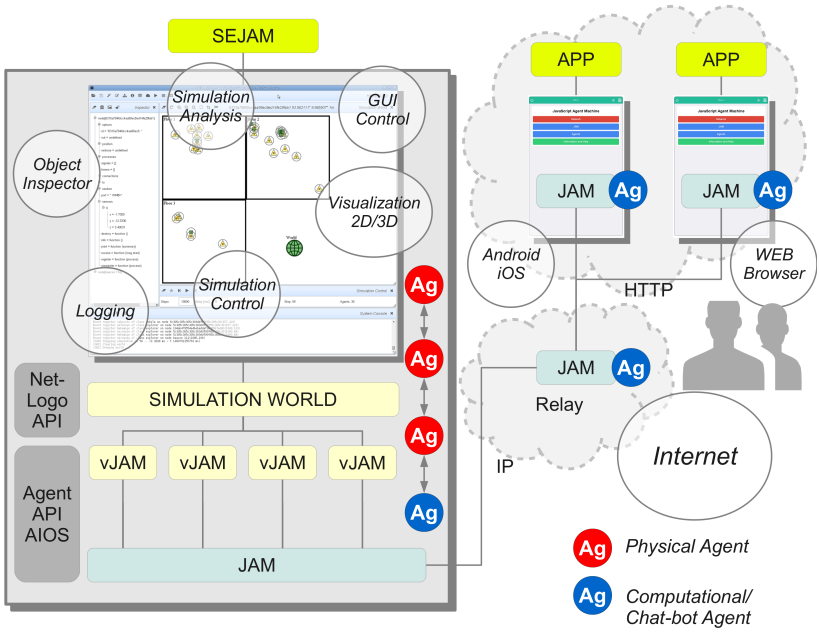


Abb. 19. Softwarearchitektur von SEJAM

4.17. JAM/SEJAM: Beispiele

[j19.1] S. Bosse, D. Lehmhus, *Material-integrated cluster computing in self-adaptive robotic materials using mobile multi-agent systems*, Cluster Computing, doi 10.1007/s10586-018-02894-x, Volume 22, Number 3, pp. 1017-1037, 2019 ISSN 1386-7857

[j19.3] S. Bosse, U. Engel, *Real-time Human-in-the-loop Simulation with Mobile Agents, Chat Bots, and Crowd Sensing for Smart Cities*, Sensors (MDPI), 2019, doi: 10.3390/s19204356

[c19.2] S. Bosse, D. Lehmhus, *Robust detection of hidden material damages using low-cost external sensors and Machine Learning*, 6th International Electronic Conference on Sensors and Applications (ECSA), 15-30 Nov. 2019, MDPI

[c18.3] S. Bosse, M. Koerdt, A. v. Hehl, *Robust and Adaptive Non Destructive Testing of Hybrids with Guided Waves and Learning Agents*, 3. Internationale Konferenz Hybrid Materials and Structures 2018, 18-19.4.2018, Bremen, Germany

[c18.5] S. Bosse, *Smart Micro-scale Energy Management and Energy Distribution in Decentralized Self-Powered Networks Using Multi-Agent Systems*, FedCSIS Conference, 6th International Workshop on Smart Energy Networks & Multi-Agent Systems, 9-12.9.2018, Posznan, Poland, 2018

[c18.7] S. Bosse, U. Engel, *Augmented Virtual Reality: Combining Crowd Sensing and Social Data Mining with Large-Scale Simulation Using Mobile Agents for Future Smart Cities.* Proceedings, Volume 4, ECSA-5 5th International Electronic Conference on Sensors and Applications 15–30 November, 2018 DOI 10.3390/ecsa-5-05762

[j17.1] 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

[c17.2] 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

4.18. SEJAM2

5. Agentenmodelle und Architekturen

5.1. Agentenbasiertes Modellieren (ABM)

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

- ➤ Viele numerische (funktionssbasierte) 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.

5.2. Verhaltensmodelle

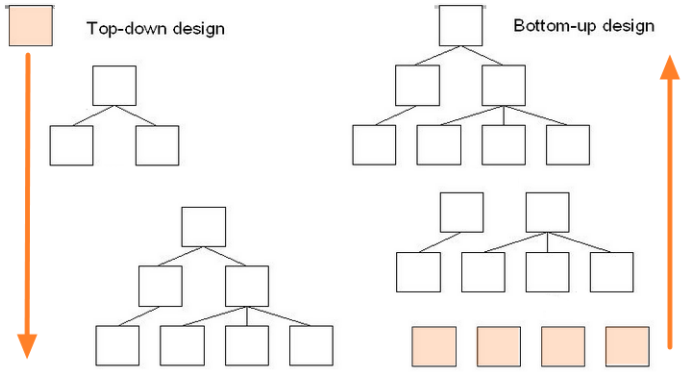
Man unterscheidet zwei grundsätzliche Methoden der Modellierung von Agenten:

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.



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)

Man unterscheidet bei ABM und ABS folgenden Entitäten:

Aktive Agenten
: Agenten die durch eine Menge privater Daten, einem veränderlichen Ort, Eigenschaften, und einem Verhaltensmodell gekennzeichnet sind. Diese Agenten interagieren (kommunizieren) aktiv mit anderen Agenten und verändern die Welt (Daten)

Passive Agenten
: Agenten die optional durch eine Menge privater Daten, i.A. Ortsgebundenheit, und Eigenschaften gekennzeichnet sind. Diese Agenten sind Ressourcen die nicht aktiv mit anderen Entiäten interagieren.

Verbindungen
: Es gibt Verbindungen (links) zwischen den Agenten die eine Interaktion (Kommunikation) ermöglichen

Welt
: Eine i.A. diskrete Welt mit Koordinaten, Eigenschaften, die mit einer Menge von Agenten besetzt ist.

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* beschreibt das Verhalten eines Multiagentensystems anhand von Berechnungen und Anweisungen als *ganzes*:

- ❑ Modellierung der gesamten Simulationswelt in einem Modell

5.3. 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 (passiver Agent)

➤ Ein Patch besteht aus einer *Visualisierung* (Farbe) und optional *Datenvariablen* (Eigenschaften)

  ❏ Die Datenvariablen entsprechen im Modell der Zellulären Automaten den Zellvariablen

➤ Patches sind nicht mobil

➤ Patches ermöglichen die Diskretisierung der analogen Welt

  ❏ Vor- und Nachteil zugleich. Warum?

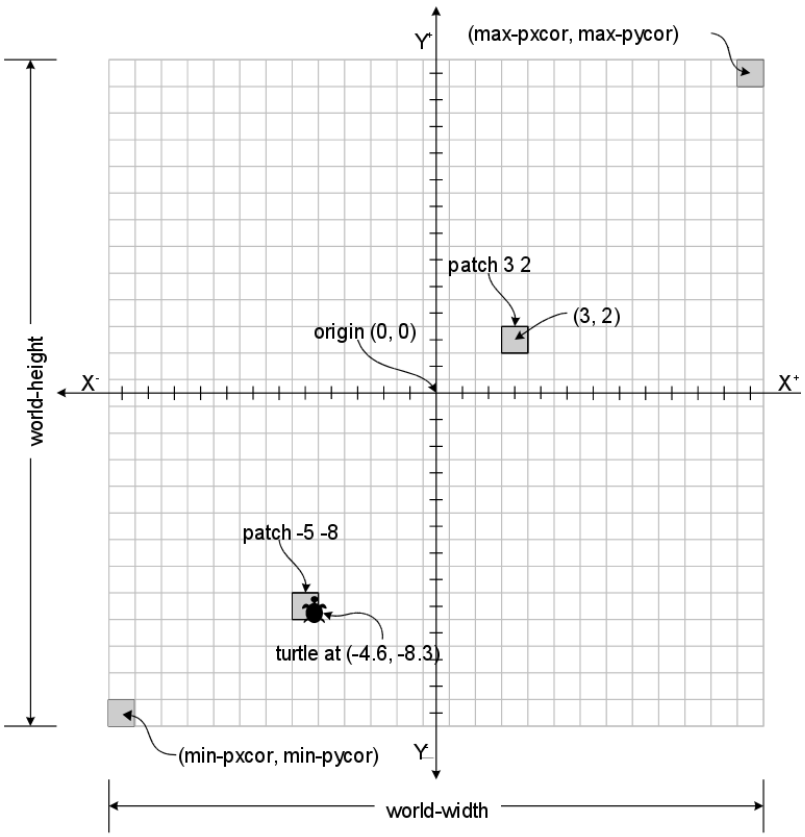


Abb. 20. Simulationswelt und kartesische Koordinaten

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
```

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 ]
  .. ]
```

Dynamische Erzeugung von Agenten

Definition 5. (*Erzeugung von neuen Agenten auf dem aktuellen Patch*)

```
sprout num
sprout-class num
sprout num [ statements ]
sprout-class num [ statements ]
```

Aufgabe

1. Öffne NetLogo und das Simulationsmodell *Sample Models* → *Biology* → *Ant Lines* und stelle folgende Liste zusammen:
   - ➤ Arten von Agenten (Turtles), Unterteilung in passive und aktive Agenten
   - ➤ Visualisierung der Agenten (shape)
   - ➤ Globale Variablen
   - ➤ Agentenvariablen
   - ➤ Prozeduren
2. Wo werden die Patches erzeugt/verändert?
3. Wo wird die Simulationswelt erzeugt?
4. Wie werden die Agenten in der Welt bewegt?
5. Welche Sensoren verwenden die Agenten?
6. Wie kommunizieren die Agenten?

Setup

➤ Ein Setup der Simulation besteht i.A. aus folgenden Schritten und wird am Anfang durch Aufruf einer Prozedur durchgeführt:

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
```

Simulationsschritt

➤ In jedem Simulationsschritt wird im Simulationsmodell eine Prozedur ausgeführt die die Agenten und die Welt verändert.

➤ Die Simulationszeit wird um den Wert eins erhöht indem die `tick` Anweisung ausgeführt wird.

Beispiel

```
to go
  ask wolfes
    [
       if (hungry > 100)
        [ forward 10 ]
    ]
  tick
end
```

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 6.

```
ask agentset  [
  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!

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 7.

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

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 und innerhalb einer `ask` Anweisung anwendbar!

5.4. Soziologische Modelle und Simulation

➤ Neben biologischen Modellen werden soziologische Modelle durch ABM/ABS behandelt

➤ Häufige soziologische Themen:

- ❏ Soziale Netzwerk- und Gruppenbildung
- ❏ Segregation (räumliche Gruppenbildung)
- ❏ Migration

Soziales Sakoda Interaktionsmodell

- ➤ Das ursprüngliche Sakoda-Verhaltensmodell besteht aus sozialen Interaktionen zwischen zwei Gruppen von Individuen, die sich in einem Netzwerk nach spezifischen Einstellungen hinsichtlich Anziehung, Abstoßung und Neutralität entwickeln.
- ➤ Ein Individuum bewertet seine gesellschaftliche Erwartung an allen möglichen verfügbaren Standorten (beginnend am aktuellen Standort),
  - ❏ bevorzugt Nähe zu Personen, die mit attraktiven (positiven) Einstellungen verbunden sind, und
  - ❏ vermeidet Nähe zu Personen, die mit abstoßenden (negativen) Einstellungen assoziiert sind.
- ➤ Dieses Verfahren wird unter allen möglichen Individuen zufällig wiederholt;
  - ❏ Von nun an wird der Algorithmus von Sakoda wiederholt iteriert, wobei ein räumlich verteiltes soziales System zu einem organisierten räumlichen Muster entwickelt wird.
- ➤ Dies hängt jedoch vom Parametersatz des Modells, der Personendichte und der Individualisierung ab.
- ➤ Der Einfachheit halber gibt es eine zweidimensionale Gitterwelt, die aus diskreten Orten (x, y) besteht.
- ➤ Ein Agent nimmt eine Stelle des Gitters ein.
- ➤ Maximal ein Agent kann einen Ort besetzen.
- ➤ Die Agenten können sich auf dem Gitter bewegen und ihre Position ändern.
- ➤ Es wird angenommen, dass es zwei Gruppen gibt, die sich auf die Klassen a und b von Individuen beziehen.
- ➤ Die soziale Interaktion ist gekennzeichnet durch unterschiedliche Einstellungen eines Individuums zwischen verschiedenen und zwischen gleichen Gruppen, die durch vier Parameter gegeben werden:

$$S = (s_{aa}, s_{ab}, s_{ba}, s_{bb})$$

- Das Weltmodell besteht aus N Stellen x_i.
- Jeder Platz kann von keinem oder einem Agenten belegt werden, entweder von Gruppe α oder β, ausgedrückt durch die Variable $x_\mathrm{i} = \{0, -1, 1\}$
- Die gesellschaftliche Erwartung eines Agenten i am Ort x_i ist gegeben durch:

$$f_i(x_i) = \sum_{k=1}^{N} J_{ik}\delta_s(x_i, x_k)$$

- Der Parameter J_ik ist ein Maß für die soziale Entfernung (gleich eins für die Moore-Nachbarschaft mit Entfernung eins) und nimmt für längere Entfernungen ab.
- Der Parameter δ drückt die Einstellung zu einem Nachbarort aus, gegeben durch:

$$\delta_s(x_i, x_k) = \begin{cases} s_{\alpha\beta} & , \text{ if } x_i \neq 0 \text{ and } x_k \neq 0 \text{ with } \alpha = x_i, \beta = x_k \\ 0 & , \text{ otherwise} \end{cases}$$

- Ein einzelner Agent ag_i einer Gruppe α (Klasse aus der Gruppe von Gruppen) kann seine Position ändern, indem er vom aktuellen Ort x_i an einen anderen Ort x_m wandert,
  - wenn dieser Platz nicht belegt ist ($x_\mathrm{m} = 0$) und
  - wenn $f_\mathrm{i}(x_\mathrm{m}) > f_\mathrm{i}(x_\mathrm{i})$.

Aufgabe

1. Implementiere Agenten mit dem Sakoda Verhalten
2. Die Welt soll parametrisierbar mit einer Anzahl a/b Agenten bevölkert werden
3. Der S Parametersatz soll auch parametrisierbar sein
4. Führe exemplarisch Simulationen durch. Welche Strukturen ergeben sich für die folgenden S Vektoren: (1,-1,-1,1), (1,1,1,1) (-1,1,1,-1)

5.5. 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

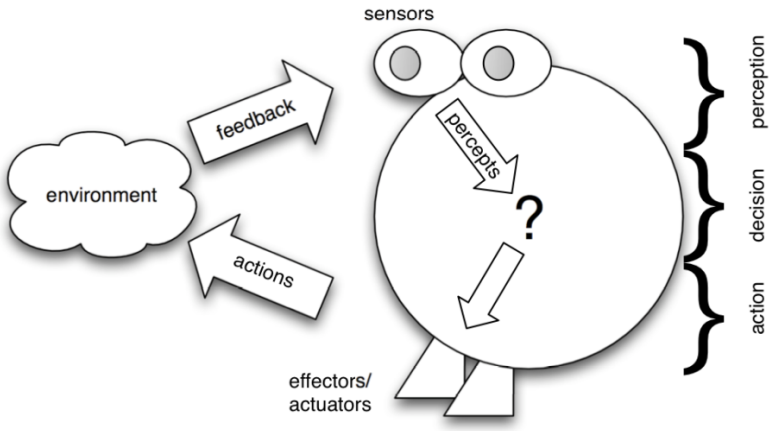


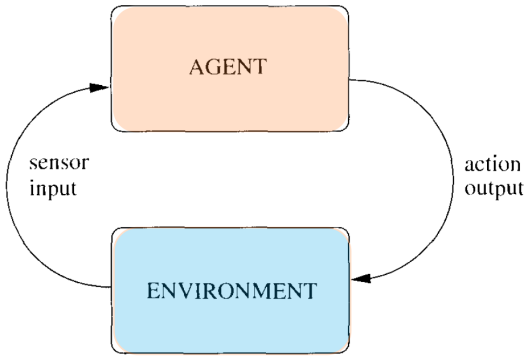
Abb. 21. Wechselwirkung von Agent und Umgebung [B]

Agentenmodell

➤ Ein Agent in seiner Umgebung nimmt sensorische Eingaben aus der Umgebung auf und produziert Ausgabe-Aktionen, die ihn *und* die Umgebung beeinflussen.

➤ Die Interaktion ist gewöhnlich fortlaufend und nicht-terminierend!

Einfachstes Agentenmodell



[J]

Beispiel

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

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

5.6. 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!

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 …

5.7. 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

5.8. 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

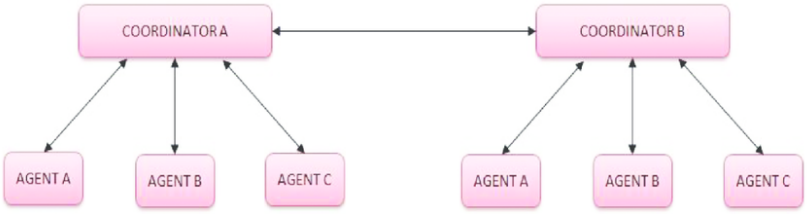
Auflösung des Wettbewerbskonflikts durch *entweder oder* (EXOR) Entscheidung.

- ➤ 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

Architekturen

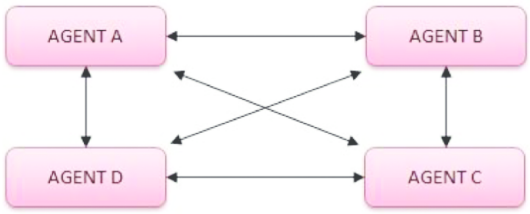
Zentralisiert und hierarchisch

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



Dezentralisiert

Es gibt eine Selbstorganisation ohne ausgewiesene zentrale Koordination



5.9. 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.

5.10. 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:

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

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 (Ziele erfüllen)!

5.11. 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

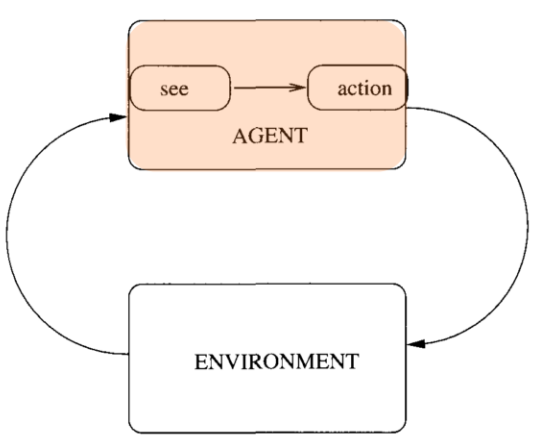


Abb. 22. Verkettete Wahrnehmungs- und Aktionssubsysteme

➤ Rein reaktive Agenten werden eine Aktion a nur auf dem *aktuellen* Zustand $e(t) \in E$ basierend auswählen:

$$Ag_r : E \rightarrow Ac$$

➤ Die Historie der Welt wird nicht in Betracht gezogen

Beispiel: Heizungsagent

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

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:

$$action : Per* \rightarrow Ac$$

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:

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

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

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

5.12. Weltzustand und Perzeption

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

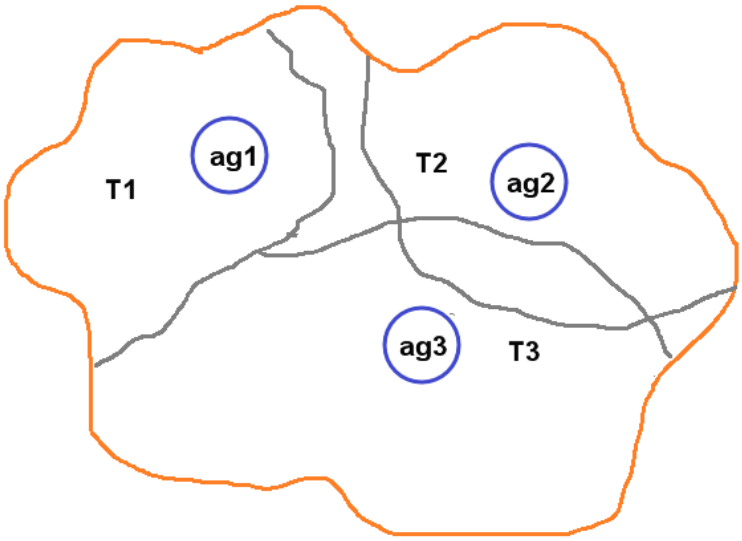


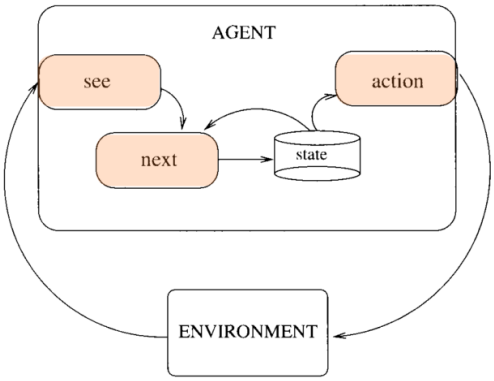
Abb. 23. Die Welt besteht aus drei Agenten und drei Bereichen mit den Perzeptionen T_1, T_2, und T_3. Der Weltzustand $e_i \in E(T_1, T_2, T_3, ag_1, ag_2, ag_3)$ wird von den einzelnen Agenten unterschiedlich und nur teilweise wahrgenommen (Perzeption $ag_1 = \{T_1\}$, $ag_2 = ag_3 = \{T_2, T_3\}$)

5.13. 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



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:

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

Verhalten

➤ Das Verhalten eines 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) !?

5.14. 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 u ∈ U können Weltzuständen (zu den der Agent auch gehört) oder Durchläufen eine reelle Bewertungszahl (i.A. [0,1]) 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.

Fliesenwelt ist ein Beispiel für eine dynamische Welt!

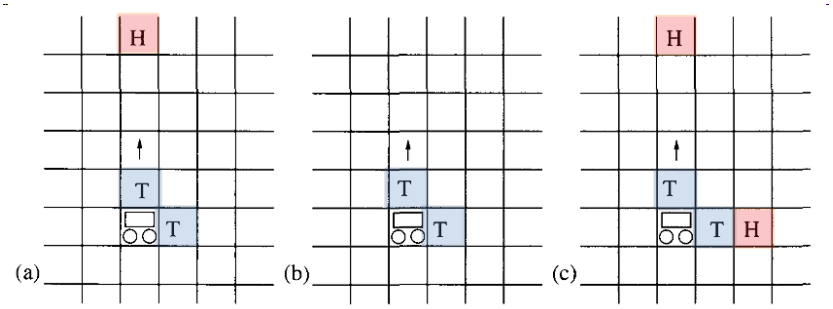


Abb. 24. Ein Agent in einer 'Fliesenwelt' mit Fliesen (T) und Löchern (H): Verschiedene Situationen (a) - (c)

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.

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)

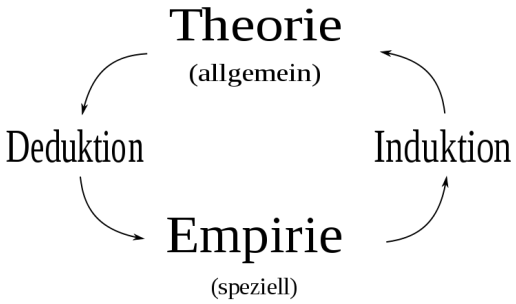
➤ Eine mögliche Nützlichkeitsfunktion könnte das Verhältnis der aufgetauchten Löcher zu den erfolgreich während eines Durchlaufs r gefüllten in Relation stellen und das Agentenverhalten beeinflussen:

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

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

5.15. 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 $p \rightarrow q$ (sprich: wenn p, dann q) wahre Aussagen, so ist auch q eine wahre Aussage.
- 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.

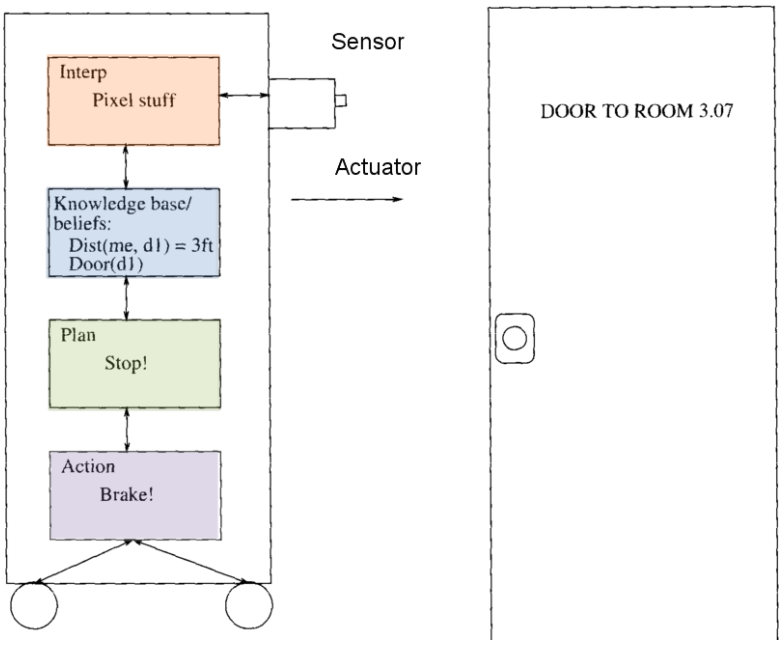


Abb. 25. Beispiel eines robotischen Agenten der eine symbolische Beschreibung der Welt besitzt [B]

Deduktive Agenten sind Kognitive Systeme

- ➤ 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 8. (*Verhalten deduktiver Agenten*)

Perzeption → Interpretation → Wissen → Planung → Aktion

- ➤ 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.
- 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 $Do(\alpha)$ für Aktionen α ∈ 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 $\neg Do(\alpha)$ gibt die sich derzeit nicht aus der Datenbank ableiten lässt (d.h. es ist nicht verboten es zu tun), *dann* wähle diese Aktion aus.

```
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. 26. Aktionsauswahl als Theoremprüfung [B]

Beispiel: Staubsageragent

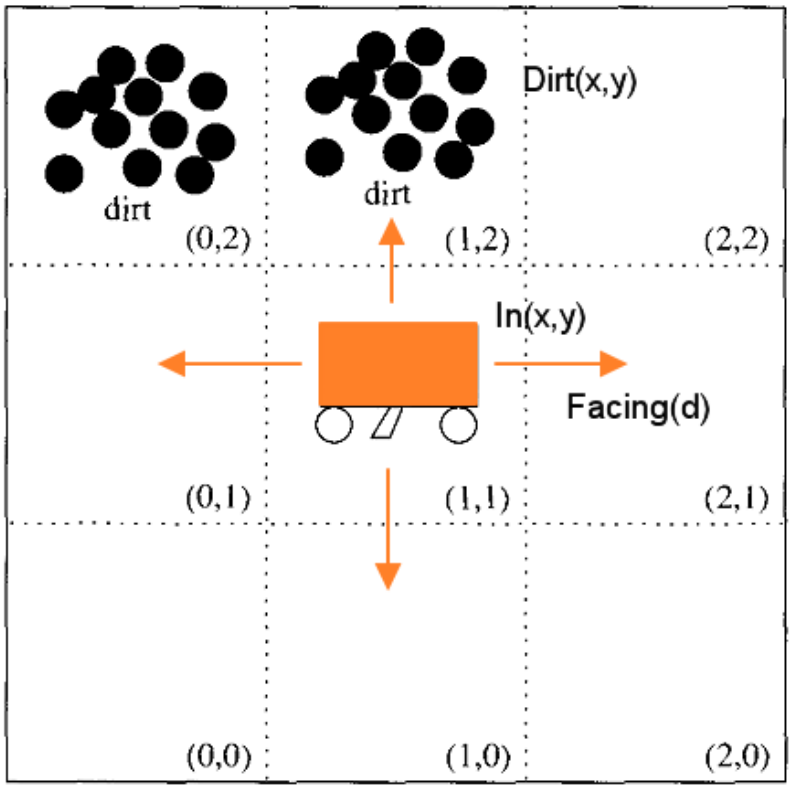


Abb. 27. Ein Agent in einer zweidimensionalen diskreten Welt aus Feldern (Position *x,y*) mit den Attributen *Dirt(x,y)*, *In(x,y)*, *Facing(dir)*

➤ 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 $old(\Delta)$ identifiziert und mittels einer *new* Funktion durch die *next* Funktion aktualisiert:

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

- Regeln in der Wissensdatenbank (Auswahl):

  $In(x, y) \wedge Dirt(x, y) \rightarrow Saugen$
  $In(x, y) \wedge \neg Dirt(x, y) \rightarrow Forwaerts(?)$, 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)*

5.16. 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

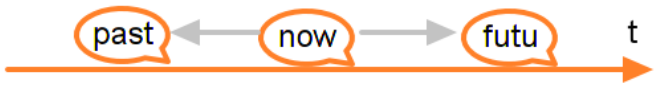


Abb. 28. Zeitliche Zusammenhänge von Bedingungen und Aussagen werden durch Operatoren beschrieben [B]

5.17. 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.

Eigenschaften

- ➤ Agent0 ist eine einfache Multiagenten Programmiersprache mit einer dazugehörigen Agentenarchitektur und einem integrierten Kommunikationsmodell zwischen Agenten
- ➤ Die Agentenumgebung ist verteilt und partitioniert.
- ➤ Jeder Agent verfügt über einen lokalen Nachrichtenpuffer und eine jeweils andere lokale Umgebung.
- ➤ Agenten arbeiten synchron - ihre Wahrnehmungs- und Aktions-Zyklen sind synchronisiert. Zeit ist nur die Zyklusnummer bestimmt.
- ➤ Jeder Agent kann nur Nachrichten sehen, die von anderen Agenten im vorherigen Zyklus an ihn gesendet wurden.

Aktionen und Nachrichten

- ➤ Aktionen können privat/intern oder kommunikativ sein → **Nachrichten**
- ➤ Nachrichtentypen:
  - ❏ Anfragen (*REQUEST*, eine Aktion zukünftig durchzuführen)
  - ❏ Abbruch von Anfragen (*UNREQUEST*)
  - ❏ Informativ (*INFORM*, Wissensübertragung)
- ➤ Die Anfragen verändern i.A. die Commitments (Ziele) des Agenten,
- ➤ Die informativen Nachrichten verändern das Wissen (Beliefs)
- ➤ Für jede Kommunikationsaktion muss der empfangende Agent angegeben werden.
- ➤ Die REQUEST-Nachricht kann sich auf eine der sechs Agentenaktionen beziehen (folgend)

Regeln für Zusagen (Commitments)

- Zusagen werden als Paare (*agent*, *action*) gehalten, wobei die Zusage an Agent *agent* gebunden ist.
- Es gibt eine Reihe von Regeln der Form:

```
COMMIT (Nachrichtenmuster, Mentalcond, Agent, Aktion)
```

- Die Aktion kann eine der sechs zulässigen Aktionsformen haben oder eine Folge davon sein [a]:

```
DO(time,privateaction)
INFORM(time,agent,fact)
REQUEST(time,agent,action)
UNREQUEST(time,agent,action) - removes commitment
REFRAIN(action) - will not enter a commitment to action
IF mentalcondition THEN action
```

Definition 9. (*Commitment mit Kommunikation*)

```
COMMIT( (Ag, REQUEST(Act)),
        (_,condition(Ag),
        Ag,
        Act)
```

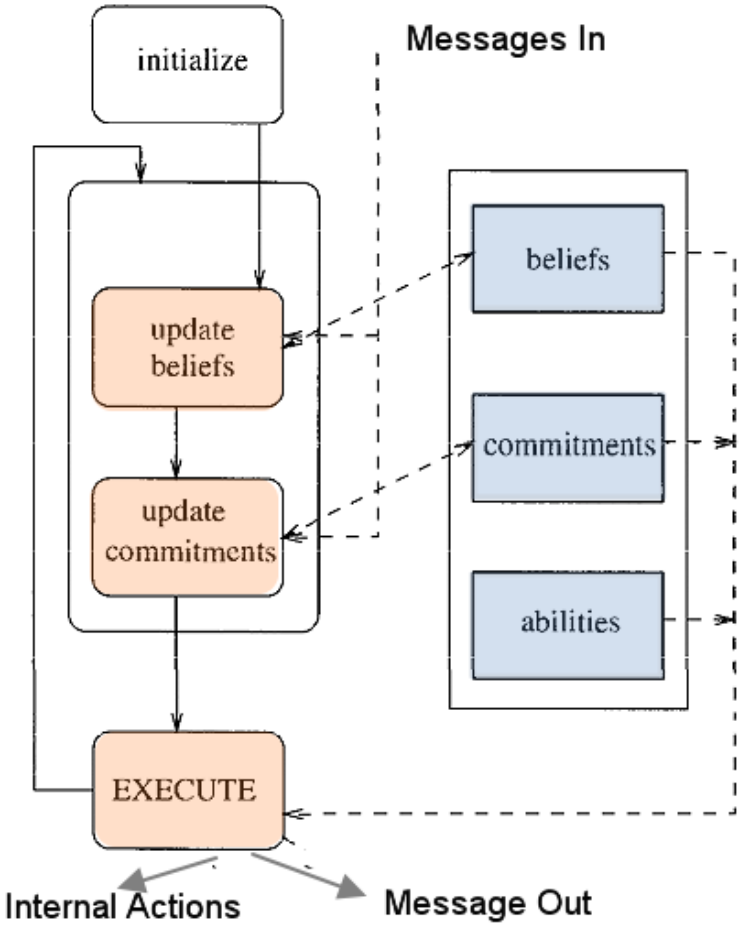
Bedeutet: Kann sich möglicherweise dazu verpflichten, die Aktion *Act* für Agent *Ag* auszuführen, wenn *Ag* gerade *Act* angefordert hat und der Agent glaubt, dass *Ag* die Bedingung *condition* erfüllt.

Textuelle Formulierung

```
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) )
```

'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, if I am not committed to doing any other action, then commit to doing action at time.'

Nachrichtenbasierter Kontrollfluss in AGENT0


initialize
Messages In
beliefs
update beliefs
commitments
update commitments
abilities
EXECUTE
Internal Actions
Message Out

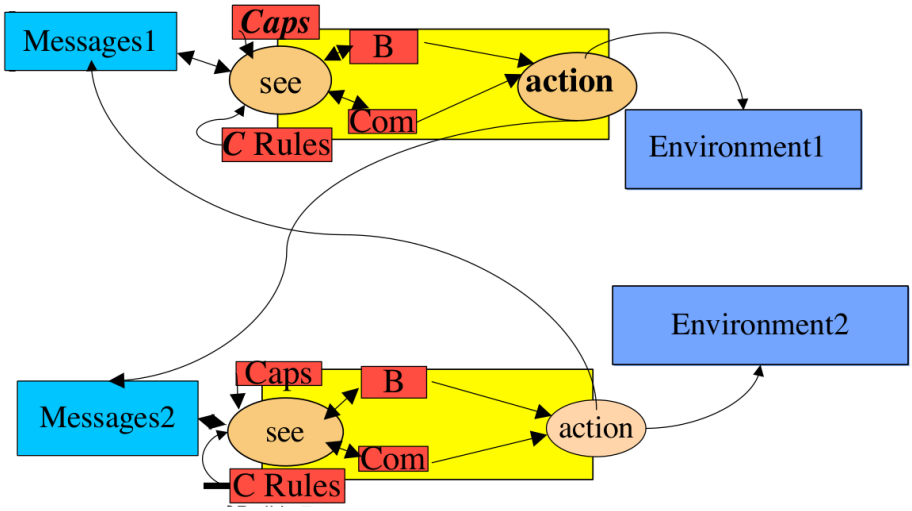


Abb. 29. Interaktion zweier *Agent0* Agenten über Nachrichten [a]

5.18. Agentenorientiertes Programmieren: AGENT0 Zyklus

See Funktion

- Testet jede eingehende Nachricht M gegen jede commitment Regel [a]

```
COMMIT(messagepattern,mentalcond,agent,action)
```

Nachrichten erzeugen ein Commitment (*agent,action*)

wenn

- M ist in *messagepattern* enthalten, und
- *mentalcond* ist momentan wahr, und
- (unter Benutzung der Capabilityfakten wenn erforderlich)
- der Agent nimmt momentan an er ist fähig *action* auszuführen, und
- der Agent ist nicht verpflichtet (REFRAIN) die Aktion zu unterlassen.

oder

- *action* ist in der Form REFRAIN(*act*), und
- der Agent ist momentan nicht für *act* verpflichtet.

Revision

Algorithm 1.

If *a read − in message is of the form* :
"*UNREQUEST*(*Act*) *from an agent Ag*"
Then If *present* (*Ag*, *Act*) *Then remove commitment* (*Ag*, *Act*)

If *it is an INFORM*(*Fact*) *message from agent Ag* :
Then *add* (*Ag*, *Fact*) *to agent*′*s set of beliefs and*
If *present* (, *comp*(*Fact*))
Then *remove any belief* (, *comp*(*Fact*))
Where *comp*(*Fact*) *is the complement of Fact*

Action Funktion

- Durchsucht eine Reihe von Zusagen mit einer Aktion für die aktuellen Zykluszeit

Algorithm 2.

If *action is an unconditional action*,
Then *execute the action*.
If *it is a conditional action of the form* :
"*IF mentalcond THEN action*"
Then *execute action if mentalcond holds*.

5.19. Das konzeptuale Agentenmodell

Ein Agent wird durch drei interagierende Kontroll- und Wissensebenen definiert:

- Eine **verhaltensbasierte Ebene**, die Reaktivität und Verfahrenswissen für Routineaufgaben enthält.
- Eine **lokale Planungsebene**, die die Möglichkeit bietet, über Mittel und Ziele nachzudenken, um lokale Aufgaben zu erfüllen und zielgerichtetes Verhalten zu erzeugen.

➤ Eine **kooperative Planungsebene**, die es Agenten ermöglicht, über andere Agenten zu schlussfolgern, und die koordinierte Aktion mit anderen Agenten unterstützt.

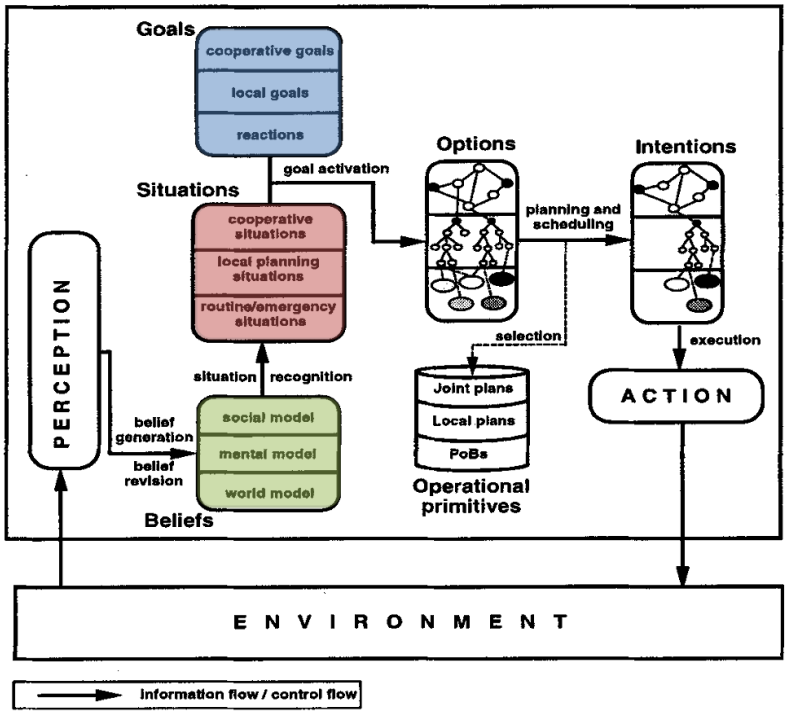


Abb. 30. Informationsfluss im verhaltens- und wissensbasierten Agentenmodell und Steuerungsarchitektur [A]

Mentale Kategorien

Der mentale Zustand eines Agenten setzt sich aus verschiedenen Komponenten zusammen:

➤ Die **aktuelle Wahrnehmung** des Agenten.

➤ Eine Reihe von **Überzeugungen**, die den Informationszustand beschreiben.

➤ Eine Reihe von **Situationen**, die relevante strukturierte Teile der Überzeugungen des Agenten beschreiben.

➤ Eine Reihe von (kontextunabhängigen) **Zielen**, die der Agent möglicherweise hat.

➤ Eine Reihe von **Optionen**, die den Motivationszustand (motivational state) des Agenten darstellen.

- ❏ Basierend auf der aktuellen Situation wird eine Reihe von **kontextabhängigen Optionen** aus der Reihe der möglichen Ziele ausgewählt.
- ❏ Daher sind Optionen Ziele, die der Agent unter bestimmten Umgebungsbedingungen verfolgen kann

➤ Eine Reihe von **Absichten**, die den Überlegungszustand (deliberative state) des Agenten definieren, d.h. die Optionen, mit denen der Agent sich selbst verändert hat, und die Definition der nächsten Aktion, die der Agent ausführen wird.

➤ Eine Reihe von **operativen Primitiven**, die den Motivationszustand aller Agenten mit ihrem Überlegungszustand verbinden.

Funktionen

Annahmenbildung und Annahmenprüfung
erklären die Beziehung zwischen den Überzeugungen eines Agenten und seiner aktuellen Wahrnehmung. Sie beschäftigen sich mit der Frage, wie Wahrnehmung in Überzeugungen umgewandelt wird (**Annahmenbildung**) und wie sich bestehende Überzeugungen auf der Grundlage von Wahrnehmung ändern (**Annahmenrevision**).

Situationserkennung
extrahiert (strukturierte) Situationen aus den (unstrukturierten) Überzeugungen eines Agenten.

Zielaktivierung
beschreibt, welche der möglichen Ziele eines Agenten in einer Reihe von Situationen derzeit Optionen sind.

Zielplanung
ordnet die aktuellen Ziele des Agenten operativen Grundelementen zu, um sie zu erreichen, d.h. lokale Pläne oder Verhandlungsprotokolle und gemeinsamen Pläne. **Planen bedeutet zu entscheiden, was zu tun ist**.

Ablaufplanung
ist der Prozess des Zusammenführens von Teilplänen für verschiedene Ziele zu einem Ausführungsmodul, bei dem Kompatibilitäten und (z.B. zeitliche und vorrangige) Einschränkungen zwischen verschiedenen Teilplänen berücksichtigt werden. **Ablaufplanung (Scheduling) bedeutet also, zu entscheiden, wann was zu tun ist**.

Ausführung

ist für die korrekte und fristgerechte Umsetzung der in der Planungs- und Terminierungsphase festgelegten Verpflichtungen verantwortlich.

5.20. Belief-Desire-Intentions Architektur

➤ Die BDI Architektur implementiert **Praktische und Prozedurale Schlussfolgerungsagenten**

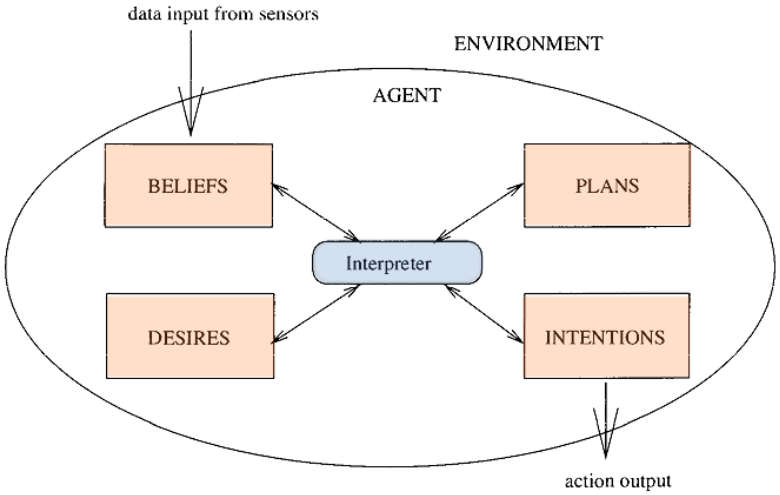


Abb. 31. BDI Architektur und prozedurale Schlussfolgerung über einen Interpreter

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***

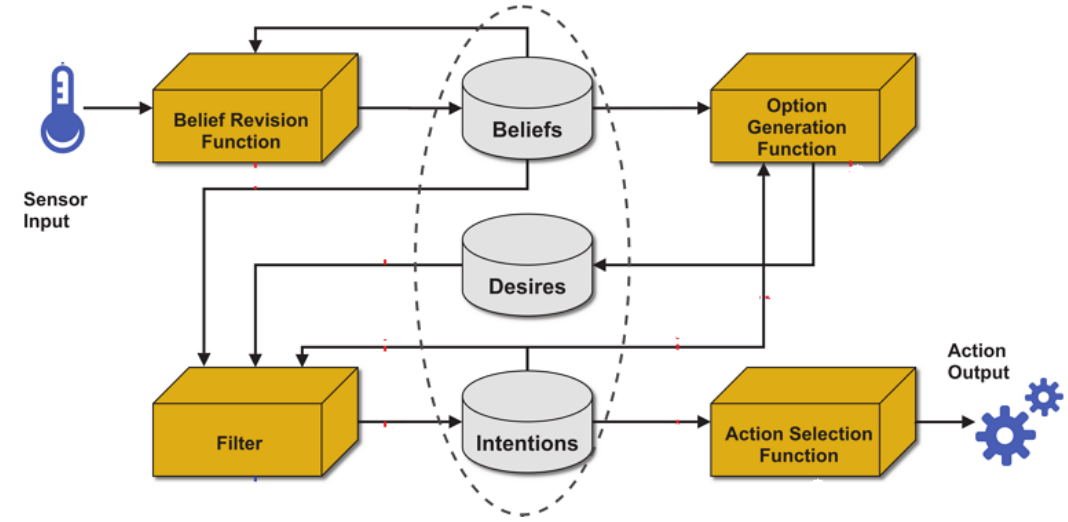


Abb. 32. Verfeinerte BDI Architektur

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

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`)
- ➤ 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 $\rightarrow$ "erreiche P"

$--!P$: Ziel P verworfen

$+B$: Neue Annahme B hinzufügen

$--B$: Annahme B verwerfen

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 ←  lter(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
```

5.21. 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).
```

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**

Definition 10.

```
.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 &in; {tell,untell,achieve,unachieve,
               askOne,askAll,askHow, tellHow,untellHow}
```

- *tell*/*untell*: Hinzufügen/Entfernen von Annahmen
- *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))
```

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").
```

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.
```

5.22. Hybride Architekturen

Die zwei Architekturen symbolische Schlussfolgerungsagenten und reaktive Agenten wurden historisch zu hybriden Architekturen fusioniert:

- 1956–present: Symbolic Reasoning Agents
  - Its purest expression, proposes that agents use explicit logical reasoning in order to decide what to do.
- 1985–present: Reactive Agents
  - Problems with symbolic reasoning led to a reaction against this—led to the reactive agents movement, 1985–present.
- 1990-present: Hybrid Agents
  - Hybrid architectures attempt to combine the best of symbolic and reactive architectures.

6. Verteiltes Rechnen mit JAM Agenten (ABC)

Zusammenfassung der JAM Plattform und Anwendung im Mobilen Crowdsensing

6.1. Motivation

- Crowdsensing ist verteilte Datenverarbeitung
- **Verteilte Datenverarbeitung** bedeutet **verteilte Kommunikation zwischen Prozessen**
- Traditionelle verteilte Netzwerke wie das Internet der Dinge (IoT) müssen mit einer Vielzahl von Kommunikationsprotokollen und Netzwerkstrukturen umgehen können → **Stark heterogene Systeme**
- **Datenrepräsentation** ist eine weitere Hürde in solchen verteilten Systemen
- Häufig HTTP basierte Klienten-Server Kommunikation (zentrale Serverinstanz)

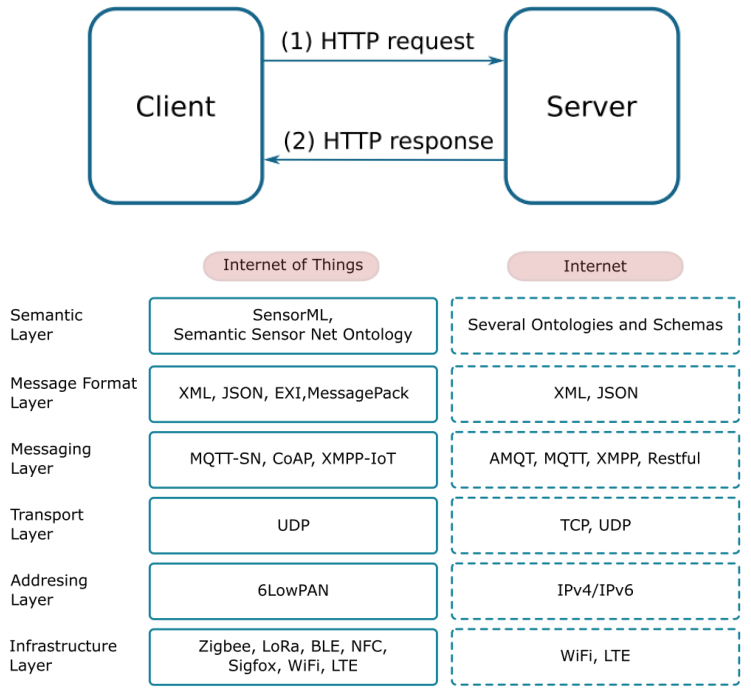


Abb. 33. Große Diversität an IoT Protokollen und Datenrepräsentation für die Kommunikation → nicht einheitlich [4]

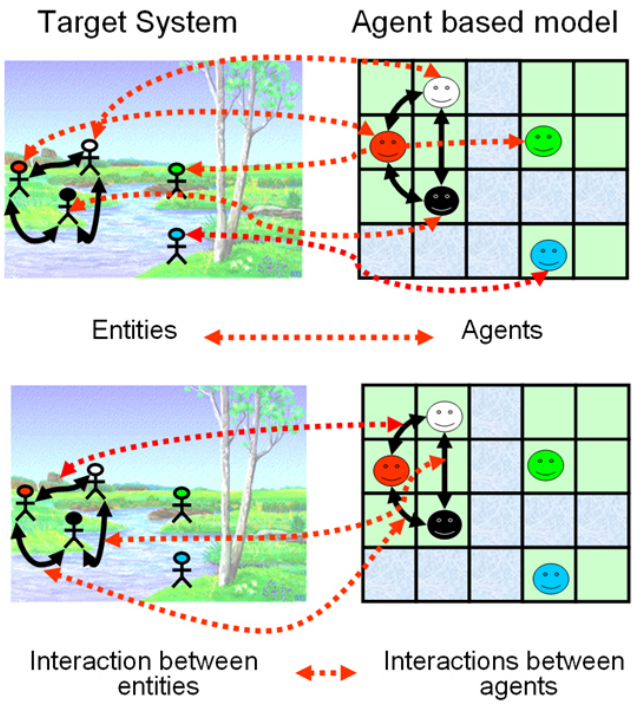
6.2. Agenten

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



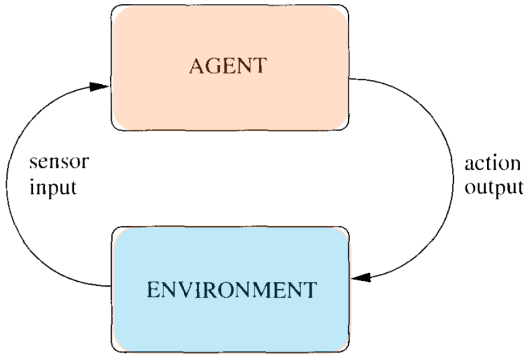
[Galan,2009]

Abb. 34. Beziehung realer natürlicher Welt mit Lebewesen zu virtueller Welt mit Agenten

Agentenmodell

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

Einfachstes Agentenmodell



[J]

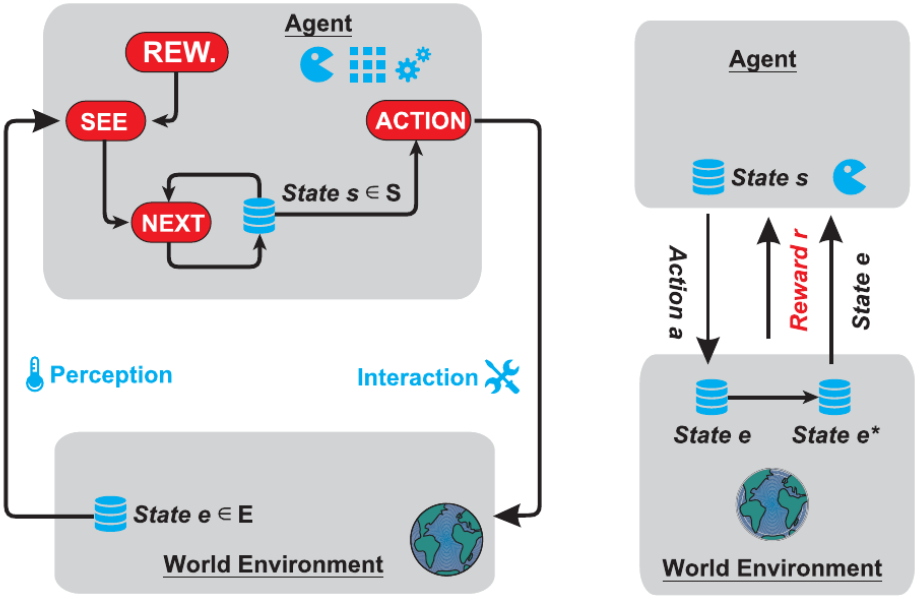
Beispiel

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

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

Reaktive und Zustandsbasierte Agenten

- Reaktive und zustandsbasierte Agenten führen einen Zyklus durch:
  - Perzeption ⇒ Verabeitung ⇒ Zustandsänderung ⇒ Entscheidung ⇒ Aktion



6.3. 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.

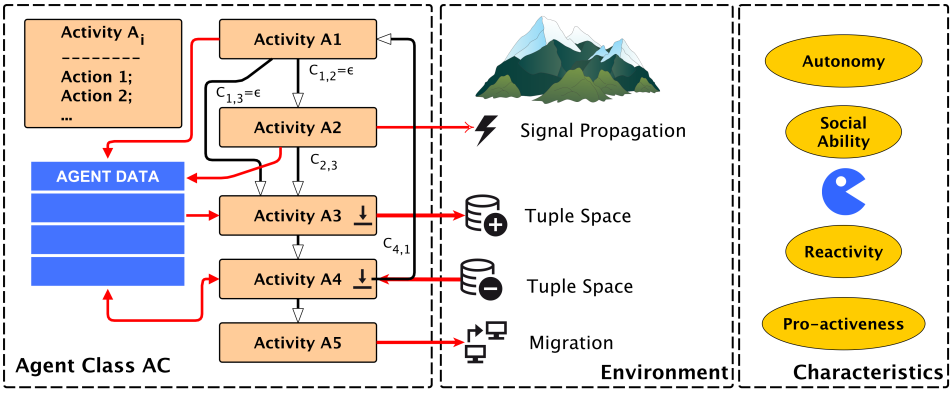


Abb. 35. Links) Agentenverhalten, das von einem Aktivitätsübergangsgraphen vorgegeben ist, und die Interaktion mit der Umgebung, die durch Aktionen erfolgt, die in Aktivitäten ausgeführt werden (Rechts) Agentenmerkmale

- 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 Unterzieles 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**.
- 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

6.4. DATG Modell

- ➤ Ein ATG beschreibt das vollständige Agentenverhalten.
- ➤ Jedes Unterdiagramm und jeder Teil des ATG kann einem Unterklasseverhalten eines Agenten zugeordnet werden.
- ➤ Daher führt das Modifizieren der Menge von Aktivitäten $\mathbb{A}$ und Übergängen $\mathbb{T}$ des ursprünglichen ATG zu mehreren Unter- und Oberverhaltensweisen, die Algorithmen implementieren, um verschiedene unterschiedliche Ziele zu erfüllen.
- ➤ Die Rekonfiguration der Aktivitäten führt zu einer Menge von Aktivitätsmengen $\mathbb{A}^* = \{\mathbb{A}_i \subset \mathbb{A}, \mathbb{A}_j \subset \mathbb{A}, \mathbb{A}_k \supset \mathbb{A}, ..\}$, die von der ursprünglichen Menge $\mathbb{A}$ abgeleitet sind, und die Modifikation oder Rekonfiguration von Übergängen $\mathbb{T}^* = \{\mathbb{T}_1 \subset \mathbb{T}, \mathbb{T}_2 \subset \mathbb{T}, ..\}$ ermöglicht die dynamische ATG-Zusammensetzung (Komposition) und Agentenunterklassifizierung zur Laufzeit,

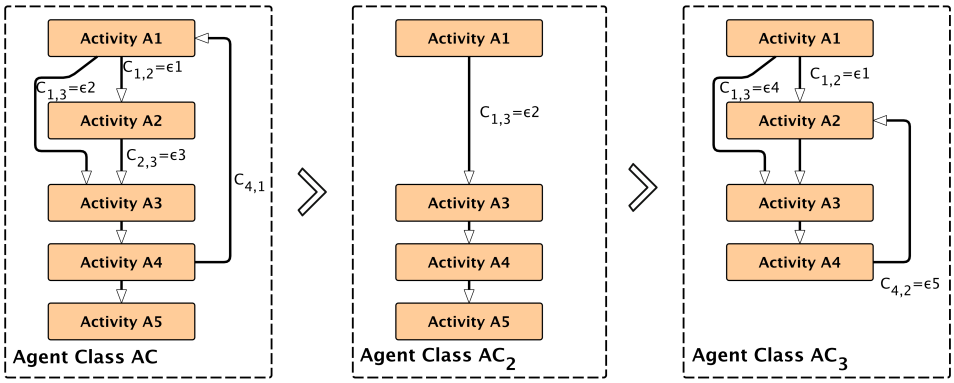
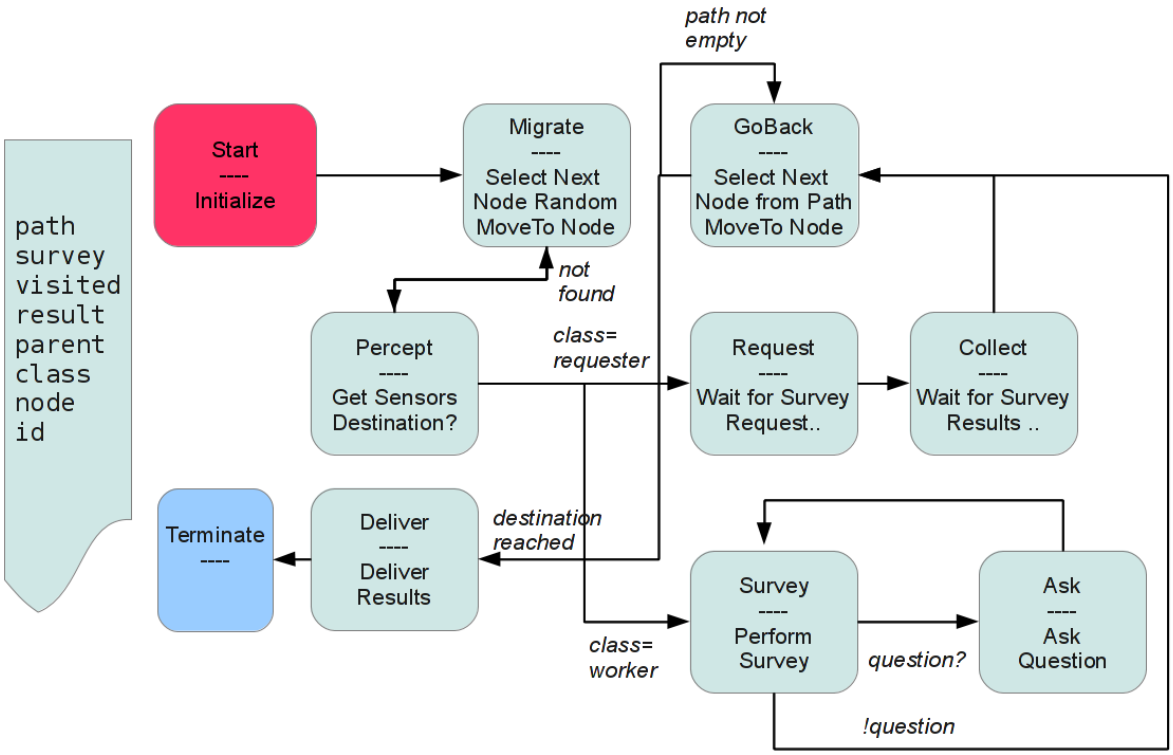


Abb. 36. Dynamischer ATG: Transformation und Komposition

6.5. Beispiel eines Agenten

ATG eines einfachen Umfrageagenten mit Unterklassen (*master*, *requester*, *worker*)



6.6. 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.

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 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.

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

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

Multiagentensysteme

Definition 11.

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 $\mathbf{AC} = \{AC_1, AC_2, ..\}$ bezeichnet werden. Ein Agent gehört zu einer dieser Klassen.

Jede Agentenklasse wird dann durch das erweiterte Tupel $AC = <\mathbb{A}, \mathbb{T}, \mathbb{F}, \mathbb{S}, \mathbb{H}, \mathbb{V}, \ℙ>$ 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 ℙ die Menge der Parameter, die von der Agentenklasse verwendet werden.

Definition 12.

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.

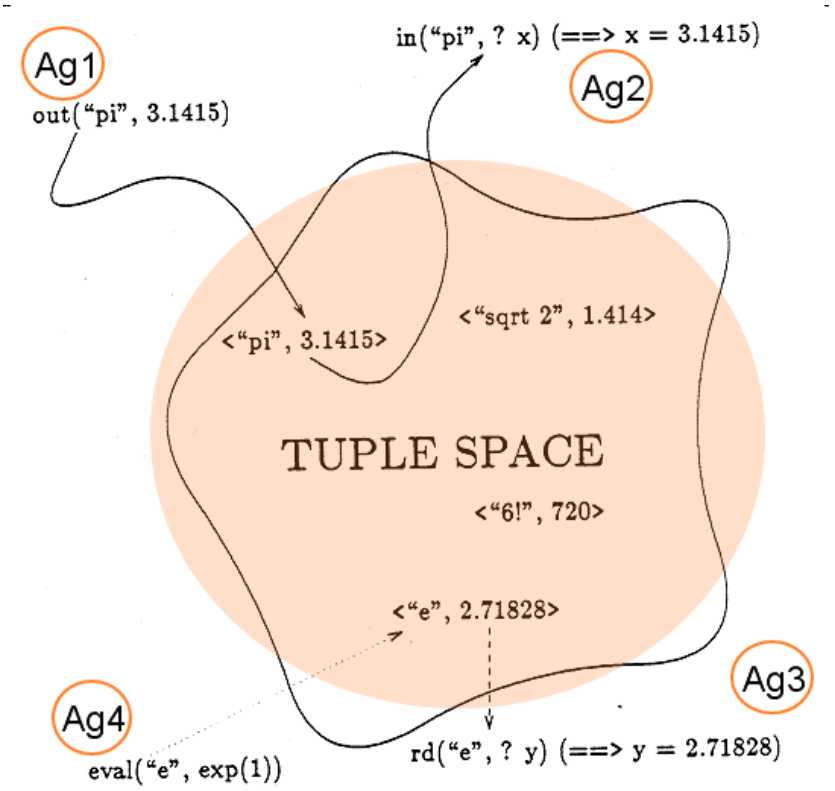
6.7. Tupelräume

- 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].



[3]

Abb. 37. Ein Schnappschuss eines Tupelraumes mit Tupeln und Tupeloperationen

➤ Kommunikation von Agenten über Tupelräume ist eine **Koordinationssprache**.

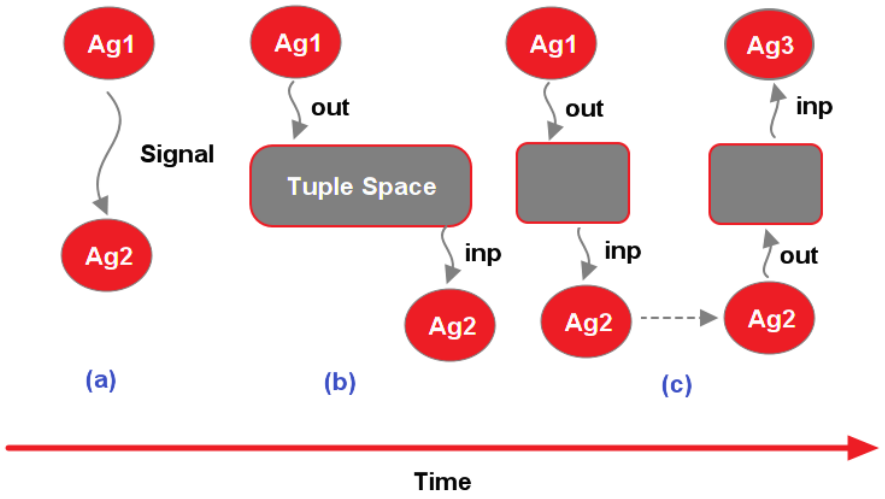


Abb. 38. Direkter Nachrichtenaustausch (a), z.B. durch Signale, im Vergleich zu generativer Kommunikation (b) und virtuelle verteilte Räume (c) durch mobile Prozesse (Agenten)

6.8. 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.

6.9. 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

`function (t:tuple)`
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.

inp

`function (p:pattern, callback: function (tuple|tuple[]|null),all?:boolean)`
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.

rd

`function (p:pattern, cb:function (tuple|tuple[]|null),all?:boolean)`
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

```
out(["Sensor",1,100]);
out(["Sensor",2,121]);
inp(["Sensor",1,_], function (t) { if (t) v=t[2]; });
inp(["Sensor",_,_], function (t) { if (t) n=t[1],v=t[2] });`
rd(["Sensor",_,_], function (t) { if (t) n=t[1],v=t[2] });`
```

try_inp, try_rd

`function (0,p:pattern,callback?:function,all?:boolean)` → `tuple|tuple[]|null`

Nichtblockierende Version von *inp/rd*. Wird kein passendes Tupel gefunden wird die Operation ergebnislos terminiert.

try_inp, try_rd

`function (tmo:number,p:pattern,callback:function,all?:boolean)` → `tuple|tuple[]|null`
Teilblockierende Version von *try_inp/try_rd*, Wird innerhalb einer Zeit von *tmo* kein passendes Tupel gefunden wird die Operation ergebnislos abgebrochen.

➤ 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?)

test

`function (pattern)` → `boolean`
Nicht blockierender Test eines Tupels.

ts (testandset)

`function (pattern,function (tuple)` → `tuple)`
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.

rm

`function (p:pattern,all?:boolean)`
Entfernung eines oder aller passenden Tupel.

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

`function (tmo:number,t:tuple)`
Ausgabe eines Tupels t mit einer Lebenszeit τ (im Tupelraum).

alt

`function (pattern [], function (tuple|tuple[]|null),all?:boolean)`
Gleichzeitige Mehrfachabfrage (mit *inp* Semantik) von einer Menge von Mustern. Das erste Tupel welches einem der Muster entspricht wird an die Rückruffunktion übergeben. Es gibt auch eine nicht- oder partiell blockierende Variante *try_alt*.

6.10. Tupelräume - Produzenten und Konsumenten

Produzent

```
this.act = {
 percept : function () {
   out(["SENSORNODE"]);
   mark(1000,["SENSOR","CLOCK",time()])
   mark(1000,["SENSOR","GPS",
              {lati:x,long:y}])
 }
}
```

Konsument

```
this.act = {
 process : function () {
  if (test(["SENSORNODE"])
   inp(["SENSOR","CLOCK",_], function (t) {
    if (t) log('It is time '+t[2]);
   })
 }
}
```

Aufgabe

Betrachte das zeitliche Verhalten im obigen Beispiel (Laufzeiteigenschaften)

1. Was könnte im Produzenten-Konsumenten System möglicherweise falsch laufen?
2. Wie müsste der Agentencode geändert werden um Laufzeitfehler zu vermeiden?

6.11. Signale

- *Tupleraumkommunikation* ist
  - Anonym und
  - Generativ (d.h. die "Nachricht" kann eine längere Lebensdauer als der Absender besitzen)
- Eignet sich daher für die Kommunikation von lose gekoppelten Ensembles
- Neben diesen lose gekoppelten Gruppen gibt es durch Replikation von Kindagenten Gruppen mit starker Kopplung und Kenntnis der Identiäten
  - Elternagent erfährt die Identität eines Kindagenten beim Forking
  - Kindagent kann die Indentität des Elternagenten durch die Operation `myParent()` erfahren.
- **Signale sind zielgerichtete Nachrichten mit identifizierbaren Absender und Empfänger** (wobei es bei Multi- und Broadcastnachrichten viele geben kann)
- Ein Signal *signal(arg)* besteht aus einem Signalnamen oder einer Signalnummer mit einem optionalen Argument.
- Signale müssen auf Empfängerseite mit einem Signalhandler im `this.on` Abschnitt des Agentenmodells eingerichtet werden.

send

`function (target:id,signal:string|number,arg?:*)`
Sendet eine Signalnachricht *signal(arg)* an den Empfängeragenten *target*.

broadcast

`function (ac:string,r:number,signal:string|number,arg?:*)`
Sendet eine Signalnachricht *signal(arg)* an alle Empfängeragenten der Klasse *ac* im Radius *r*.

this.on["signal"]

`function (arg,from)`
Ein Signalhandler für das Signal *signal*.

Produzent

```
this.act = {
  a1 : function () {
    this.child = fork()
  },
  a2 : function () {
    send(this.child,'ALARM',100)
  }
}
```

Konsument

```
this.on = {
  ALARM : function (arg,from) {
    log('Got ALARM '+arg+' from '+from)
  }
}
```

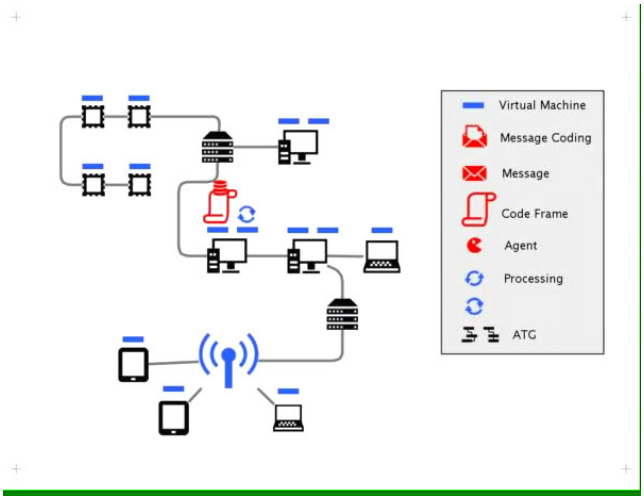
6.12. Mobile Agenten

- Ein Agent in Ausführung wird durch einen Prozess repräsentiert.
- Der Prozesszustand setzt sich zusammen aus:
  - Körpervariablen (Datenzustand)
  - Nächste Aktivität und Mikroschedulingblöcke (Kontrollzustand)
  - Signale, Nachrichten
  - Pfad im Netzwerk (Plattformen auf denene der Agent bereits ausgeführt wurde)
  - Verhalten (D.h. ATG)!
- **Mobile Agenten bedeuten mobile Prozesse!**
- Bei der Migration muss ein Agentenprozess in einen
  1. **Prozessschnappschuss** (Container) eigenfroren und serialisiert werden,
  2. Der Schnappschuss in textueller Form (JSON+) von einer Plattform zu einer anderen übertragen werden, und

3. Der Prozessschnappschuss wieder deserialisiert werden, und der Prozess wird weiter ausgeführt.

Mobiler Code

➤ Anstelle von mobilen Daten (Kommunikation) wird durch Agenten mobiler Code in Netzwerken transportiert



➤ Eine Agentenplattform repräsentiert einen physikalischen Knoten in einem Virtuellen Netzwerkgraphen

- ❏ Es gibt virtuelle und physikalische Kommunikationsverbindungen zwischen den Netzwerkknoten
- ❏ Beispiel: Internet Protokoll (IP/UDP/HTTP)

➤ *Ports* von Plattformen verbinden und stellen *Links* her

➤ Ein Agent kann auf einen anderen Knoten (Plattform) migrieren indem er die *moveto* Operation ausführt (innerhalb einer Aktivität):

- ❏ Dabei muss der Agent als Argument das Ziel angeben
- ❏ Ziel kann ein Knotenname (`DIR.NODE(<nodename>)`, eine IP Adresse (`DIR.IP(<url>)`), oder bei gerichteten Verbindungen die Richtung sein (z.B. `DIR.NORTH`)

➤ Alle aktuell mit einer Plattform verbundenen anderen Plattformen (Namen) können mit der *link* Funktion erfragt werden:

```
{
  var connected=link(DIR.IP('%'));
  if (connected.length) {
    var next = random(connected);
    moveto(DIR.NODE(next));
  }
}
```

7. Agentenplattformen

7.1. JavaScript Agent Machine

- ➤ Vollständig in JavaScript programmiert
- ➤ Portabel:
  - ❏ Web Browser
  - ❏ Cordova WebKit App
  - ❏ CLI mit nodejs/jxcore/jerryscript
  - ❏ Eingebettet in Java Programm
- ➤ Schlank: ca 1M Byte JS Code
- ➤ AgentJS Code kann direkt ausgeführt werden (nur Sandboxing erforderlich)!
- ➤ Eine physische JAM kann eine Vielzahl von virtuellen JAM Knoten ausführen (nur sequenzielles Scheduling)

AIOS

- ➤ Das AIOS kapselt eine Vielzahl von Modulen und stellt eine API für Agenten 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
- ❏ SECU: JAM Capabilities und Sicherheit
- ❏ WATCHDOG: Faires Agentenscheduling durch Laufzeitüberwachung (Time slicing)
- ❏ ML: Machine Learning → Fokus mobile Algorithmen und Modelle
- ❏ SAT: SAT Logic Solver → Knowledge Base

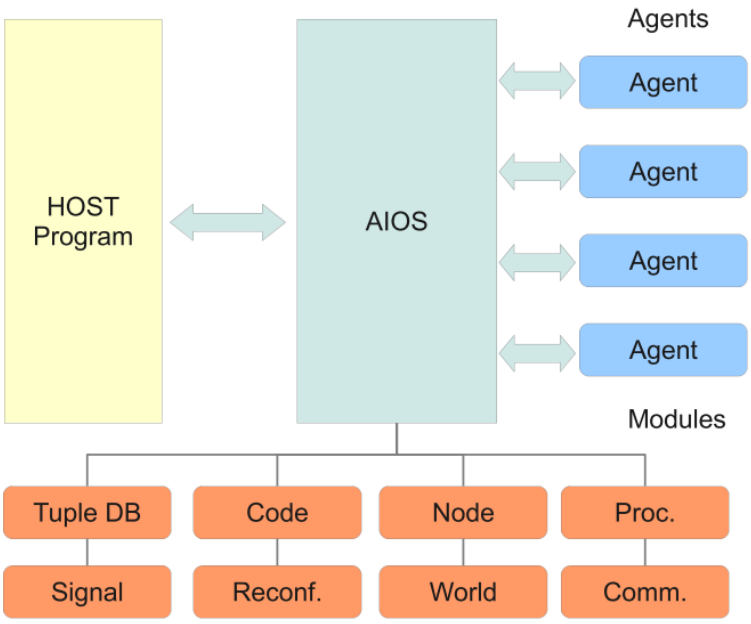


Abb. 39. Das Agent Input-Outut System als Schnittstelle zwischen Agenten und JAM und einer Hostplattform (bzw. Applikation)

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.
- 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.

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

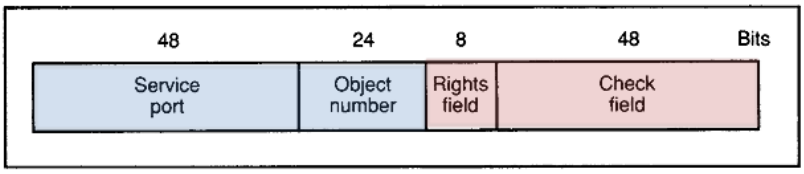


Abb. 40. Aufbau einer Capability: Der Serverport bindet die Capability an einen Service, die Objektnummer ist optional und kann eine Unterklasse des Service oder der zu schützenden Ressource darstellen, das Rechtefeld kodiert die möglichen Operationen der Serviceklasse, und das Schutzfeld (enthält Rechtefeld nochmals verschlüsselt) schützt die Capability gegen Manipulation.

Schutz von Capabilities

- ➤ Das Rechtefeld ist zentral da es die durch die Capability erlaubten Operationen angibt.
  - ❏ CPU Zeit erhöhen
  - ❏ Änderung des Privilegienlevels
  - ❏ Migration usw.
  - ❏ Zugriff auf Sensoren und persönliche Nutzerdaten
- ➤ 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 verschlüsselt.
  - ❏ Nur der Service / Agentenplattform kennt den privaten Schlüssel

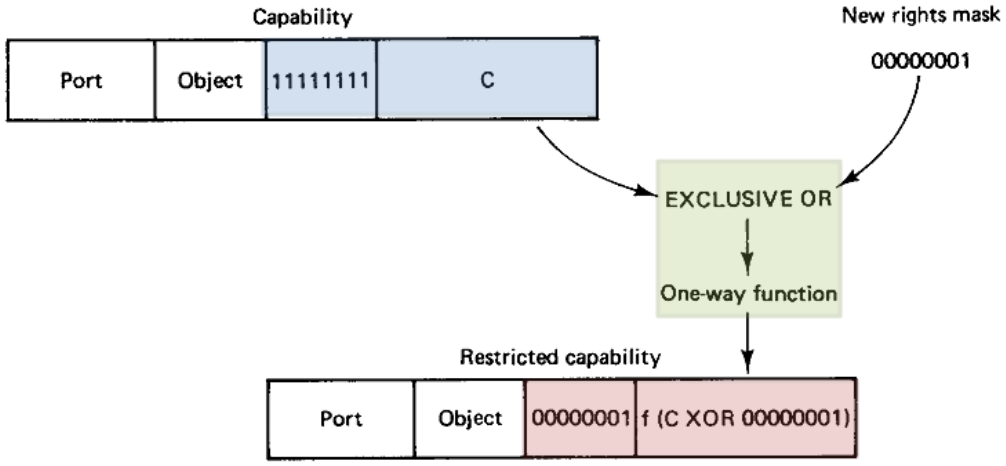


Abb. 41. Erzeugung einer öffentlichen restriktiven Capability aus einer privaten nicht eingeschränkten (enthält privaten Schlüssel C) mittels One-way Verschlüsselungsfunktion $f(C \text{ xor } R)$

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

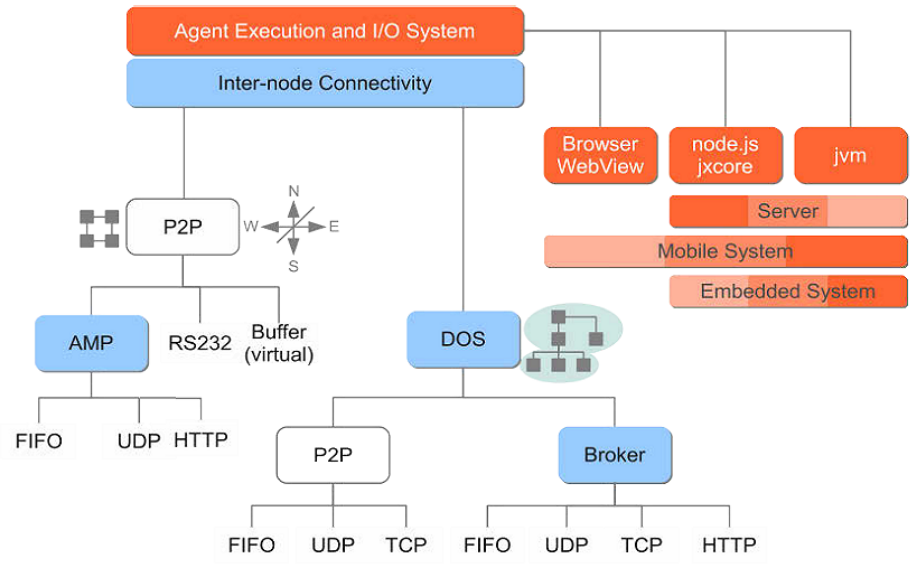


Abb. 42. JAM Konnektivität und eine breites Spektrum an unterstützten JavaScript Host Plattformen

Netzwerkstrukturen

- JAM Platfformen können über eine oder mehrere virtuelle Kanäle (AMP) miteinander verbunden werden
  - Verwendung von Relaisstationen führt zu sternartigen Kommunikationssnetzwerken
  - Browser und mobile JAM Knoten können nur an im Internet sichtbare JAM Knoten über HTTP sich verbinden → erfordert Relais Knoten (ohne Nutzerinteraktion, kopflos)

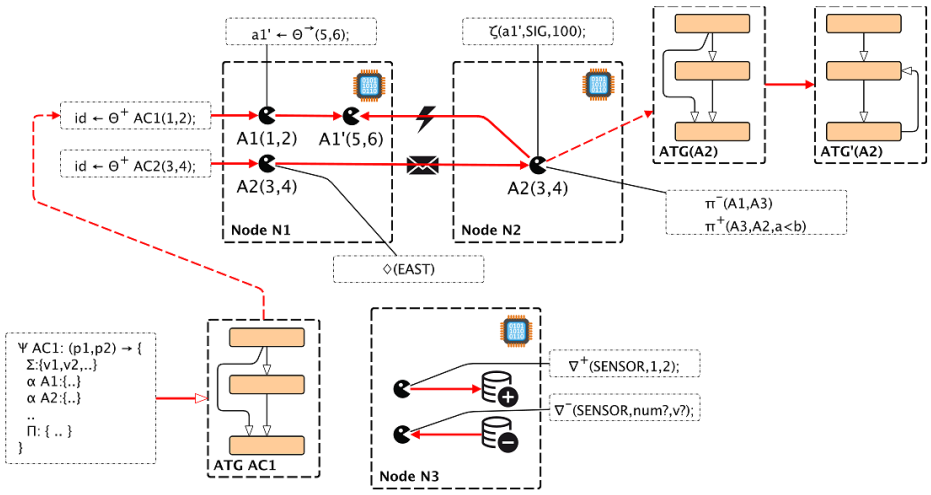


Abb. 43. Effekt von verschiedenen AAPL/AgentJS Anweisungen zur Laufzeit auf Agenten und Plattformen

8. Programmierung

8.1. JavaScript :: Daten und Variablen

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

  $\mathbf{T}_{\text{core}}$=`{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;
```

8.2. 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,..)
```

- 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 (pi) { ϵ(pi) }
array.forEach(function(elem,index) { ϵ(pi) }
```

8.3. 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
```

➤ 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"]
```

8.4. 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.

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

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

8.5. AgentJS

- AgentJS ist die JavaScript Implementierung von AAPL (Activity-based Agent Prorgramming Language)
  - 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
- 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 13. (*Konstruktorfunktion: Template einer Agentenklasse*)

```
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;
}
```

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())`)

Beispiele

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

Asynchrone Operationen

➤ Da JavaScript Funktionen als Werte erster Ordnung behandelt werden häufig sog. Rückruffunktione (callback) an Funktionen als Argumente übergeben, die von der aufrufenden Funktion dann aufgerufen werden

➤ Rein synchrone Operationen (z.B. Tupelraumoperation *out* oder jegliche Berechnung, auch mit Rückruffunktionen wie *iter*) können in einer Sequenz innerhalb einer Agentenaktivität ausgeführt werden

➤ Es gibt aber auch asynchrone Operationen ((z.B. Tupelraumoperation *rd*), die zwar aufgerufen werden, aber erst später (ggfs. nach Durchlauf einer Aktivität) ausgeführt werden.

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!

➤ Es gibt drei verschiedene Blockkonstruktoren:

- ❏ B: Ein sequenzieller Scheduling Block
- ❏ L: Ein iterativer Schleifenblock
- ❏ I: Ein Iteratorblock für Arrays und Objekte (Strukturen)

Definition 14. (*Mikroaktivitäten und Blöcke*)

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

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 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 test (pattern) → boolean
function ts (pattern,callback:function(tuple) → tuple)
function alt.try inp.try rd.try (tmo:number,..)
```

➤ *collect*, *copyto*, und *store* sind verteilte Tupelraumoperationen und wirken

auf entfernten Tupelräumen

Beispiele

```
out(['MARKING1',1]);
out(['SENSORA',100,true]);
inp(['SENSORA',_,_],function (tuple) {
  if (tuple) this.s =tuple[1];
});
rm(['SENSORA',_,_],true);
rd.try(0,['SENSORA',_,_],function (tuple) { .. });
// oder bei Timeout 0 alternativ ohne callback =>
var t = rd.try(0,['SENSORA',_,_]);
ts(['MARKING',_],function (t) { t[1]++ });
alt([
  ['SENSORA',_,_],
  ['SESNORB',_],
  ['EVENT'],
],function (tuple) {
  if (tuple && tuple[0]=='EVENT') {..}
  else ..
});
```

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 (path:string) , NODE(id:string),
            IP (ip:string) ,
            CAP {cap:string}
} : dir
```

➤ Signalhandler werden außerhalb von Aktivitäten in der `this.on` Sektion ausgeführt. Der Objektattributname ist der Signalname.

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 (path:string) , NODE{node:string),
            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.
- ➤ Als Ziel kann auch ein JAM Knotenname angegeben werden (`DIR.NODE("name")`).

moveto

`function moveto (to:dir)`
Der Agent wird eingefroren, zu der neuen Plattform gesendet, und dort weiter ausgeführt. Die Rcihtung kann einen geometrischen Bezug haben (z.B. DIR.NORTH), oder eine Plattformreferenz beschreiben (z.B. DIR.NODE('hanaqude')). Eine geometrische Richtung macht nur in Maschennetzwerken it P2P Verbindungen Sinn.

- ➤ Wenn keine Verbindung in die angegebene Richtung existiert wird der Agent terminiert!

link

`function link (dir) → boolean|string|[]`
Liefert Informationen über aktuell erreichbare Nachbarknoten. 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 (Routes) zurück (Aufruf mit Argument `DIR.IP('*')`).

- ➤ Die von der aktuellen Plattform (JAM Knoten) erreichbaren JAM Knotennamen können z.B. mit der `link(DIR.IP("%"))` Operation als Liste ausgegeben werden.
- ➤ 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!

8.6. Simulation

➤ Neben der "realen" Ausführung von mobilen Agenten in Computernetzwerken stellt die Simulation von verteilten Netzwerken aus Agentenplattformen und "virtuellen" mobilen Agenten ein wichtiges experimentelles Werkzeug dar um auch komplexe Szenarien raum- und zeitaufgelöst zu untersuchen.

SEJAM

➤ SEJAM ist das *Simulation Environment for JAM* und baut auf einer "physischen" JAM Plattform auf

- ❏ JAM wird um eine Bedieneroberfläche und Visualisierung ergänzt
- ❏ Eine virtuelle Welt besteht aus einer Vielzahl virtueller (logische) JAM Knoten und eier geometrischen zweidimensionalen Welt
- ❏ Agenten können in der virtuelen Welt zwischen virtuellen Knoten migrieren

➤ Die virtuellen Knoten sind in der geometrischen Welt mobil

➤ Die virtuellen Knoten verfügen über virtuelle Kommunikationsverbindungen (z.B. radial wie Mobilfunk)

➤ Die JAM Plattform in SEJAM kann mit realen Netzwerken gekoppelt werden!

➤ Mobile Agenten können nicht nur zwischen virtuellen Knoten migrieren, sondern auch die auf externe Plattformen in der realen Welt migrieren

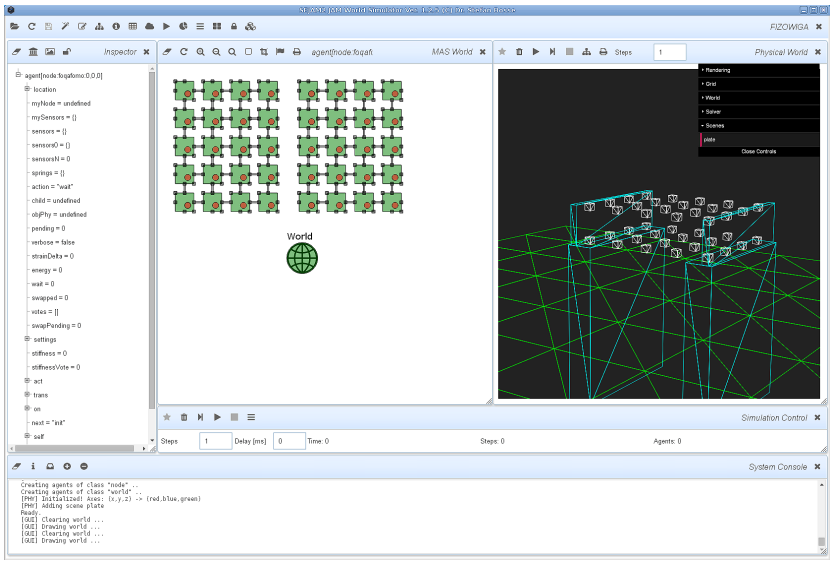


Abb. 44. SEJAM2 mit zweidimensionaler Plattformwelt (stationäre und mobile Knoten) und Agenten; Kreise um Knoten symbolisieren Funkkommunikation

Augmented Virtuality

- ➤ Die Fusion von virtuellen Simulationswelten mit realen Netzwerken ermöglicht die Kopplung von virtuellen Welten mit realen Welten und Interaktion mit Menschen!
- ➤ Chat bots können sowohl in virtuellen als auch realen (menschbezogenen) Welten agieren!
- ➤ Das ermöglicht das Studium komplexer sozio-technischer Systeme

Human-in-the-loop simulation for Augmented Virtuality besteht daher aus:

- ➤ Virtueller Simulationswelt und Framework SEJAM2 mit JAM platform unf virtuellen JAM Netzwerken die mit dem Internet verbudnen sind
- ➤ Mobilen Gertäten mit JAM Plattform verbunden mit dem Internet oder via Baken (Bluetooth z.B.)
- ➤ Stationäre Geräte (Sensoren, IoT, und Baken) die über das Internet mit der virtuellen Welt verbunden sind

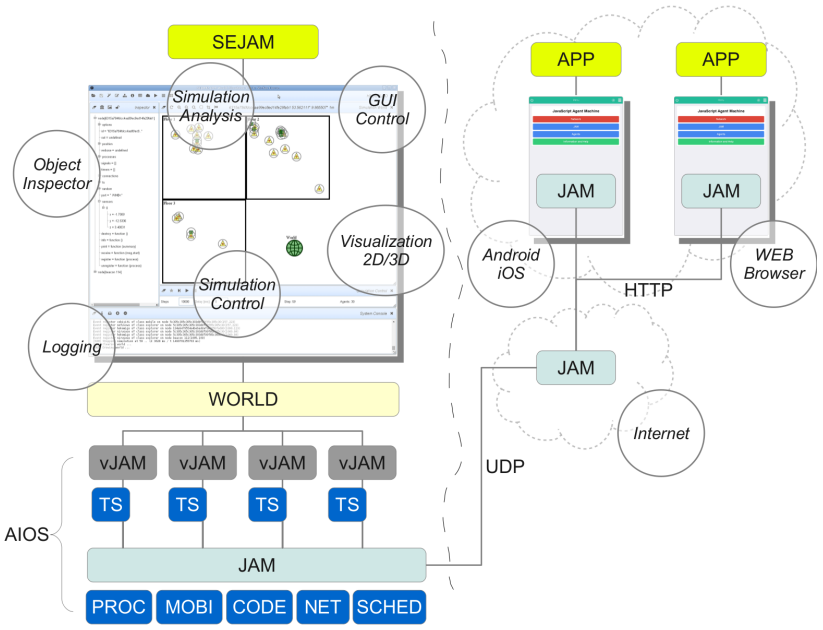


Abb. 45. Augmented Virtuality: Human-in-the-loop Simulation mit mobilen Agenten

9. Demonstrator

9.1. Smart City: Self-Organizing Light Control

Goal

Simulation and investigation of Crowd Interaction with Smart Cities

- ➤ Self-organized control of ambient light conditions (e.g., in streets or buildings) using Crowd Sensing

Virtual World

- ➤ World consists of streets and buildings
  - ❏ Beacons placed in buildings and beside streets
  - ❏ Smart Light Devices illuminating streets and buildings

- **Agents** (can) represent:
  - Technical devices → Light Control
  - Network Infrastructure and Communication
  - Computational units and Smart Controllers
  - Chat Bots performing Crowd Sensing
  - Robots
  - Artificial Humans (Artificial Crowd)

Real World

- Mobile Devices using a WEB App with Chat Bot dialog
  - Users interact with remote agents via a chat dialog;
  - Remote agents investigate user location and an assessment of satisfaction of ambient light situation.
- Additionally, the device sensor data is collected (light, position, ..) if available

Self-organizing Light Control

- Beacons sent out mobile *explorer agents* performing question-answer dialogs on mobile devices (random walk)
- Based on collected *crowd data*:
  - The light conditions in buildings and streets should be adapted (darker or brighter);
  - Addressing 1. Crowd demands 2. Energy Saving.
- If action is required, mobile *notification agents* are sent out to neighbouring nodes to change light intensity based on:
  - Directed diffusion
  - Random walk
  - Divide-and-Conquer

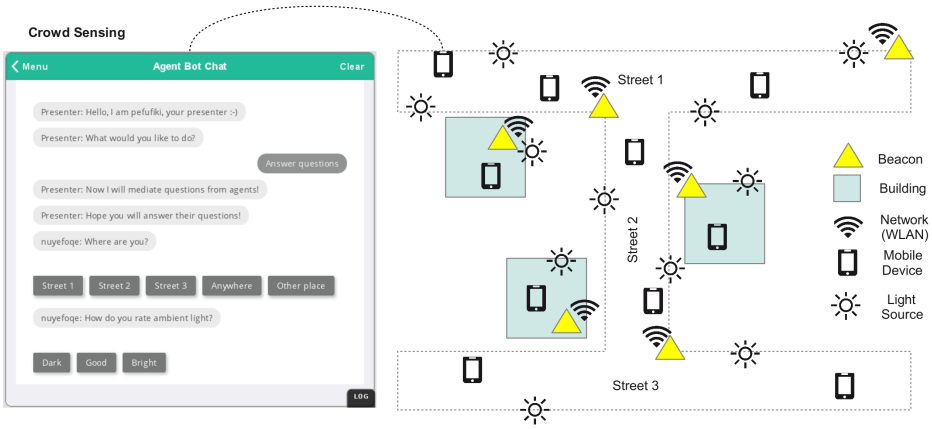


Abb. 46. Simulation world connected to the Internet performing Crowd Sensing → Real world is mapped on Virtual World!

9.2. Conclusions

1. Fusion of real and virtual worlds can contribute to investigate complex socio-technical systems with ensemble sizes beyond Millions of entities (users, devices, machines)
2. Using agent-based modelling and simulation with mobile agents create an unified model and representation for:
   - ➤ Artificial humans
   - ➤ Chat Bots
   - ➤ Devices and Machines
   - ➤ Distributed Computation
3. Augmented Virtuality means the integration and tight coupling of ABM simulation with Crowd Sensing and technical devices interaction
4. Issues:
   - ➤ Time scales in real and virtual worlds
   - ➤ Spatial scales and spaces in real and virtual worlds
   - ➤ Mapping of real on virtual worlds

10. Kommunikation und Interaktion

Agent-Agent und Agent-Welt Kommunikation

10.1. 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.
- 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!

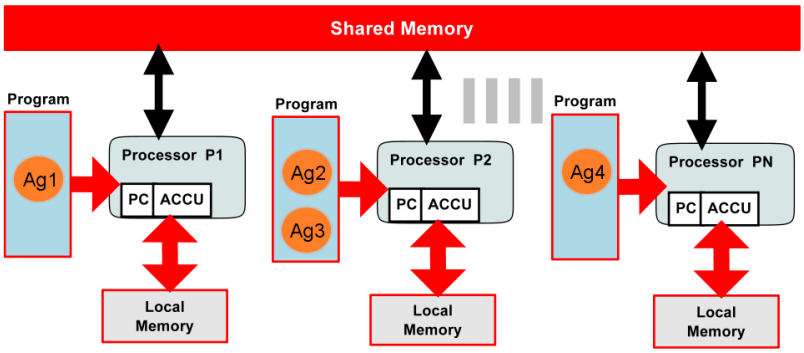
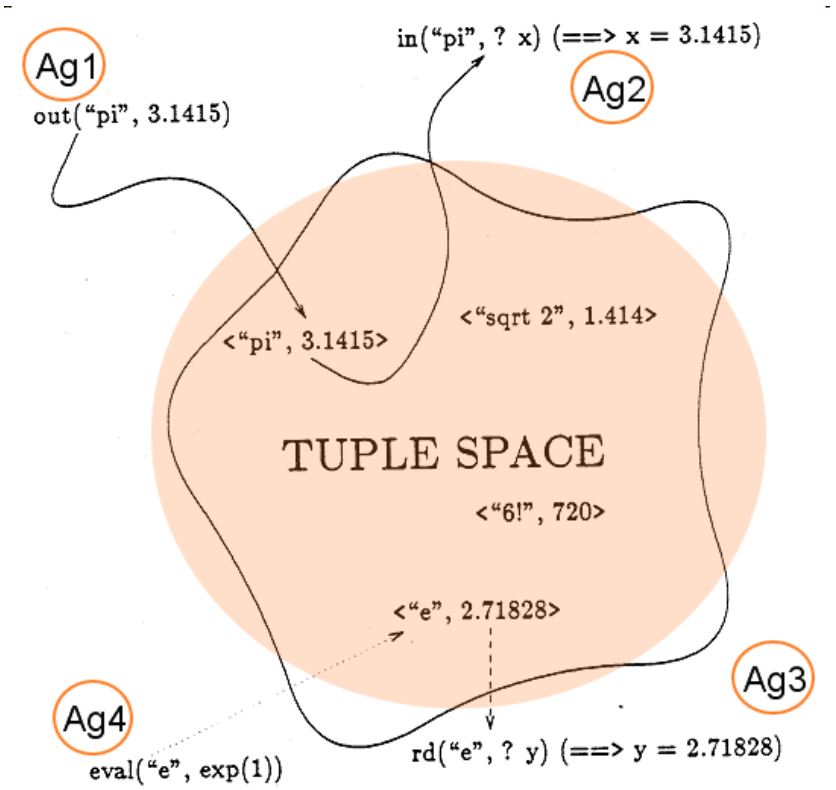


Abb. 47. Das PRAM Modell und die Verknüpfung mit Agenten die auf den PUs ausgeführt werden.

10.2. 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].



[3]

Abb. 48. Ein Schnappschuss eines Tupelraumes mit Tupeln und Tupeloperationen

➤ Kommunikation von Agenten über Tupelräume ist eine **Koordinationssprache**.

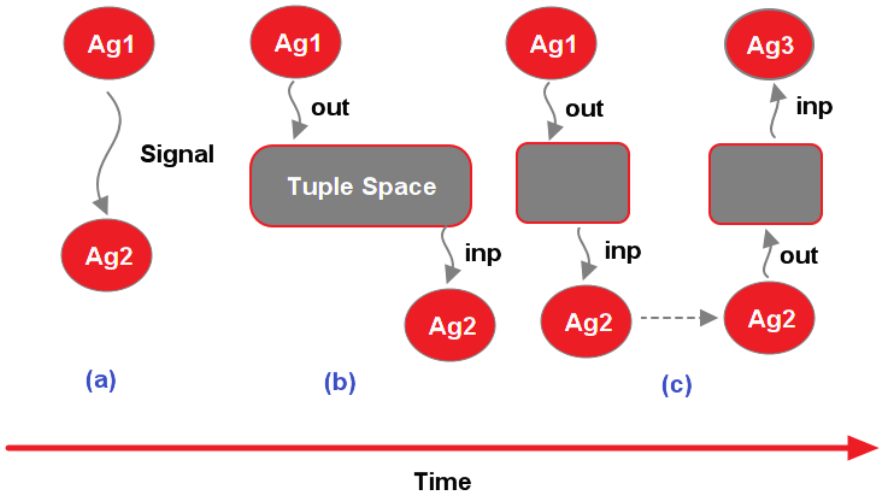


Abb. 49. Direkter Nachrichtenaustausch (a), z.B. durch Signale, im Vergleich zu generativer Kommunikation (b) und virtuelle verteilte Räume (c) durch mobile Prozesse (Agenten)

10.3. 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.

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):

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

- 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 (⊥) 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.
- 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 15.
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 = ⊥, $1 \leq i \leq$ 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.

10.4. 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);`

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?);`

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?)

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).

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 :: $= v \mid \epsilon \mid x \mid f\ (x)$ 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);
```

10.5. 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.

10.6. Tupelräume - Beispiele

```
out(['SENSOR',10,20]);
out(['SENSOR',9,23]);
out(['PI',3,14]);
out(['DATA',[1,2,3,4]]);
inp(['SENSOR',_,_]);
>> [ 'SENSOR', 9, 23 ]
inp(['SENSOR',_,_]);
>> [ 'SENSOR', 10, 20 ]
inp(['SENSOR',_,_]);
>> null
rd([_,_])
>>[ 'DATA', [ 1, 2, 3, 4 ] ]
```

10.7. 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!

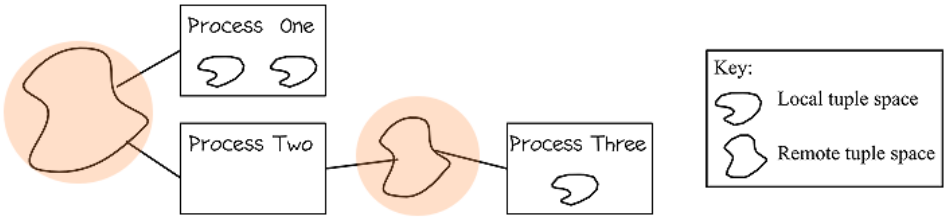


Abb. 50. Zusammenhang von lokalen und entfernten (verteilten) Tupelräumen

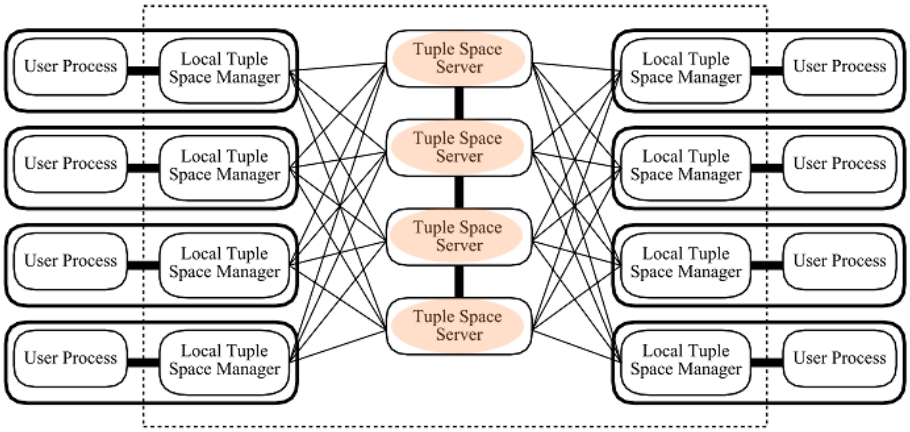


Abb. 51. Lokale und multiple globale Tupelraumserver

10.8. 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!

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

```
// Handler
process.on('SIGNAL1',function (arg) { log('Handle signal 1 with '+arg) });
// Emitter
process.emit('SIGNAL1','fun');
>>[JAM] Handle signal 1 with fun
process.emit('SIGNAL2','nofun')
process.emit('SIGNAL1','nofun')
>>[JAM] Handle signal 1 with nofun
```

- 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

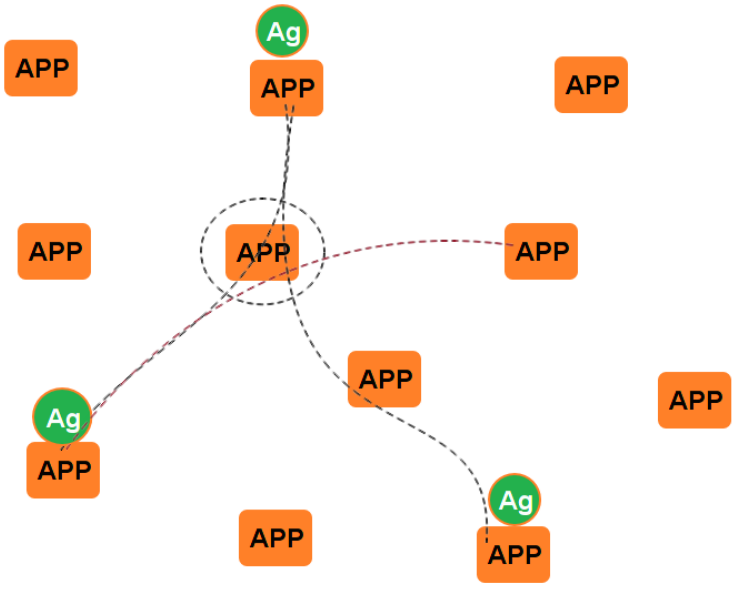


Abb. 52. Mobile Agenten in einem Netzwerk aus Plattformen (APP) und ihrer Spuren; Routing von Nachrichten über Kreuzungspunkte von Spuren

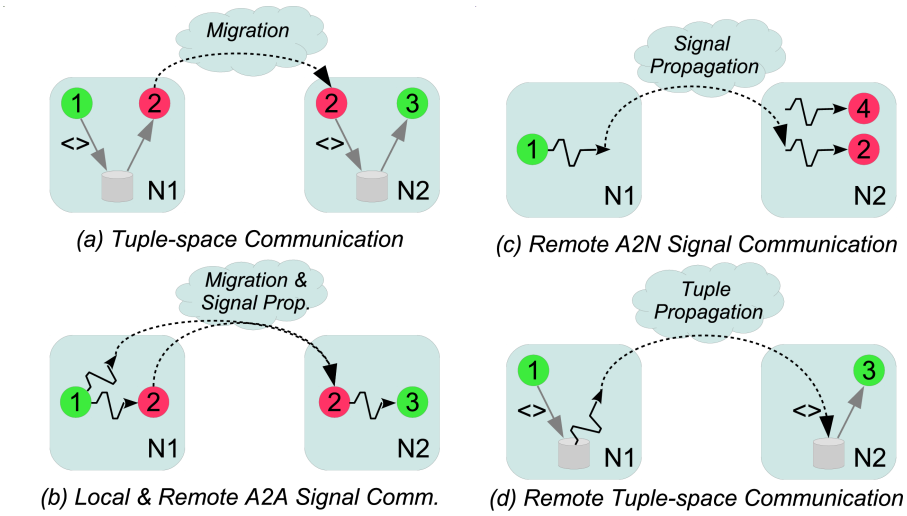


Abb. 53. Agentenkommunikation (a) Lokale Tupleräume (b) Agent-Agent Signale (c) Entfernte Agent-Knoten Signale (d) Entfernte Tupleraumoperationen

10.9. Höhere Kommunikation: Interaktion

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

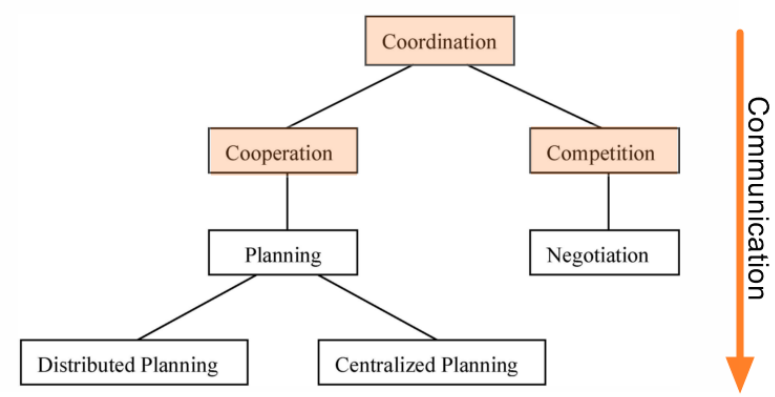


Abb. 54. Taxonomie der Agenteninteraktion die auf Kommunikation aufbaut

10.10. 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)

10.11. 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.

10.12. 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. 55. 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. 56. KQML Nachrichtentypen (Performativen): Teil 2 [C]

➤ 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. 57. 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 :onto1ogy motors
 : reply-with q1 :content (val (torque ml)))
(reply
 :sender B :receiver A
 :language KIF :ontology motors
 :in-reply-to ql :content (= (torque ml) (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 ml) (scalar 12 kgf)))
(tell
 :sender B :receiver A
 :in-rep1y-to q1 :content (= (status ml) normal))
(eos
 :sender B :receiver A :in-reply-to q1)
```

10.13. 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. 58. FIPA Nachrichtentypen (Performativen) und ihre Anwendungsgebiete [C]

Implementierung

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

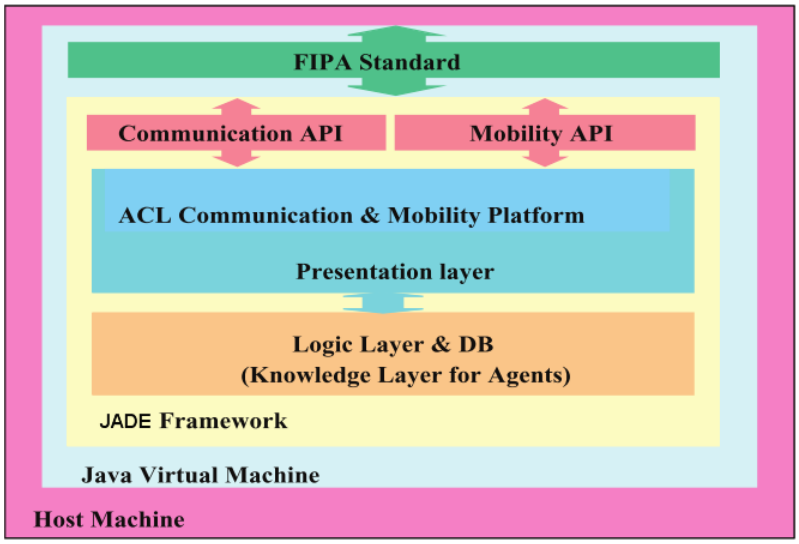


Abb. 59. Schichtenaufbau der APP mit FIPA-ACL über der Wissensdatenbank und Repräsentationsschicht [D]

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

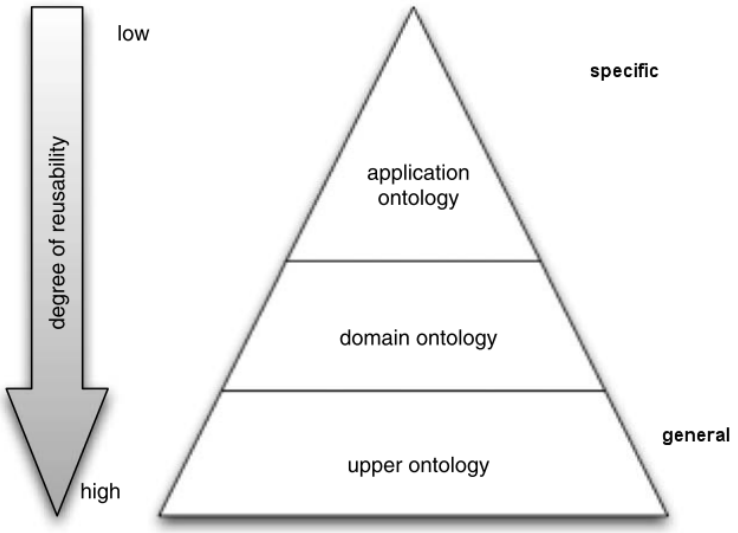


Abb. 60. Die Abstraktion von Ontologien bestimmt ihren Nutzen und eine Hierarchie [B]

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 besitzen unterschiedliche Ausdrucksstärke, je nachdem ob sie

eher informal oder formal sind.

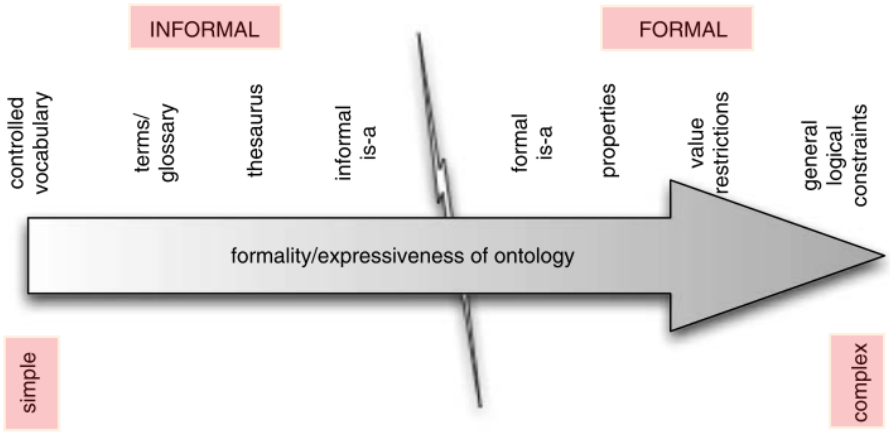


Abb. 61. Spektrum der Ontologien [B]

11. Programmiermodelle und Programmiersprachen

11.1. Modellierung von Agenten mit Programmiersprachen

➤ Es gibt eine Vielzahl von prozeduralen und deklarativen Programmiersprachen um

- ❏ Perzeption, Berechnung, Aktion, Planung und
- ❏ Interaktion

zu beschreiben.

11.2. 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.

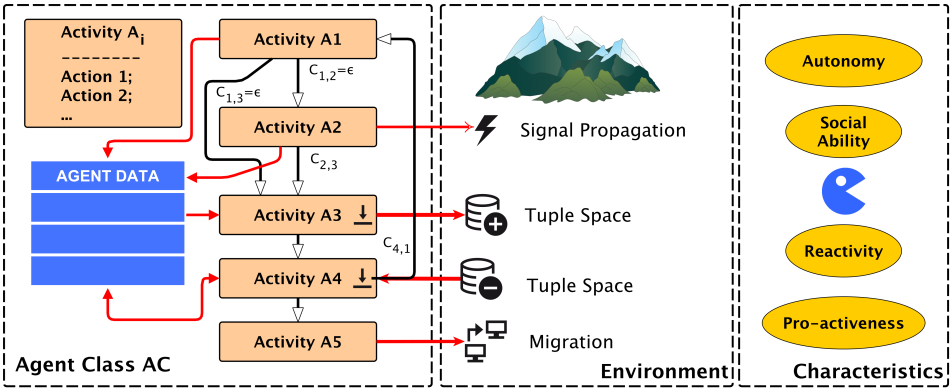


Abb. 62. (Links) Agentenverhalten, das von einem Aktivitätsübergangsgraphen vorgegeben ist, und die Interaktion mit der Umgebung, die durch Aktionen erfolgt, die in Aktivitäten ausgeführt werden (Rechts) Agentenmerkmale

- 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**.

- 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

11.3. DATG Modell

- Ein ATG beschreibt das vollständige Agentenverhalten.
- Jedes Unterdiagramm und jeder Teil des ATG kann einem Unterklasseverhalten eines Agenten zugeordnet werden.
- Daher führt das Modifizieren der Menge von Aktivitäten $\mathbb{A}$ und Übergängen $\mathbb{T}$ des ursprünglichen ATG zu mehreren Unter- und Oberverhaltensweisen, die Algorithmen implementieren, um verschiedene unterschiedliche Ziele zu erfüllen.
- Die Rekonfiguration der Aktivitäten führt zu einer Menge von Aktivitätsmengen $\mathbb{A}^* = \{\mathbb{A}_i \subset \mathbb{A}, \mathbb{A}_j \subset \mathbb{A}, \mathbb{A}_k \supset \mathbb{A}, ..\}$, die von der ursprünglichen Menge $\mathbb{A}$ abgeleitet sind, und die Modifikation oder Rekonfiguration von Übergängen $\mathbb{T}^* = \{\mathbb{T}_1 \subset \mathbb{T}, \mathbb{T}_2 \subset \mathbb{T}, ..\}$ ermöglicht die dynamische ATG-Zusammensetzung (Komposition) und Agentenunterklassifizierung zur Laufzeit,

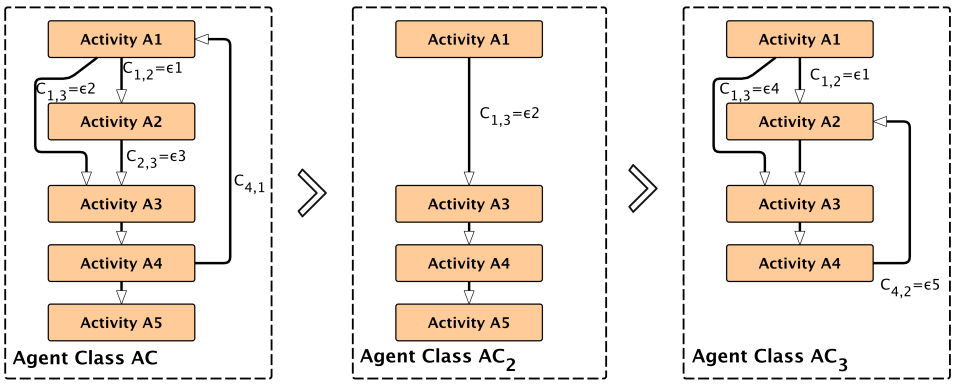


Abb. 63. Dynmaischer ATG: Transformation und Komposition

11.4. 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.

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 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.

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

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

Multiagentensysteme

Definition 16.
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 $\mathbf{AC} = \{AC_1, AC_2, ..\}$ bezeichnet werden. Ein Agent gehört zu einer dieser Klassen.

Jede Agentenklasse wird dann durch das erweiterte Tupel $AC = <\mathbb{A}, \mathbb{T}, \mathbb{F}, \mathbb{S}, \mathbb{H}, \mathbb{V}$, 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 ℙ die Menge der Parameter, die von der Agentenklasse verwendet werden.

Definition 17.
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.

11.5. 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.

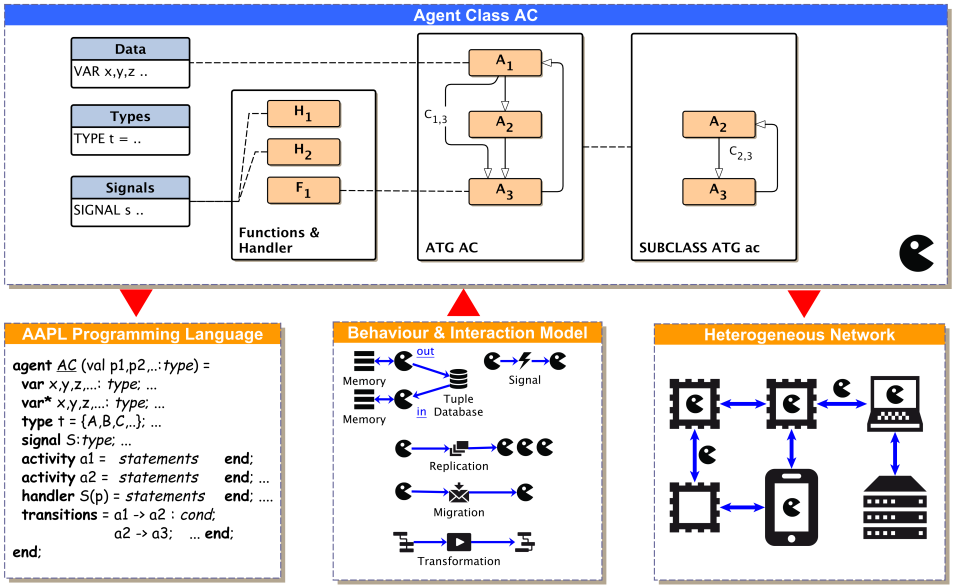


Abb. 64. Programmierebene des Agentenverhaltens mit Aktivitäten und Übergängen (AAPL, links); Agentenklassenmodell und Aktivitätsübergangsgraphen (oben); Agenteninstantiierung, -verarbeitung und -interaktion auf der Netzwerkebene (rechts)

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.

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

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 APP_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.

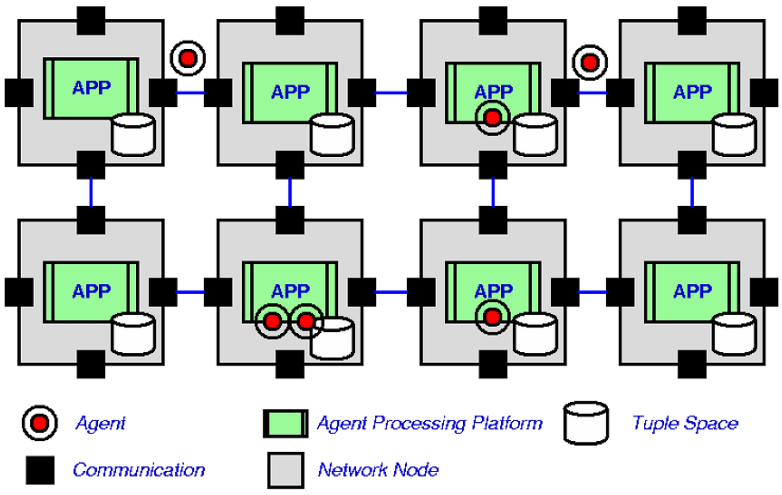


Abb. 65. Maschennetzwerk aus Netzwerkknoten mit APP und Agenten

➤ 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*, .. }

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.

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 Agentenidentifizierer (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.

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. 66. AAPL Signale

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,..);	$\nabla^{+-}(\text{id},v_1,?,?,..)$
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. 67. AAPL Tupelraum Operationen

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. 68. AAPL Verhaltensänderung und Rekonfiguration durch Veränderung der Aktivitäten- und Übgergangsmenge

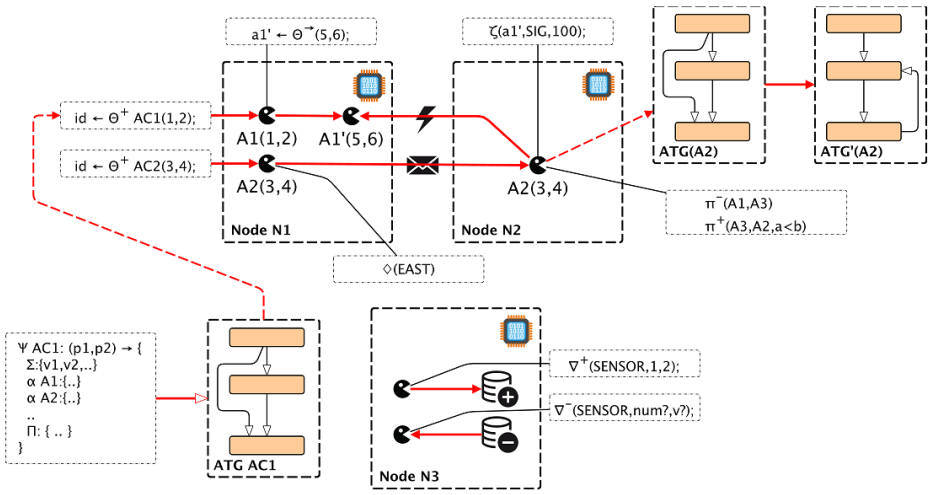
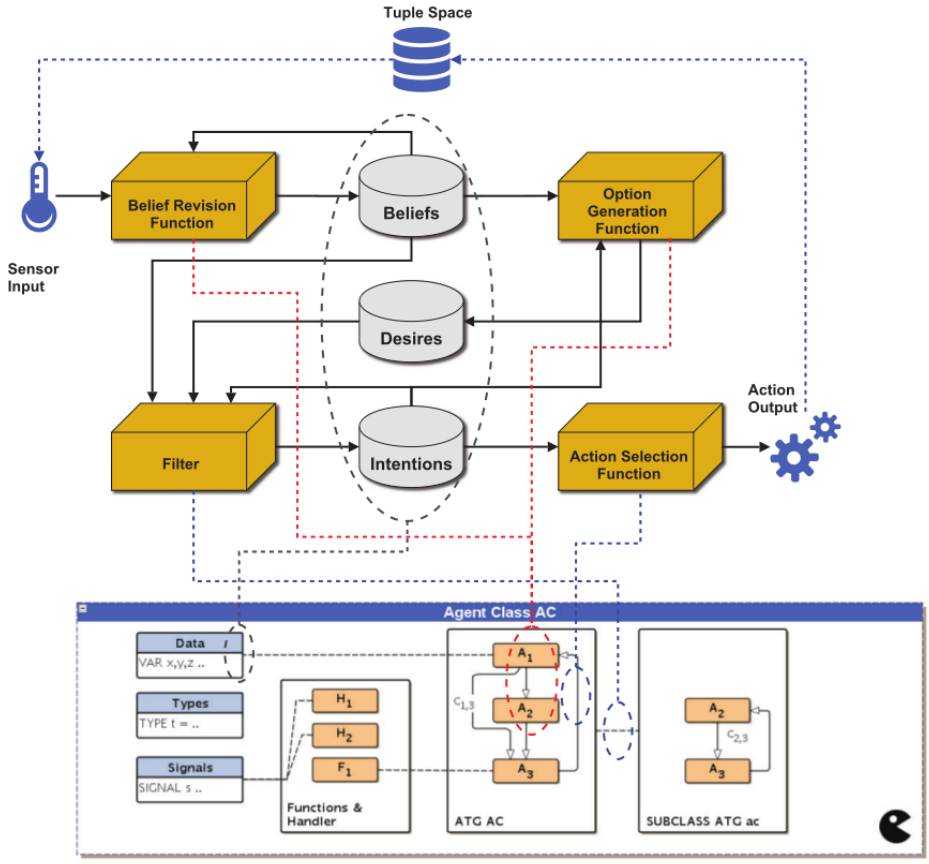
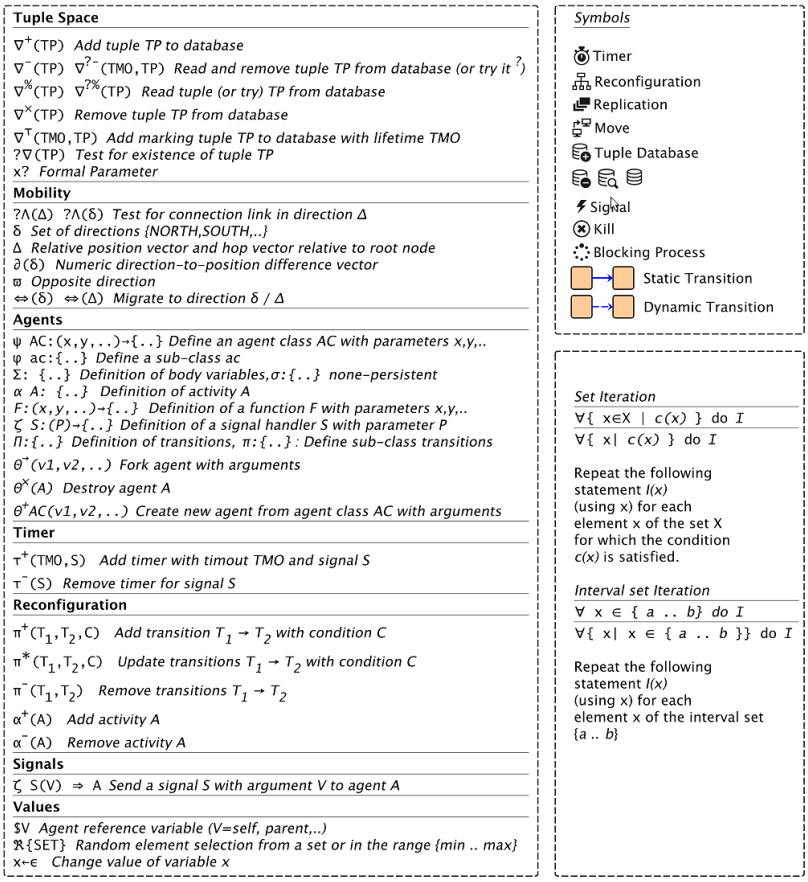


Abb. 69. Effekt von verschiedenen AAPL Anweisungen zur Laufzeit auf Agenten und Plattformen

Zusammenhang vom AAPL/DATG Verhaltensmodell mit BDI Architektur



11.6. AAPL Kurznotation



11.7. JavaScript :: Daten und Variablen

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

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

```
var υ = ϵ,..;   υ = ϵ;
```

11.8. 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,..)
```

- 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 (pi) { ϵ(pi) }
array.forEach(function(elem,index) { ϵ(pi) }
```

11.9. 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
```

➤ 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"]
```

11.10. 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.

```
function constructor (pi) {
  this.x=ϵ
  ..
}
constructor.prototype.methodi = function (..) {
  this.x=ϵ;
  ..
}
...

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

11.11. 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
- ➤ 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 18.

```
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;
}
```

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())`)

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!
- Es gibt drei verschiedene Blockkonstruktoren:
  - B: Ein sequenzieller Scheduling Block

- ❏ L: Ein iterativer Schleifenblock
- ❏ I: Ein Iteratorblock für Arrays und Objekte (Strukturen)

Definition 19.

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

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

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 ..
});
```

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
```

➤ 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.

- 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!

11.12. 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')
```

- 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);
```

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!

- 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
```

12. Plattformen

12.1. 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!

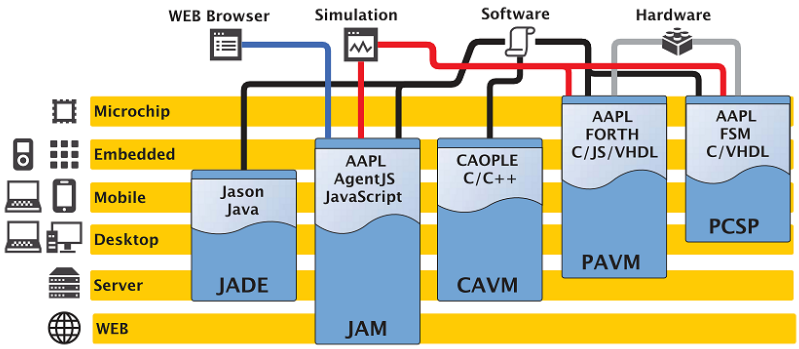


Abb. 70. Vergleich verschiedener Plattformen und ihre Einsatzgebiete

12.2. JADE

➤ JADE implementiert Agenten in Java

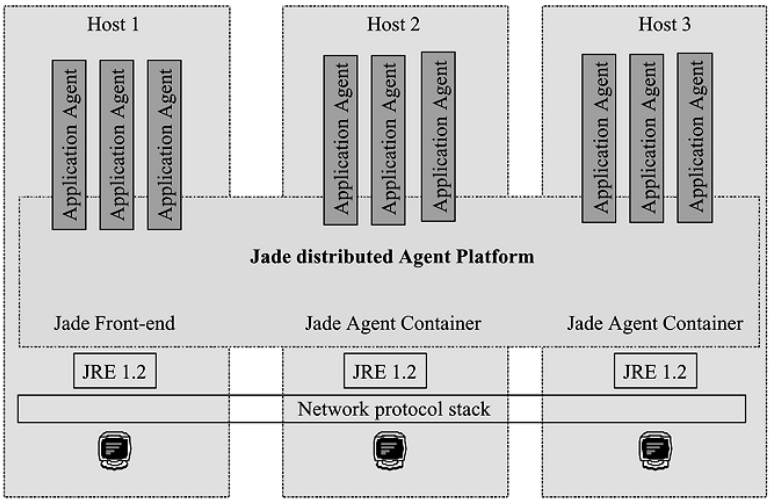


Abb. 71. Softwarearchitektur einer Agentenplattform [4]

➤ 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

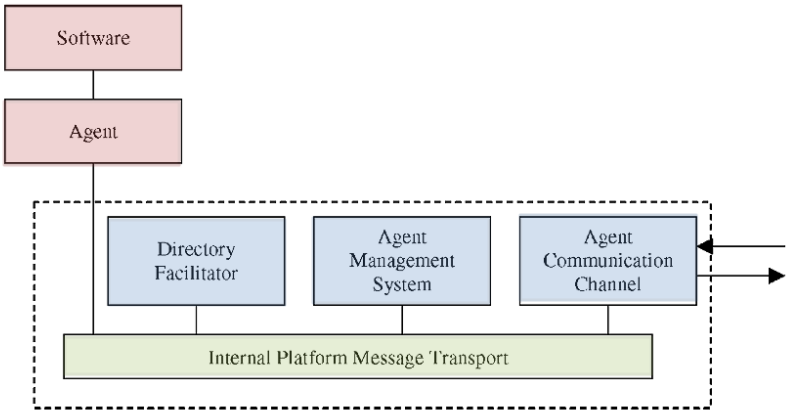


Abb. 72. Allgemeine FIPA-ACL Sofwareplattform [4]

- 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)

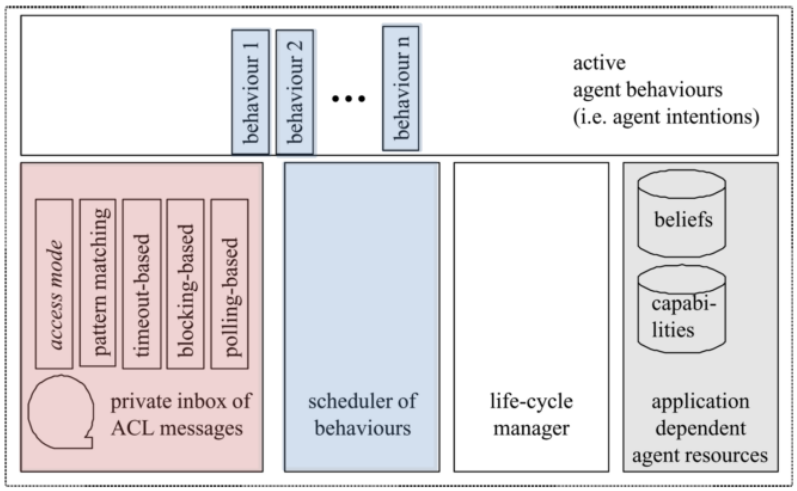


Abb. 73. JADE Agentenarchitektur [4]

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; }
    }
  } } }
```

12.3. 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
  - ❏ Corodova und WebView in Android Apps
  - ❏ usw.
- ➤ AgentJS Code kann direkt ausgeführt werden (nur Sandboxing erforderlich)!

➤ Eine physische JAM kann eine Vielzahl von virtuellen JAM Knoten ausführen (nur sequenzielles Scheduling)

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
- ❏ SECU: JAM Capabilities und Sicherheit
- ❏ WATCHDOG: Faires Agentenscheduling durch Laufzeitüberwachung (Time slicing)
- ❏ ML: Machine Learning → Fokus mobile Algorithmen und Modelle
- ❏ SAT: SAT Logic Solver → Knowledge Base

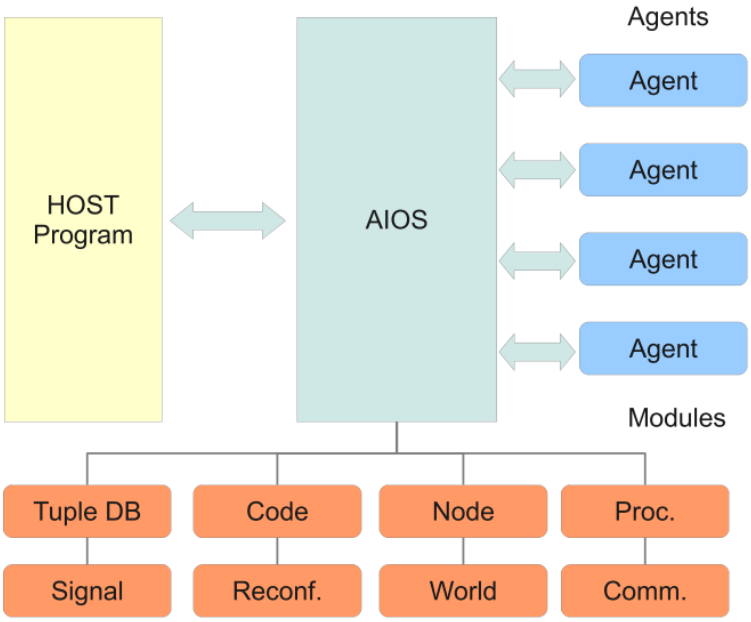


Abb. 74. Das Agent Input-Outut System als Schnittstelle zwischen Agenten und JAM und einer Hostplattform (bzw. Applikation)

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.
- 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.

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

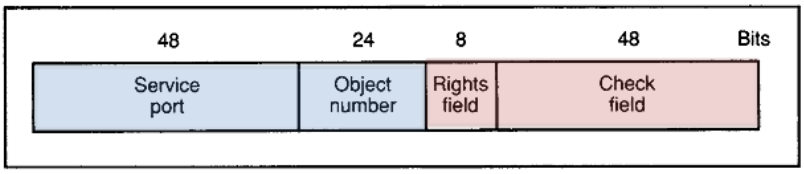


Abb. 75. Aufbau einer Capability: Der Serverport bindet die Capability an einen Service, die Objektnummer ist optional und kann eine Unterklasse des Service oder der zu schützenden Ressource darstellen, das Rechtefeld kodiert die möglichen Operationen der Serviceklasse, und das Schutzfeld (enthält Rechtefeld nochmals verschlüsselt) schützt die Capability gegen Manipulation.

Schutz von Capabilities

- ➤ Das Rechtefeld ist zentral da es die durch die Capability erlaubten Operationen angibt.
  - ❑ CPU Zeit erhöhen
  - ❑ Änderung des Privilegienlevels
  - ❑ Migration usw.
  - ❑ Zugriff auf Sensoren und persönliche Nutzerdaten
- ➤ 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 verschlüsselt.
  - ❑ Nur der Service / Agentenplattform kennt den privaten Schlüssel

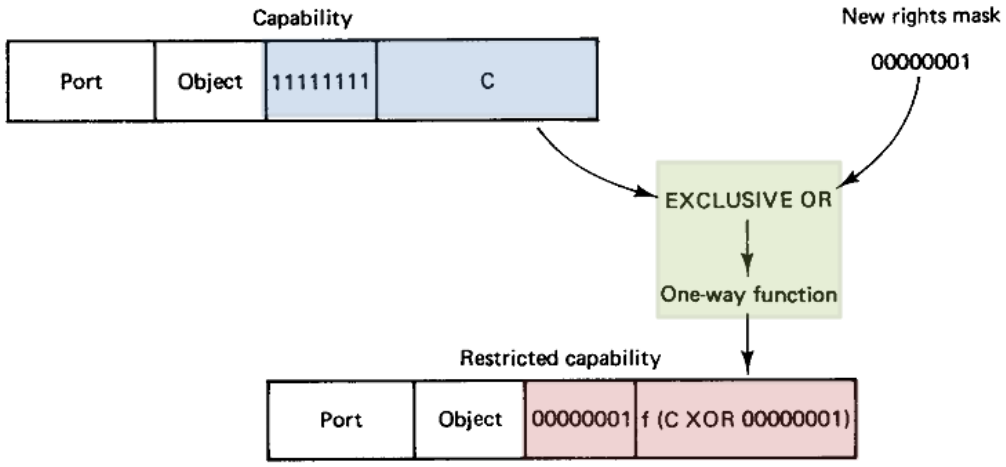


Abb. 76. Erzeugung einer öffentlichen restriktiven Capability aus einer privaten nicht eingeschränkten (enthält privaten Schlüssel C) mittels One-way Verschlüsselungsfunktion $f(C \text{ xor } R)$

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

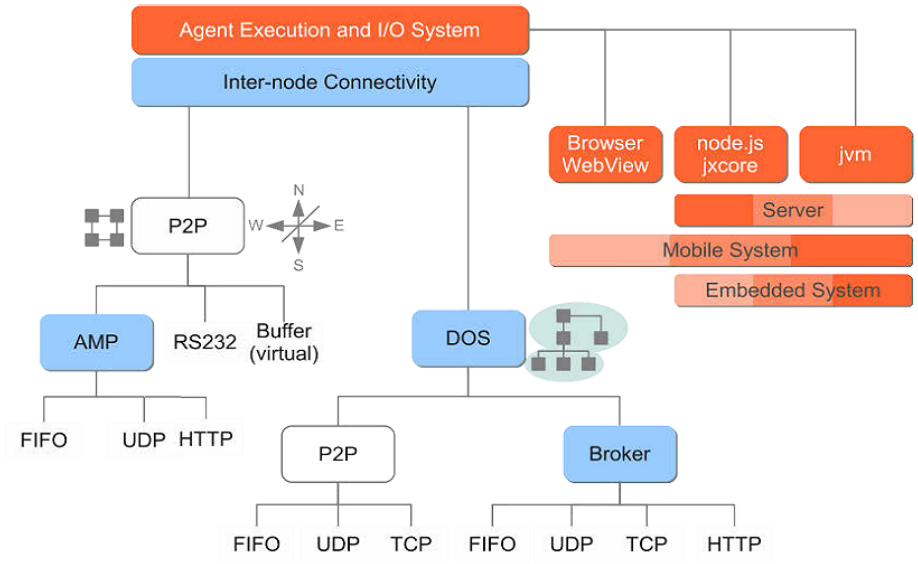
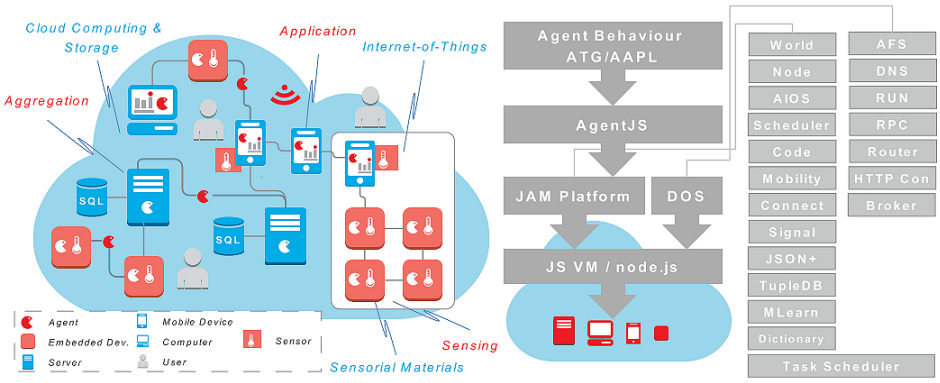


Abb. 77. JAM Konnektivität und eine breites Spektrum an unterstützten JavaScript Host Plattformen

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 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)



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
  - ❏ Transition/Übergangsfunktion
  - ❏ Signal- oder Exceptionhandler
  - ❏ Mikroaktivitätsblock (Scheduling Block)

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

1. Zeitscheibenverfahren
   - ➤ Faires Scheduling
   - ➤ Wird entweder durch eine modifizierte JS VM (mit 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 einen JS Programmkontext pro JS VM Instanz!
   - ➤ Daher müssen die Agentenprozesse (d.h. die Aktivitäts, Transitions-, und Handlerfunktionen) in einem maskierten Kontext ausgeführt werden → Sandkiste
   - ➤ Die Agenten werden zu Text serialisiert und ohne externe Referenzen außer der AIOS API gemäß wieder deserialisiert (d.h. in Code gewandelt)

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, Graph, und Gewichte sind portabel)
  - ❏ Support Vector Machines, kNN
  - ❏ Reinforcement Learning

13. Verteiltes Verhalten und Gruppen

13.1. Gruppenentscheidung und Verhandlung

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

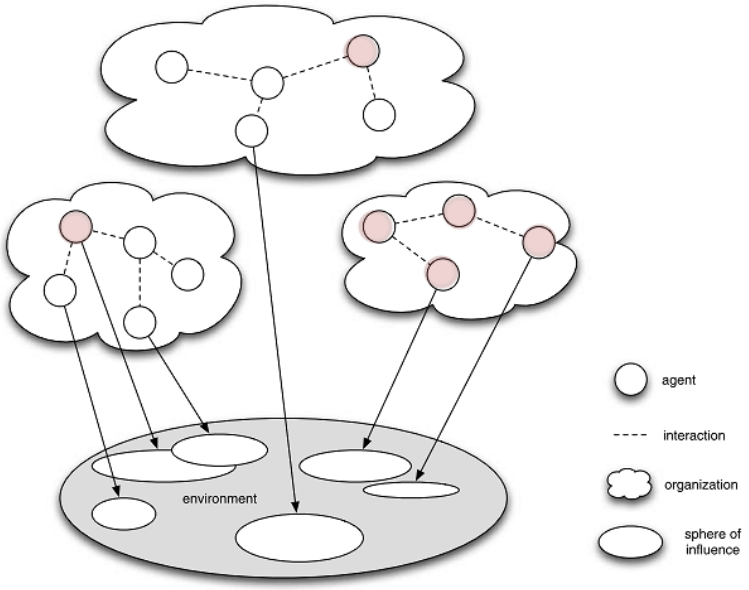


Abb. 78. Typische Gruppenstrukturen in Multiagentensystemen

- Dabei können Gruppenentscheidungen und Verhandlungen auf Basis von **Nützlichkeitsfunktionen** geführt werden
- Es sei eine Gruppe von Agenten (Prozessen) $\boldsymbol{G}=\{ag_1,ag_2,..,ag_{\mathsf{m}}\}$
- Es gibt nun verschiedene Stufen der Verhandlung:
  - Wahl eines Gruppenführers (Leader) oder Mediators
  - Abgabe von Stimmen / Absichten
  - Bestimmung eines gemeinsamen 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) …

13.2. 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.

Automatisierte Aushandlung kann in Multiagentensystemen 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.  i <- arg max  ui( )
1. Propse  i
2. Receive  j proposal
3. if ui( j) > ui( i)
4.   then Accept  j
5.   else  i <-  i' such that uj( i') >= e+uj( i)
6. loop 2.
```

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

13.3. 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.

➤ 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]

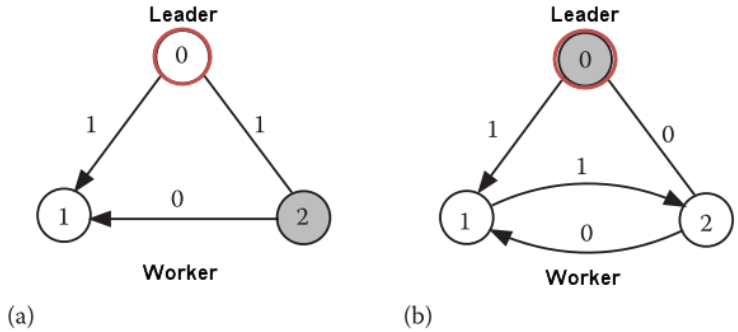


Abb. 79. Byzantinisches Generalproblem: (a) Leader 0 ist fehlerfrei, Worker 2 ist fehlerhaft (b) Leader 0 ist fehlerhaft, Worker 1 und 2 sind fehlerfrei [E]

➤ 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

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 $\{0, 1\}$ an jeden Worker j i.

ii. Jeder Worker j akzeptiert den Wert von i als Befehl vom Leader i.

Definition 20.

Algorithmus OM(m)

1. Leader i sendet einen Wert v $\{0, 1\}$ an jeden Worker j 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.

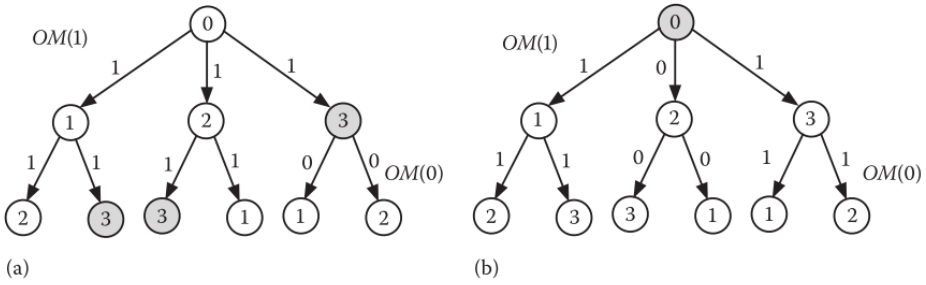


Abb. 80. Eine Illustration von OM(1) mit vier Prozessen und einem fehlerhaften Prozess: die Nachrichten auf der oberen Ebene spiegeln die Eröffnungsnachrichten von OM(1) wider und die auf der unteren Ebene spiegeln die OM(0)-Meldungen wider, die von den Mitteilungen der oberen Ebene ausgelöst werden. (a) Prozess 3 ist fehlerhaft. (b) Prozess 0 (Leader) ist fehlerhaft. [E]

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

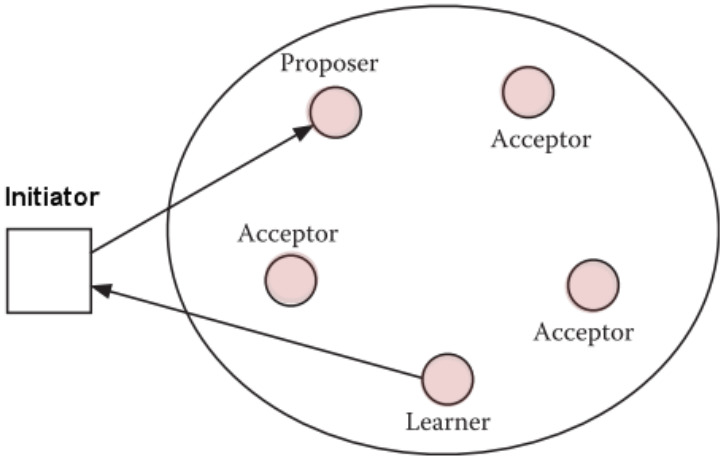


Abb. 81. Typische Rollenverteilung beim Paxos Algorithmus

- 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.
- 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, ,) 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').

2. **Aufforderung zur Annahme eines Eingabewertes**

   - ➤ Wenn ein Antragsteller *ack*(n, ,) 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.

13.4. 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

13.5. 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.

➤ 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 partielles geometrisches Modell des Netzwerkes benötigt

Beispiel eines Event Agenten 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') }
```

```
    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 }}
```

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

13.6. 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.
- 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.

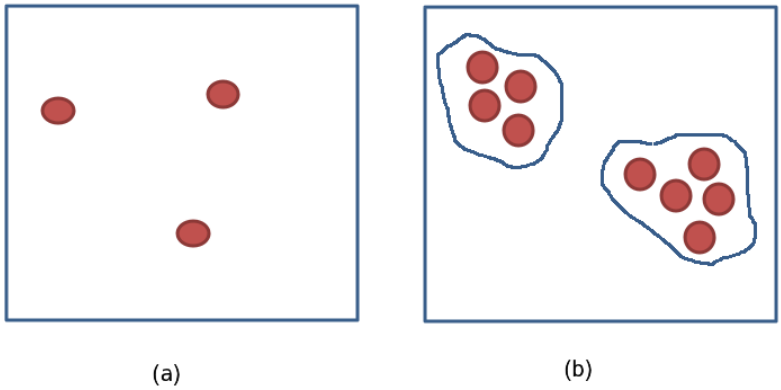


Abb. 82. (a) Nichtkorellierte Sensorstimuli (b) Korrelierte Sensorstimulibereiche

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.

➤ 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* {-*R*, .., +*R*} ähnlich dem Wert S an der aktuellen Position ist (Differenz ist kleiner als der Parameter d), wird H um eins erhöht.

➤ 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.

➤ 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.

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    }
```

- Aktivitäten *reproduce* und *diffuse*

```
28    α reproduce: {
29      live--;
30      ∇ˣ(H,$self,?);
31      if ?∇(FEATURE,?) then ∇⁻(FEATURE,n?) else n ← 0;
32      ∇⁺(FEATURE,n+1);
33      if live > 0 then
34       π*(reproduce → init)
35       ∀{nextdir∈δ | nextdir ≠ backdir ∧ ?Λ(nextdir)} do
36         Θ⃗(nextdir,radius)
37       π*(reproduce → exit)
38    }
39    α diffuse: {
40      live--;
41      ∇ˣ(H,$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
```

➤ 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    Π: {
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; ∇ᵗ(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     }
```

13.7. 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

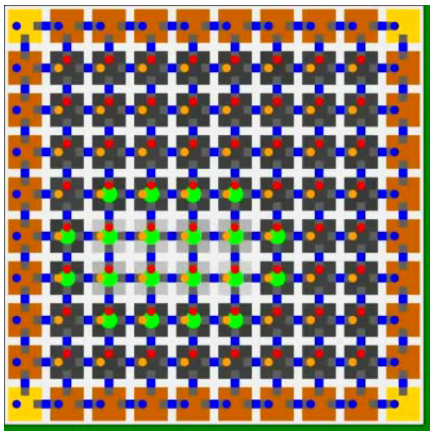
➤ Sensorwerte können dann per Randomwalk oder gerichteter Diffusion verteilt werden.

Gerichtete Diffusion

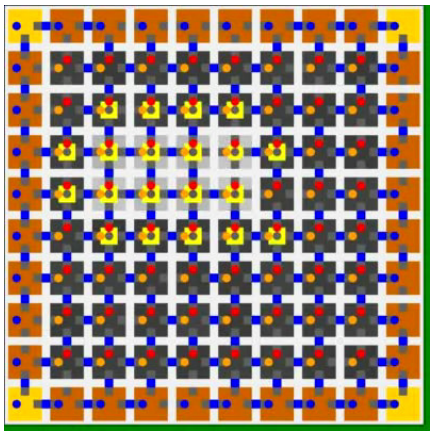
Von einem Quellknoten werden Duplikate von Verteilungsagenten in alle Richtungen ausgesendet, und an den Rändern des Netzwerks zu den Senkeknoten (Rechnern) geleitet.

Beispiel in Aktion: Positive Ereigniserkennung

Cluster mit 100% intakten Netzwerkverbindungen

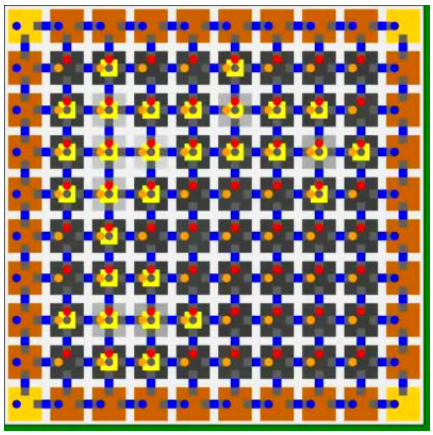


Cluster mit 60% intakten Netzwerkverbindungen



Beispiel in Aktion: Gemischte Situation

Cluster und Störungen



14. Agenten und Lernen

14.1. Maschinelles Lernen

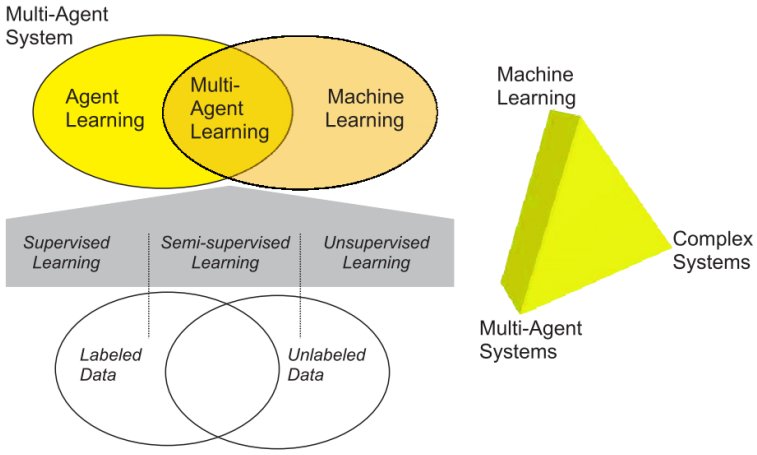


Abb. 83. Maschinelles Lernen lässt sich mit Agenten kombinieren

14.2. 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

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

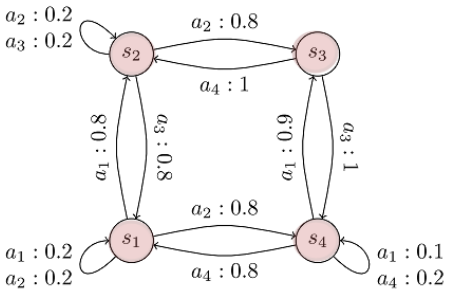
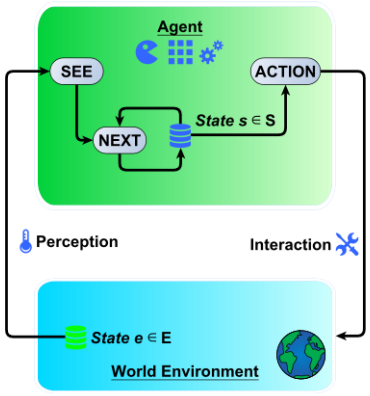
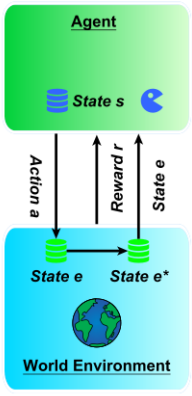


Abb. 84. Grafische Darstellung eines Markov-Entscheidungsprozesses zusammen mit Werten für die Übergangs- und Belohnungsfunktionen. Der Startzustand sei s1, und T die Übergangsfunktion T(s_i, a, s_j) für den Übergang $s_i \rightarrow s_j$.

- 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. π:**S** $\rightarrow$ **A**
- *Ein verstärkender Lernagent muss die Strategie* ☆ *∈ π finden, die seine zukünftigen Belohnungen maximiert → optimales Erreichen von Zielen*!

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





14.3. 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.
- 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.

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

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

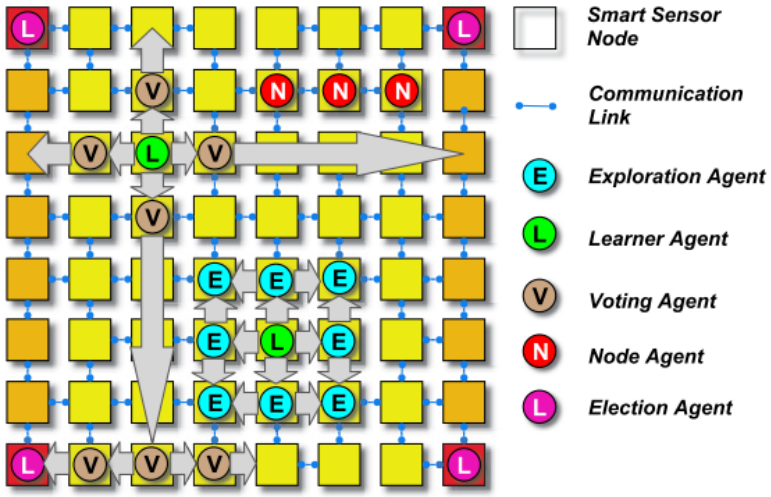


Abb. 85. Logische Netzwerkansicht und Population mit mehreren Agenten, die aus verschiedenen Verhaltensklassen instantiiert wurden.

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).

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.

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.

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.

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.

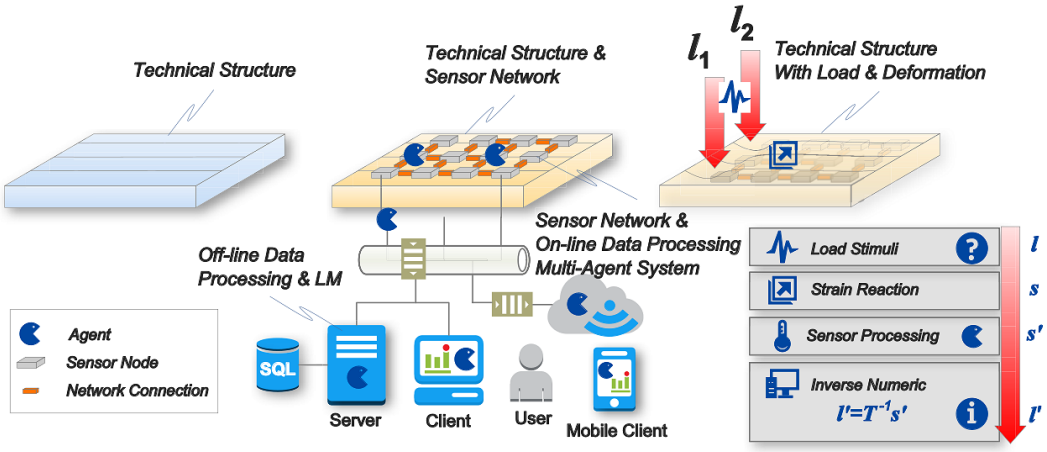


Abb. 86. Vom Material zur intelligenten überwachten Struktur mit mobilen Agenten

Use Case: Erdbebenüberwachung und Crowd Sensing

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

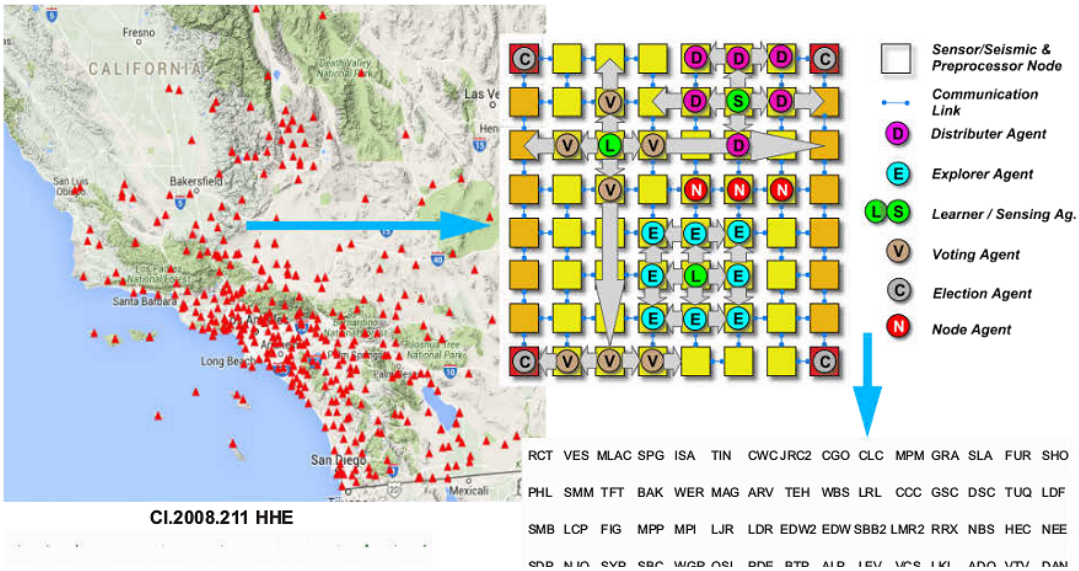


Abb. 87. (Oben, links): Das südkalifornische seismische Sensornetzwerk [Google Maps] (Oben, rechtes) Sensornetzwerk mit Stationen, die auf einer logischen zweidimensionalen Mesh-Grid-Topologie mit räumlicher Nachbarschaftsplazierung abgebildet wurden und Beispielpopulation mit verschiedenen mobilen und immobilen Agenten [1]

14.4. Verteiltes Inkrementelles Lernen

- ➤ Neben dem grob granulierten Zyklus **Lernen** → **Modell** → **Klassifikation** mit einer Datenmenge (Trainingsdaten) $\mathbf{D}=\{D_1,D_2,..,D_n\}$ 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. Entscheidungsbaum)
  - ❏ Entscheidungsbäume bestehen aus Knoten die Eingabevariablen nutzen um die Klassifikation optimal durch Pfaditeration zu erreichen
  - ❏ Welche Variablen optimal sind (Merkmalsselektion) wird von den Trainingsdaten bestimmt
  - ❏ Sind diese nur unvollständig bekannt, kann eine nachträgliche Erweiterung des Baumes zur Verwendung ungeeigneter (schwacher) Variablen führen → Schlechte Klassifikationsergebnisse sind die Folge!

Rückkopplung

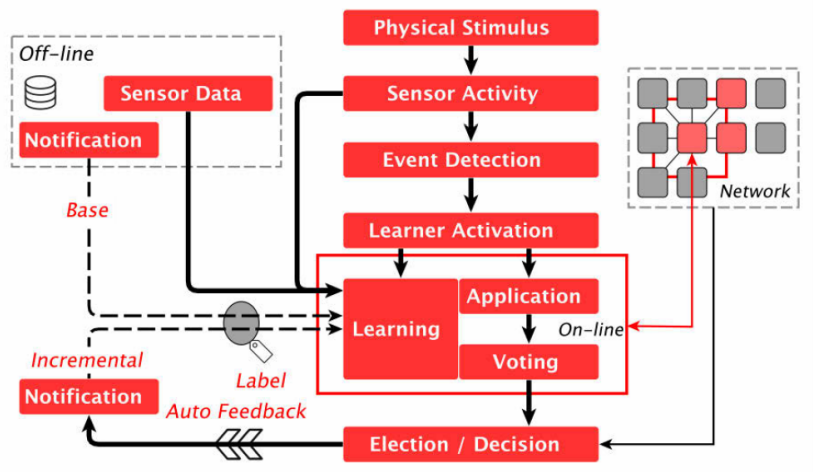


Abb. 88. Das Grundkonzept: Globales Wissen basierend auf Mehrheitsentscheidungen wird an lokale Lerninstanzen zurückgegeben, um das erlernte Modell zu aktualisieren.

15. References

15.1. Books

A. J. 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 Systems - 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.

15.2. 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

15.3. Präsentationen

a. Keith L. Clark, *Intelligent Agents*, Department of Computing, Imperial College

15.4. Videos and WEB