

### Eine soziale Simulation von Theoriebewertungen innerhalb von Wissenschaftsgemeinschaften

Kurzawe, Daniel

Erstveröffentlichung / Primary Publication

Forschungsbericht / research report

**Empfohlene Zitierung / Suggested Citation:**

Kurzawe, D. (2010). *Eine soziale Simulation von Theoriebewertungen innerhalb von Wissenschaftsgemeinschaften*. München. <https://nbn-resolving.org/urn:nbn:de:0168-ssoar-51088-4>

**Nutzungsbedingungen:**

Dieser Text wird unter einer CC BY-NC-ND Lizenz (Namensnennung-Nicht-kommerziell-Keine Bearbeitung) zur Verfügung gestellt. Nähere Auskünfte zu den CC-Lizenzen finden Sie hier:

<https://creativecommons.org/licenses/by-nc-nd/4.0/deed.de>

**Terms of use:**

This document is made available under a CC BY-NC-ND Licence (Attribution-Non Commercial-NoDerivatives). For more information see:

<https://creativecommons.org/licenses/by-nc-nd/4.0>

# Eine soziale Simulation von Theoriebewertungen innerhalb von Wissenschaftsgemeinschaften

Daniel Kurzawe

Die Beobachtung der Dynamik von Theorien und deren Auswirkung auf Wissenschaftsgemeinschaften bietet viele Perspektiven und ist somit interdisziplinäre Betrachtung des Wissenschaftsprozesses. Während die Wissenschaftstheorie und Philosophie Ideen zur Wissenschaftssprache und des Inhalts liefern, beschreibt die Soziologie die gruppenspezifischen Prozesse der Akteure. In dieser Arbeit wird die Dynamik beider Aspekte in Abhängigkeit gebracht und exemplarisch dargestellt. Dazu wird eine Computersimulation entwickelt, welche es erlaubt modellhafte Experimente durchzuführen.

Dieser Text ist eine überarbeitete Version meiner Magisterarbeit unter dem Titel „Eine computergestützte Simulation von Theoriebewertungen“, welche 2010 an der Ludwig-Maximilians-Universität München bei Prof. Wolfgang Balzer eingereicht wurde und die Vorarbeit zu meiner Promotion zu gleichem Thema geboten hat.

Ich danke hierbei insbesondere Prof. Wolfgang Balzer und Dr. Ruth Reiche für die Diskussionen, Ideen und die große Unterstützung.

*Daniel Kurzaue*

*München - Berlin - Göttingen, Februar 2010 - 2017*

**Schlagnorte:** Wissenschaftstheorie, Philosophie, Soziologie, Computersimulationen, Theoriebewertung, Strukturalismus, Multiagentensimulation, Wissenschaftsgemeinschaften, Theorieevolution

# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>6</b>
1.1	Überblick . . . . .	7
1.2	Sozialebene . . . . .	11
1.2.1	Die soziale Simulation . . . . .	11
1.2.2	Das Überzeugungssystem . . . . .	12
1.3	Strukturebene . . . . .	13
1.3.1	Strukturen . . . . .	14
1.3.2	Datenstruktur . . . . .	15
1.3.3	Passung . . . . .	16
1.4	Die Zeit . . . . .	18
1.5	Prolog . . . . .	19
1.5.1	Eine Theorie in Prolog . . . . .	21
<b>2</b>	<b>Die Simulation</b>	<b>25</b>
2.1	Die Agenten . . . . .	25
2.2	Die Hypothesen . . . . .	26
2.3	Aufbau . . . . .	28
2.4	Ablauf . . . . .	29
2.5	Die Schleifen . . . . .	29
2.5.1	Die Aktionen . . . . .	31
2.5.2	Ablauf der Handlungen . . . . .	33
2.5.3	Das Passungsprogramm . . . . .	37
2.5.4	Die Urban Regel . . . . .	38
<b>3</b>	<b>Simulationsläufe und Beispiele</b>	<b>40</b>
<b>4</b>	<b>Das Prologprogramm</b>	<b>42</b>
4.1	simulation.pl . . . . .	42

4.2	actions.pl . . . . .	48
4.3	agents.pl . . . . .	53
4.4	config.pl . . . . .	54
4.5	world1.pl . . . . .	56
4.6	function.pl . . . . .	57
4.7	fit.pl . . . . .	60
4.8	urban_cart.pl . . . . .	60
4.9	Das Beispiel KSM . . . . .	61
<b>5</b>	<b>Fazit</b>	<b>63</b>



# 1 Einleitung

Mit dieser Arbeit wird eine computergestützte Simulation zur Theoriebewertung innerhalb einer Multiagentensimulation vorgestellt. Durch diese Simulation wird gezeigt, wie Faktoren der Sozial- und Strukturebenen im Bezug zur Theoriebewertung ineinandergreifen.

Für die Umsetzung wird die Programmiersprache Prolog verwendet, welche sich als deklarative Programmiersprache besonders anbietet und welche durch eine verhältnismäßig einfache Lesbarkeit besticht. Als Prologumgebung wird *SWI-Prolog 5.8.2* genutzt.<sup>1</sup> Auf die Syntax von Prolog werde ich in dieser Arbeit nur flüchtig eingehen. Für weitere Lektüre im Bezug zu Prolog, verweise ich auf Bratko, *Prolog - Programmierung für künstliche Intelligenzen* und die umfangreiche Dokumentation von SWI-Prolog Wielemaker und Anjewierden, *Programming in XPCE/Prolog*.

Die Arbeit ist in fünf Abschnitte unterteilt: Zunächst werde ich einen kurzen Überblick über die Hintergründe des Themas und die verwendeten Begrifflichkeiten geben. Im zweiten Teil wird die Konzeption der Simulation erläutert. Dort werden exemplarisch Teile der Umsetzung in Prolog diskutiert und zentrale Abschnitte im Detail erklärt. Im Anschluss folgt das Prolog Programm mit den vorgestellten Konfigurationen. Diese wird auch in den wichtigsten Punkten erläutert und mit Beispielen dargestellt. Im fünften Teil werde ich dann ein konkretes Beispiel erläutern. Abschließend werde ich im auf offene Punkte dieser Arbeit eingehen und einen Ausblick auf weitere Ansatzpunkte geben.

---

<sup>1</sup>SWI-Prolog ist unter <http://www.swi-prolog.org> zu finden (Stand: April 2010) und steht unter der *LGPL* Lizenz frei zur Nutzung.

## 1.1 Überblick

Betrachtet man die Handlungen von Forschern, lässt sich die Wissenschaft als System dieser Handlungen verstehen.<sup>2</sup> Dieses System lässt sich in zwei Ebenen aufteilen: Zum einen in die Sozialebene, auf der Wissenschaftler agieren, forschen und sich miteinander im Wettbewerb messen und zum anderen auf die Strukturebene in der u.a. die Theorien formuliert sind und Daten auf ihre Passung überprüft werden.<sup>3</sup> Zentrale Fragen für die Bewertung der Theorien sind: *Wann* passen Theorie und Daten *in welchem Verhältnis* zusammen? Und daraus resultierend die qualitative Frage nach der Güte der Theorie: *Wie gut* liefert eine Theorie eine Beschreibung unserer Wahrnehmungswelt? Betrachtet man die Prozesse im Hinblick auf die Sozialebene, folgt daraus die Frage nach den Auswirkungen der Theoriebewertungen auf die Entwicklung des Wissenschaftsprozesses. *Wie beeinflussen Theoriebewertungen den Wissenschaftsprozess?*

Kuhn hat den Wissenschaftsprozess in einen historischen und sozialen Kontext gestellt. Nach Kuhn ist die Theorieentwicklung in einem evolutionären Prozess zu erklären.<sup>4</sup> Der Wissenschaftsprozess ist hierbei in mehrere Phasen unterteilt: Die normale Wissenschaft, in welcher ein Paradigma jeweils die „geltende“ und allgemein anerkannte Theorie darstellt und die revolutionäre Phase, in der ein Paradigma fällt und eine neue Theorie diesen Status erlangt. Theorien gelten solange als *Paradigma*<sup>5</sup>, bis hinreichend starke Anomalien zu einer Krise führen. Nach einer Phase der Neuentwicklung tritt ein neues Paradigma an die Stelle des vorhergehenden. Diese Anomalien lassen sich im Prinzip durch ein schlechtes Passungsverhältnis, also eine hohe Entropie zwischen Daten und Erwartungswerten verstehen.

Hierbei hat Kuhn den Kern des Theoriebegriffs weitestgehend unangetastet gelassen und sich auf soziale und historischen Untersuchungen des Wissenschaftsprozesses konzentriert. In Stegmüller, *Probleme und Resultate der Wissenschaftstheorie und Analytischen Philosophie Band II - Theorie und Erfahrung* und Balzer, Moulines und Sneed, *An Architectonic for Science : the structuralist program* wurde der Theoriebegriff, aufbauend

---

<sup>2</sup>Vgl. Balzer, *Die Wissenschaft und ihre Methoden*, S.26ff

<sup>3</sup>Vgl. ebd., S.11f

<sup>4</sup>Siehe etwa Kuhn, *Die Struktur wissenschaftlicher Revolutionen, zweite revidierte Auflage* und Stegmüller, *Hauptströmungen der Gegenwartphilosophie - Band II 6., erweiterte Auflage*, S.726ff

<sup>5</sup>Dieser Begriff wurde wegen seiner Mehrdeutigkeit stark kritisiert. Eine genauere Übersicht über die unterschiedlichen Lesarten ist in Masterman, „Die Natur eines Paradigmas“ zu finden. Wie später noch genauer beschrieben, wird im Folgenden die engere Definition aus Balzer, Moulines und Sneed, *An Architectonic for Science : the structuralist program* verwendet.



## 1 Einleitung

auf Arbeiten von Suppes und Sneed, aufgearbeitet und ein Brückenschlag zwischen den Ansatz Kuhns und den Formalen Untersuchungen zum Theorieaufbau, wie er bei Suppes zu finden ist, geschlagen. Doch gab es auch in der von Kuhn vorgestellten Darstellung Kritik: Lakatos hat gezeigt, dass das Schema, wie Kuhn es skizzierte, an einigen Stellen zu lückenhaft. Kuhn kritisierte mit seiner Arbeit unter anderem Poppers Modell der Falsifikation als Grundlage der Theorieentwicklung. So führt die Falsifikation nicht direkt zur Verwerfung einer Theorie, sondern viel mehr zum Aufstellen von Hilfhypothesen. Um dies zu untermauern hat Kuhn einige Beispiele der Wissenschaftsgeschichte aufgearbeitet. Lakatos hat mit seiner Arbeit versucht Poppers und Kuhns Arbeit in einer übergreifenden Interpretation zu verbinden. Im Kern wird beschrieben, dass die Bestätigung und Widerlegung von Theorien ist in vielen Fällen Zusatzannahmen notwendig werden.<sup>6</sup> Es werden zusätzliche Hypothesen aufgestellt um ein angegriffenes Paradigma zu retten. Das Passungsverhältnis wird durch zusätzliche Annahmen verbessert und dadurch das ursprüngliche Paradigma modifiziert und eine Krise zumindest aufgeschoben.

Hier wird die Verflechtung zwischen der Theorie- und Sozialebene deutlich: Theorien werden auch aus sozialen Gefügen weiter belebt und gepflegt. Es werden Hypothesen aufgestellt und diese an ihren Anwendungsgebieten gemessen. Verschiedene Wissenschaftler und Forschungsgruppen bieten Modelle, Daten und Messmethoden. Soziale Effekte, welche auf die Theorie zurückwirken, lassen sich etwa durch die Finanzierung von Projekten und Anträgen darstellen. Es entstehen Spannungsverhältnisse, wie etwa bei der Konkurrenz bei der Vergabe von Forschungsgeldern. Ökonomische Faktoren haben starken Einfluss auf die Forschungsbedingungen einzelner Forschungsgruppen. Unterfinanzierte Gruppen können aufgrund schlechterer Bedingungen weniger effektiv arbeiten, als hochgradig geförderte Projekte. Das Fehlen von Finanziellen Mitteln bietet weniger Möglichkeiten für aufwendigere Experimente.

Die wirtschaftliche Anwendbarkeit des jeweiligen Arbeitsbereiches ist hier mitbestimmend. Der Konkurrenzdruck kann ebenfalls durch staatliche Förderungen, wie etwa durch *Spitzenforschungsprogramme* gestärkt werden.

Erfolgreiche Forschungsgruppen, also Gruppen, welche ein besseres Passungsverhältnis aufweisen und in Folge dessen attraktivere Theorien anbieten oder aus andern Gründen<sup>7</sup>

---

<sup>6</sup>Vgl. Lakatos und Musgrave, *Kritik und Erkenntnisfortschritt - Wissenschaftstheorie Wissenschaft und Philosophie Band 9* und Lauth und Sareiter, *Wissenschaftliche Erkenntnis*, S. 129f

<sup>7</sup>Diese können z.B. auch eine *einfachere* und somit in der Lehre erfolgreichere Theorie oder eine, durch soziale Einrichtungen, stark geförderte Theorie sein. So wurden etwa aus religiösen Bestrebungen vielfach Einfluss auf die Entwicklung der Wissenschaft und Lehre ausgeübt: Vgl. Mason, *Die*

## 1 Einleitung

erfolgreicher sind, werden mehr finanzielle Erfolge im Sinne von Förderungen wie etwa durch geförderte Forschungsprojekte Projekte, Forschungsstipendien oder auch kommerzielles Interesse oder ähnlich verbuchen.

Die Verbreitung von Wissen und Theorien in Forschung und Lehre zeigt eine weitere Verbindung: Ist die Hürde für Veröffentlichungen zu hoch gesetzt, etwa durch hohe Kosten oder strenge Gutachter, wird vermutlich die Zahl der wissenschaftlichen Veröffentlichungen sinken. Im weiteren Verlauf kann eine geringere Anzahl von Neuerscheinungen doch auch dazu führen, dass die geringere Anzahl an Veröffentlichungen deutlich stärker rezipiert werden. Sinkt die Hürde, können einzelne Arbeiten in der Masse von Veröffentlichungen untergehen. Dabei sind die Gutachtersysteme ebenso eine Verbindung zwischen Sozialen- und Strukturebene. Den gerade hier werden Bewertungen durchgeführt, welche sich zum einen auf die Theoriegüte stützen aber zum anderen auch Auswirkungen auf die Theorieentwicklungen haben und so im Idealfall auch in Verbindung zur Theorie stehen.

Eine Computersimulation bietet die Möglichkeit Faktoren dieser Art zu modellieren. Auch lässt sich automatisiert nach signifikanten Zusammenhängen zwischen einzelnen Faktoren suchen. Diese Vorteile der Simulation als Methodik zur Datengewinnung oder zum Test von Hypothesen haben dazu geführt, dass sich gerade in den Naturwissenschaften Computersimulationen als Methodik etabliert haben<sup>8</sup>.

In der Wissenschaftstheorie finden ebenfalls Computersimulationen ihre Anwendung. So beschreibt etwa Langley, *Scientific Discovery: Computational Explorations of the Creative Processes* mit seinem Programm *Bacon* eine Methodik, wie mittels heuristischen Algorithmen, Hypothesen automatisiert aus Messdaten gebildet werden können. Einen ähnlichen Weg wird mit der Implementierung *Eurequa* von Schmidt und Lipson, "Distilling Free-Form Natural Laws from Experimental Data" verfolgt. Hier analysiert das Programm gegebene Messdaten und versucht diese mittels einer Funktion zu beschreiben. Nach erfolgreicher Analyse gibt das Programm Empfehlungen für weitere Messreihen ab. In Urban, "Zusammenspiel von Problemösungsstrategien und Theorieentwicklung: Ein Ansatz zur Simulation" wird ein Ansatz für eine Computersimulation zur Theorieentwicklung besprochen.

Aus den empirischen Wissenschaften gibt es ebenfalls Ansätze: Ein Beispiel ist King

---

*Geschichte der Naturwissenschaften, zweite Auflage, S.210-229.*

<sup>8</sup>Hofmann, *Dynamik sozialer Praktiken und ihrer zu Grunde liegenden Einstellungen - Modellierung und Simulation*

## 1 Einleitung

u. a., "The Automation of Science" mit dem Projekt *Robot Scientist*. Der *Robot Scientist* führt empirische biologische Experimente durch um Messdaten zu generieren. Auf Grundlage dieser Ergebnisse wird dann eine Hypothese aufgestellt und anhand dieser werden weiterer Experimente durchgeführt. Dabei hat das Programm einen eigenen Handlungsspielraum selbständig neue Experimente zu planen und mit Hilfe des Roboters durchzuführen. Dies ist eine Entwicklung, welche vermutlich die Art und Methodik des Forschens nachhaltig verändern wird. Das hat nicht zuletzt auch Einfluss auf die Rolle des Forschers.

In der hier vorgestellten Simulation werde ich nicht auf die Generierung von Daten eingehen. Die Arbeit konzentriert sich auf das Passungsverhältnis zwischen gegebene Daten und Hypothesen und die daraus gegebenen Wechselwirkungen zwischen der Theorie und Sozialebene.

Hierfür ist ein Grundgerüst, welches die Bewertung von Theorien ermöglicht, notwendig. Dadurch, dass mit dem strukturalistischen Ansatz eine Möglichkeit gegeben ist Theorien formal in ihrer Struktur darzustellen und nicht als einfache Satzmengen aufzufassen, können Theorien übersetzt und in einer künstlich geschaffenen Welt simuliert und - wie im Folgenden zu zeigen ist - anhand ihres Passungsverhältnisses bewertet werden.

Der Aufbau lässt sich, wie bereits erwähnt, in zwei Ebenen unterteilen:

1. **Sozialebene** Hier werden die Akteure<sup>9</sup> beschrieben welche Handlungen ausführen. Zu jeder Zeiteinheit<sup>10</sup> der Simulation haben die Akteure Handlungsmöglichkeiten, welche die Akteure und auch die Simulation in neue Zustände überführen können. Die Auswirkungen der Handlungen kommen auch auf der Strukturebene zu tragen.
2. **Strukturebene** In dieser werden die Theorien betrachtet und die Passung zwischen den Modellen und Daten berechnet. Je nach Passungsgrad werden auf der Sozialebene andere Prozesse ausgelöst. Abhängig von Prozessen der Sozialebene, wird die Passung zwischen Daten und Modellen berechnet.

---

<sup>9</sup>Akteure werden in der Simulation auch als *Agenten* bezeichnet.

<sup>10</sup>Die Zeiteinheiten werden auch als *Tick* bezeichnet.

## 1.2 Sozialebene

Betrachtet man die Wissenschaft als soziales System, lässt sich dieses durch die zugrunde liegenden Prozesse<sup>11</sup> und weniger durch die Überzeugungen<sup>12</sup> der Akteure beschreiben. Gerade die Dynamik zwischen den Prozessen, welche sich direkt auf die Überzeugungen beziehen oder zwischen Gruppen stattfinden, stellen ein sehr komplexes Feld da, welches hier leider nicht im Detail besprochen werden kann.

Ein umfangreicherer Ansatz zur Simulation dieser Dynamik ist etwa in Hofmann, *Dynamik sozialer Praktiken und ihrer zu Grunde liegenden Einstellungen - Modellierung und Simulation* zu finden. Hier wird die Dynamik von sozialen Praktiken innerhalb von Gruppen in einer Computersimulation dargestellt. Hofmann versteht unter sozialem Praktiken das wiederholten von Mustern gesellschaftlichen Verhaltens<sup>13</sup>.

Forschungshandlungen, also Handlungen mit dem Ziel der Wissenserzeugung<sup>14</sup> können folglich auch als soziale Praktiken verstanden werden. Die agierenden Wissenschaftler gliedern sich hierbei in Gruppen und führen gemeinsame Handlungen aus. Sie forschen, diskutieren, reflektieren, vergleichen, bewerten und publizieren. Durch die Handlung des Forschens werden *Fakten* gewonnen. Diese werden in Diskussionen und Publikationen verbreitet. Durch die Reflexion und den Vergleich wird eine Bewertung ermöglicht. Auf weitere Details werde ich in der vorgestellten Beispielkonfiguration der Simulation eingehen.

### 1.2.1 Die soziale Simulation

In der Simulation werden Handlungen von Agenten, also den Akteuren im System, ausgeführt. Agenten repräsentieren dabei die Wissenschaftler, also die *Akteure der Wissenschaft*. Sie können innerhalb definierte Handlungen ausführen, welche sich auf ein dynamisches Überzeugungssystem beziehen und sind somit *rational agierende Akteure*.

Ziel der Agenten ist es Daten zu sammeln und den Passungsgrad zwischen Daten

---

<sup>11</sup>Vgl. Krohn und Küppers, *Die Selbstorganisation der Wissenschaft - Wissenschaftsforschung Report 33 Science Studies*, S.22f

<sup>12</sup>Vgl. Krohn und Küppers, *Selbstorganisation: Aspekte einer wissenschaftlichen Revolution (Wissenschaftstheorie, Wissenschaft und Philosophie 29)*, S.307

<sup>13</sup>Vgl. Hofmann, *Dynamik sozialer Praktiken und ihrer zu Grunde liegenden Einstellungen - Modellierung und Simulation*, S.9

<sup>14</sup>Vgl. Balzer, *Die Wissenschaft und ihre Methoden*, S.37

und Hypothesen möglich gering zu gestalten. Liegt ein schlechtes Passungsverhältnis zwischen Daten und der vom Agenten vertretenen Hypothese vor, kann die Überzeugung zu den Daten als auch zur Hypothese geschwächt werden. Des Weiteren versuchen die Agenten ihre Überzeugungen mit weiteren Messungen zu belegen. Die Handlungen der Agenten sind an Wissensbasen geknüpft und je nach subjektiver Faktenlage und Passung von Modellen und Daten oder entsprechend Daten und Modellklassen, können sich diese Wissensgrundlagen ändern.<sup>15</sup>

### 1.2.2 Das Überzeugungssystem

Für die Konstruktion der Agenten auf ist eine präzise Darstellung der Überzeugungsbasis notwendig. Für die Simulation wird ein reduzierter *Überzeugungsbegriff* verwendet. Jedem Satz innerhalb einer Überzeugungsbasis eines Agenten ist eine Bewertung zugeordnet, welche die Überzeugungsstärke zu einem Satz darstellt. Eine feiner granuliert Unterscheidung zwischen Überzeugungen und Wissen, etwa in der Form von *gerechtfertigten Überzeugungen*, werde ich hier nicht ziehen. Ein separater Wissensbegriff, wie etwa in Fagin u. a., *Reasoning about knowledge* wird ausgespart und ist auch nicht Thema dieser Arbeit. Doch ließe sich der Begriff der *Rechtfertigung* sicherlich durch den Grad der Überzeugung und die entsprechende Entstehungsgeschichte dieser Überzeugung definieren.

Inhalt der Überzeugungsbasis sind Sätze. Also etwa Daten aus Messungen, aber auch Hypothesen und Theorien. Die Überzeugung zu diesen Sätzen wird je nach Faktenlage neu bewertet und kann selbst Grundlage für neue Überzeugungen sein.

Die Überzeugung lässt sich zunächst als einfaches zweistelliges Prädikat darstellen  $believe(X, Y)$ . Ein Agent  $agent(a)$  glaubt an eine Aussage  $proposition(p)$ :  $believe(a, p)$ . Aufbauend auf dieser einfachen Notation lässt sich entsprechendes Prädikat in Prolog konstruieren und noch um die Stärke  $believe$  und Stufe  $order$  erweitern:

```
believe([Order, Agent, Believe, Content]).
```

Die Stufe beschreibt hierbei die Ebene der Überzeugung. In der Simulation werden die Überzeugungssätze auf die erste Stufe beschränkt. Jedoch besteht prinzipiell die Mög-

---

<sup>15</sup>Ich werde nicht genauer auf erkenntnistheoretische Fragestellungen zum Wissens-, Überzeugungs- und Glaubensbegriff eingehen, sondern mich auf die erwähnte Literatur stützen.

lichkeit auch Überzeugungen höherer Stufen zu konstruieren.<sup>16</sup> Beispiel für einen höherstufingen Satz wäre etwa:

**Beispiel 1** *Agent Anna glaubt an die Aussage: „Es regnet gerade draußen.“ . Nun gibt es einen weiteren Agent Bernd, welcher glaubt, dass Agent Anna die Aussage „Es regnet gerade draußen.“ glaubt. Daraus ergibt sich folgendes Beispiel:*

$$\text{believe}(\text{Bernd}, \text{believe}(\text{Anna}, \text{„Es regnet gerade draußen.“}))$$

Die Überzeugung wird durch einen Wert  $n \in \mathbb{R}$  ausgedrückt. Dieser Wert steht für die Stärke der Überzeugung zu dem Inhalt des Satzes und ist ein Wert  $0 \leq n \leq 1$ . Ist  $n = 0$  so wird ein Fakt für Falsch erachtet, wohingegen  $n = 1$  eine absolute Überzeugung ausdrückt. Der Satz  $\text{believe}([1, \text{a21}, 0.8, \text{fact}(1, 1, \text{m}(\text{p1}, \text{t1}, 21))])$ . beschreibt, dass Agent *a21* mit einer Stärke von *0,8* von dem Fakt  $\text{fact}(1, 1, \text{m}(\text{p1}, \text{t1}, 21))$  überzeugt ist.

### 1.3 Strukturebene

Eine Theorie *T* besteht aus den Komponenten:<sup>17</sup>

$$T = \langle M, I, D, U \rangle$$

Die Modelle der Theorie *T* werden in der Modellklasse *M* zusammengefasst. Die intendierten Systeme werden mit *I* bezeichnet. Wichtig ist, dass alle Daten  $x \in D$  einer Untersuchung jeweils aus dem gleichen, fest definierten Systeme stammen.

Die Messmethoden sind entscheidend für die Datengewinnung. Je nach vorliegender Messapparatur können Daten genauer und effektiver gewonnen werden und entsprechend anderes fallen auch die späteren Datenstrukturen aus, was wieder Auswirkungen auf die Passung hat. Die Art der Datengewinnung hat einen starken Einfluss auf die Daten selbst und so müssen die Daten immer im Bezug zur Messapparatur interpretiert werden <sup>18</sup>.

---

<sup>16</sup>Dies führt in der Praxis noch zu weiteren Hürden, wie der Berechenbarkeit solcher Sätze.

<sup>17</sup>Wobei ich mich auf die vereinfachte Definition aus Balzer, *Die Wissenschaft und ihre Methoden* beziehe

<sup>18</sup>Der Einfluss der Messmethode je nach Modell und Anwendungsbereich unterschiedlich stark ausgeprägt

Der Approximationsapparat wird mit  $U$  beschrieben. Eine Theorie ist *brauchbar* wenn Modelle und Daten mit einer gewissen *Güte* zusammenpassen<sup>19</sup>. Eine Theorie muss ebenso zwei Bedingungen erfüllen um als solche zu gelten.

1. Wenn Modelle und Datenstrukturen nicht zusammenpassen, scheint das Modell nicht richtig gewählt und die Theorie inhaltsleer. Die Theorie sagt anscheinend nichts über die Daten aus.
2. Das gegenteilige Extrem ist aber auch nicht erstrebenswert: Ein Modell, welches immer erfüllt wird, und somit nicht widerlegbar ist, macht einen Diskurs unmöglich und ist ebenso nicht erwünscht.

Die Passung ist gerade in den empirischen Wissenschaften von besondere Bedeutung. In der Praxis werden kaum ideale Passungen erreicht. Messfehler und Ungenauigkeiten beeinträchtigen bereits die Generierung der Daten. Neben den teils technischen Problemen bei der Konstruktion und Konzeption geeigneter Messapparaturen und Messmethoden, gibt es auch prinzipielle Probleme, welche eine Unschärfe beherbergen.

### 1.3.1 Strukturen

Eine Struktur  $x$  besteht aus einer Menge  $D$  von Hauptbasismengen  $D_1, \dots, D_k$  und Relationen über diese Daten  $R_j$ <sup>20</sup>.

$$x = \langle D_1, \dots, D_m, R_1, \dots, R_n \rangle$$

Zählt man die Menge der mathematischen Objekte  $A$  hinzu, ergibt sich folgende Darstellung:

$$x = \langle D_1, \dots, D_k, A_1, \dots, A_m, R_1, \dots, R_n \rangle$$

So ist eine Struktur  $x$  ein Modell für eine Theorie, wenn  $D_1, \dots, D_i, A_1, \dots, A_j, R_1, \dots, R_k$  existieren, in dieser Theorie vorkommen und die Hypothesen durch genau die Daten erfüllt werden. Im Laufe der Simulation soll gerade gezeigt werden, *wie stark* die Hypothesen durch die Daten erfüllt werden. Die Modellklasse beherbergt alle Strukturen, welche die Hypothesen erfüllen.

Teilstrukturen der Form  $x' = \langle D'_1, \dots, R'_1 \rangle$  von  $x = \langle D_1, \dots, R_1 \rangle$  sind jeweils Teilmengen

---

<sup>19</sup>Vgl. Balzer, *Die Wissenschaft und ihre Methoden*, S.58

<sup>20</sup>Vgl. Balzer und Moulines, "Introduction", S.6

der Komponenten der Hauptbasismengen und Relationen.  $D'_1 \subseteq D_1$  und  $R'_n \subseteq R_n$ <sup>21</sup>. Dies lässt abgekürzt folgend notieren:

$$x' \sqsubseteq x$$

Es gilt: Jede Struktur beinhaltet sich selbst als Teilstruktur:

$$x \sqsubseteq x$$

### 1.3.2 Datenstruktur

Da Daten als einfache Sätze und nicht mengentheoretisch gegeben sein können, müssen diese unter Umständen erst aufbereitet werden indem sie in Datenstrukturen überführt werden. Dies ist notwendig um die Passung zwischen den Daten und Modellen zu berechnen. Auf diese Weise können dann die beiden Strukturen miteinander verglichen werden. Es sollen atomare Relationen in mengentheoretische Sätze überführt werden<sup>22</sup>. Aus einer Relation  $R(a_1, \dots, a_n)$  wird ein mengentheoretischer Ausdruck  $\langle a_1, \dots, a_n \rangle \in R$  und aus einer Funktion  $F(a_1, \dots, a_n)$  wird  $\langle a_1, \dots, a_n, b \rangle \in F$ <sup>23</sup>. Zur Erzeugung der Datenstrukturen sind drei Schritte notwendig<sup>24</sup>:

1. Zunächst werden alle Daten in die mengentheoretische Form überführt. So entstehen zu jeder Relation n-Tupel von Daten welche diese erfüllen. Es entstehen Daten in der Form:

$$\langle a_m^s, \dots, a_n^t \rangle \in R_p$$

2. Im zweiten Schritt werden alle Tupel welche eine Relation erfüllen zu einer Menge zusammengefasst.  $R^+ = \{ \langle a_m^s, \dots, a_n^t \rangle, \dots \}$
3. Zuletzt werden die Objekte entsprechend der Zugehörigkeit auf die Grundmengen  $\langle D_1, \dots, D_k, A_1, \dots, A_m, R_1, \dots, R_n \rangle$  verteilt.

---

<sup>21</sup>Vgl. Balzer, *Die Wissenschaft und ihre Methoden*, S.196

<sup>22</sup>Vgl. ebd., S.190

<sup>23</sup>Da die Simulation auf das Vorkommen von positiven Datenstrukturen beschränkt ist, werde ich mich im Folgenden ebenfalls auf diese beschränken.

<sup>24</sup>Vgl. Balzer, *Die Wissenschaft und ihre Methoden*, S.191



### 1.3.3 Passung

Balzer definiert die ideale Passung zwischen einem Modell  $x$  und einer Datenstruktur  $z$  folgend:<sup>25</sup>

**Definition 1 (Passung)** *Eine (positive) Datenstruktur  $z^+$  passt zu einem Modell  $x$  genau dann, wenn jede Komponente aus  $D$  eine Teilmenge der entsprechenden Komponente aus  $M$  ist:*

$$z^+ \sqsubseteq x$$

Wobei  $z$  und  $x$  von der selben Struktur  $\langle D_1, \dots, D_k, A_1, \dots, A_m, R_1, \dots, R_n \rangle$  sein müssen.

Für die Passung zwischen Datenstrukturen (siehe 1.3.2) und Modellklassen gilt folgendes:

**Definition 2 (Passung zwischen Datenstrukturen und Modellklassen)** *Eine Menge  $X$  von Datenstrukturen passt zu einer Modellklasse  $M$  genau dann, wenn  $\forall x \in X \exists y \in M$  die Passungsbedingung  $x \sqsubseteq y$  gilt*

Daraus ergibt sich folgende Definition für die Passungsbedingung:

**Definition 3 (Passungsbedingung)** *Für alle Datenstrukturen  $x$  aus  $D$  gibt es ein Modell  $z$  aus  $M$  so das gilt:  $x$  passt zu  $z$*

Ist die Passung nicht ideal, also treffen die gerade beschriebenen Bedingungen nicht zu, wird die Passung zwischen den Modellen und gebildeten Datenstrukturen wird mittels des Passungsgrades  $\epsilon$  beschrieben. Datenstruktur(en) und Modell(e) passen mit dem Grad  $\epsilon$  zueinander, wobei  $0 \leq \epsilon \leq 1$  Um den Passungsgrad zu bestimmen wird eine Hilfsstruktur  $y$  eingeführt. Diese stellt eine idealisierte Struktur da, mit welchen die zu testende Datenstruktur  $x \in D$  verglichen wird.

**Definition 4 (Approximative Passung)**  $\forall z \exists x \exists y : x \sqsubseteq y \wedge d(y, z) \leq \epsilon$

Der Abstand zwischen  $y$  und  $x$  muss nun  $\leq \epsilon$  sein. Um dem Abstand zu bestimmen, muss ein Abstandsbegriff gewählt werden, der es erlaubt, das zwei voneinander

<sup>25</sup>Vgl. Balzer, *Die Wissenschaft und ihre Methoden*, S.204-212

## 1 Einleitung

verschiedene Objekte den Abstand  $d(X, Y) = 0$  voneinander haben und zum anderen aber auch einen unendlichen Abstand zueinander haben können. Der Abstand  $d(X, Y)$  wird deshalb innerhalb eines quasimetrischen Raumes  $\langle X, d \rangle$  bestimmt.

**Definition 5 (Quasimetrik)** Eine Funktion  $d$  bildet  $X \times X$  auf  $\mathbb{R}^*$  ab, wobei  $X \neq \emptyset$ <sup>26</sup>.<sup>27</sup> <sup>28</sup>. In diesem speziellen Fall nutzen wir diese Funktion, um den Abstand von  $D \times D$  zu bestimmen und einen qualitativen Zahlenwert für die Passung zu erhalten.

$$d : D \times D \rightarrow \mathbb{R}^*$$

$\forall x, y, z \in X$  gilt: Der Abstand eines Punktes zu sich selbst ist konstant 0.

$$d(x, x) = 0 \tag{1.1}$$

und die Dreiecksungleichung

$$d(x, y) \leq d(x, z) + d(z, y) \tag{1.2}$$

sowie die Symmetrie

$$d(x, y) = d(y, x) \tag{1.3}$$

$$d(x, y) \geq 0 \tag{1.4}$$

Die praktische Vorgehensweise in der Simulation unterscheidet sich jedoch von dem hier gezeigten Passungsbegriff. Der Unterschied wird in Kapitel 2 mit anhand des Programms genauer dargestellt.

---

<sup>26</sup>Vgl. Querenburg, *Mengentheoretische Topologie, zweite neubearbeitete und erweiterte Auflage*, S.7f und S.122

<sup>27</sup>Vgl. Balzer, *Die Wissenschaft und ihre Methoden*, S.215

<sup>28</sup> $\mathbb{R}^* = \mathbb{R}^+ \cup \{\infty\}$  wobei gilt  $\forall x \in \mathbb{R}^+ : x < \infty, x + \infty = \infty, \infty + \infty = \infty$

## 1.4 Die Zeit

Um in der Simulation sowohl zeitliche Abfolgen (Sequenzen) und gleichzeitig eintretende (parallele) Ereignissen darstellen zu können, wird der gesamte Ablauf in Zeiteinheiten gegliedert. Ein Computer, entsprechend der von-Neumann Architektur, ist in der Lage Befehlsketten sequenziell pro Rechenkern zu verarbeiten. Dies geschieht in so einer hohen Geschwindigkeit, dass auch komplexe Operationen als Parallel erscheinen können, dies jedoch nicht sind. Computerprogramme bestehen in ihrer einfachsten Form ebenfalls aus einer Sequenz von Befehlen, welche der Reihe nach abgearbeitet werden.<sup>29</sup> Um parallel Ereignisse darzustellen, muss eine Methodik gewählt werden, welche sowohl sequenzielle als auch parallel Ereignisse gestattet. Wie bereits erwähnt, wird die Zeit in diskrete Zeiteinheiten unterteilt. Diese einzelnen Zeiteinheiten werden *die Ticks* genannt. Jede Aktion erhält einen Zeitstempel, welcher angibt wann diese ausgeführt werden soll. Nun können die Berechnungen sequenziell ausgeführt werden, gelten jedoch als parallel.

### Parallelität

Aktionen  $a$  und  $a'$  die während des gleichen Ticks  $t \in T$  im gleichen Simulationslauf  $r \in R$  ausgeführt werden, finden parallel statt. Die Auswirkungen von Aktionen während eines Ticks werden erst im dem darauf folgenden zugänglich. Dadurch wird gewährleistet, dass sich parallele Ereignisse auf die Gleiche Faktenbasis beziehen. Die Anzahl der Ticks ist zu Beginn der Simulation definiert. Es gilt, je mehr Ticks der Simulation zur Verfügung stehen, desto genauer kann die *Auflösung* gewählt werden. Die Auflösung beschreibt im Grunde das Verhältnis von Ticks zu Handlungen. Wählt man eine größere Anzahl von Ticks, ist es möglich Aktionen über mehrere Ticks laufen zu lassen und so komplexere Regelwerke zu erstellen in dem man Agenten z.B. Geschwindigkeiten zuweist: Agent 1 kann die Aktion  $a$  in 5 Ticks ausführen, wohingegen Agent 2 nur 3 benötigt. Es ermöglicht also eine feinere Rasterung in der Simulation. Dies macht sich jedoch erst bei komplexeren Beispielen bemerkbar und ist in der hier vorgestellten Simulation zu vernachlässigen.

---

<sup>29</sup>Die Nutzung von *Nebenläufigkeit* wird hier nicht beachtet. Im Prinzip lässt sich durch nebenläufiges Programmieren ein Prozess in *Threads* aufteilen, welche an einzelne Recheneinheiten, wie CPUs oder GPUs verteilt werden können. An dieser Stelle geht es jedoch zunächst um den konzeptionellen Hintergrund zur Abbildung der Simulationszeit.

## 1.5 Prolog

In diesem Abschnitt werde ich die grundlegenden, von mir verwendeten Strategien in Prolog erläutern. Prolog ist eine deklarative, dynamisch typisierte Programmiersprache und orientiert sich an der Syntax der Prädikatenlogik (PL1). Dadurch ist das Programm mit etwas Vorwissen schnell lesbar und verständlich. Prolog Programme bestehen aus Fakten<sup>30</sup>. Fakten können aus *Atomen, Variablen, konstanten und komplexen Ausdrücken bestehen*. Atome sind einfache Zeichenketten. Um Atome von anderen Konstrukten, wie etwa Variablen zu unterscheiden, ist es notwendig diese zu kennzeichnen:

```
1 atom
2 'Atom'
```

Variablen beginnen mit einem Großbuchstaben. Diese können in komplexen Ausdrücken gebunden werden. Komplexe Ausdrücke<sup>31</sup> bestehen aus einem Funktor und einer Menge von Argumenten:

```
1 ausdruck(Argument_1,Argument_2,Argument_n).
```

Relationen lassen sich in Prolog über Regeln ausdrücken. Diese lassen sich in einen Kopf und einem Regelkörper aufteilen:

```
1 Kopf :-Koerper.
```

Das Konzept entspricht der Idee von Horn-Klauseln:  $Regel_0 \wedge Regel_1 \wedge \dots \wedge Regel_i \rightarrow Kopf_j$ . Das Antecedens bezeichnet den Körper und das Konsequenz als Regelkopf. Also gilt schematisch:  $Körper \rightarrow Kopf$  und äquivalent in Prolog: `kopf:-koerper..`

Die Konjunktion  $p \wedge q$  wird in Prolog durch `p,q` und die Disjunktion  $p \vee q$  durch `p;q` gekennzeichnet. Die Negation ist ein etwas komplexerer Fall. Hier gibt es zwei Prinzipien. Im Prinzip versucht Prolog den entsprechenden Ausdruck zu beweisen und kehrt den Erfolgsstatus um (Erfolg oder Fehlschlag). Der `neg/1` Operator ist genau dann wahr, wenn das Argument nicht von Prolog bewiesen werden kann. In aktuellen Interpretern wird anstelle des `neg/1` Operators, der Beweisbarkeitsoperator `!\1` in Kombination mit der Bitumkehrung `+\1` verwendet: `!\+1`. Auf weitere Spezialfälle und Feinheiten in der Syntax und Semantik von Prolog werde ich nicht eingehen. Zum genaueren Verständnis sei auf die Literatur in der Einleitung verwiesen.

<sup>30</sup>Fakten werden auch oft als Terme bezeichnet.

<sup>31</sup>Diese werden auch als gebunden Terme oder Prädikate bezeichnet

## 1 Einleitung

In Prolog stehen keine Mengen als Konstrukte zur Verfügung. Doch gibt es Listen, welche verwendet werden können und auch als (eingeschränkte) Menge interpretiert werden können. Hierbei gibt es zwei wichtige Unterschiede zwischen Mengen und Listen:

- In Listen können Elemente mehrmals vorkommen.
- Listen sind also geordnet. Jedes Element hat eine feste Position in der Liste.

Will man eine Liste als Menge interpretieren muss man also zwei Schritte beachten:

1. Doppelte Vorkommen müssen eliminiert werden
2. und es müssen die Vorkommen von Elementen unabhängig von der Position geprüft werden.

Um Listen zu durchsuchen und einzelne Elemente unabhängig von ihrer Position in der Liste zu selektieren steht `member/2` zur Verfügung. Die Idee ist hierbei: Ein Element kommt entweder im Kopf der Liste vor oder im Rest. Durch Rekursion lassen sich auf diese Weise lassen sich alle Elemente untersuchen. So kann die Liste etwa sortiert, die Länge bestimmt oder mit anderen Listen verglichen werden:

```
1 member(X, [X | _]).
2     member(X, [_ | Rest]) :-
3     member(X, Rest).
```

Mit Hilfe von `member/2` lassen sich nun wichtige Listenoperationen definieren: In SWI-Prolog steht eine Funktion `list_to_set/2` zur Verfügung, welche die doppelten Vorkommen von Elementen entfernt.

```
1 list_to_set([], []).
2 list_to_set([Head|Tail], Rest) :-
3     member(Head, Tail), !,
4     list_to_set(Tail, Rest).
5 list_to_set([Head|Tail], [Head|Rest]) :-
6     list_to_set(Tail, Rest).
```

In der Simulation wird mehrfach die Länge von Listen bestimmt, dazu wird der `length/2` Operator verwendet. Dieser ist ebenfalls bereits in SWI-Prolog integriert. Die Idee hierbei ist auch sehr simpel: Zunächst wird die leere Liste abgesondert `[]`. Diese hat konstant eine Länge von 0. Danach wird rekursiv Element für Element abgespalten und

die Anzahl der Rekursionsschritte gezählt und zurückgegeben:

```

1 length([], 0).
2 length([_|Rest], L):-
3   length(Rest, LR),L is LR+1.
```

Mithilfe dieser beiden Prädikate lassen sich eine Vielzahl von Mengenoperationen auf Listen verwirklichen. Natürlich sind diverse Operationen, wie etwa das Vergleichen zweier Mengen, mit der Rückführung auf `member/2` nicht sehr performant. So macht es Sinn hier weitere Strategien zu bedenken: Entfernt man doppelte Vorkommen, und sortiert die Listen beispielsweise, können diese direkt über die von SWI-Prolog bereitgestellten Prädikate verglichen werden.

Diese drei Prädikate sollten als Beispiele dienen und gleichzeitig die Funktion in der Simulation aufzeigen.

### 1.5.1 Eine Theorie in Prolog

Nachdem die Programmiersprache Prolog mit den wichtigsten Bestandteilen umschrieben und in 1.3 der Strukturbegriff kurz dargestellt wurde, will ich nun an einem Beispiel die Umsetzung in Prolog erläutern.

**Beispiel 2 (Die klassische Stoßmechanik in Prolog)** *Als Beispiel für ein komplexeres Modell dient hierbei die klassische Stoßmechanik, welche oft als Beispiel für das strukturalistische Theoriekonzept beschrieben wird. Da in dem verwendeten Modellbegriff<sup>32</sup> keine Trennung zwischen  $M_p$  und  $M$  verwendetet wird, ist diese in folgender Definition auch nicht gegeben<sup>33</sup>. Ein Modell für die klassische Stoßmechanik hat folgende Form:*

**Definition 6 (KSM)**  *$x$  ist eine klassische Stoßmechanik, genau dann, wenn  $P, T, \mathbb{R}, \mathbb{R}^3, v, m$  existieren und  $x = \langle P, T, \mathbb{R}, \mathbb{R}^3, v, m \rangle$  und folgende Bedingungen gelten:*

1.  $P \neq \emptyset$  und endlich
2.  $T := \{t_1, t_2\}$
3.  $t_1, t_2 \in \mathbb{R}$  und  $t_1 < t_2$  4.  $v : P \times T \rightarrow \mathbb{R}^3$  und  $m : P \rightarrow \mathbb{R}$

<sup>32</sup>Nach Balzer, *Die Wissenschaft und ihre Methoden*

<sup>33</sup>Entsprechende Trennung in 6 wäre in Zeile 5 und 6 zu treffen.

## 1 Einleitung

$$5. \forall p \in P : m(p) > 0$$

$$6. \sum_{p \in P} m(p) * v(p, t_1) = \sum_{p \in P} m(p) * v(p, t_2)$$

Zunächst ist es möglich die Definitionen nahezu direkt zu übernehmen:

```
1 pot_modell(ksm,X):-  
2   X=[P,T,V,M],  
3   not(P = []),  
4   member(T,[t1,t2]),  
5   member(Pp,P),  
6   v(Pp,T,V),  
7   m(Pp,M),  
8   true,  
9   M>0.
```

Der Vektor für ein  $p \in P$  zu einem Zeitpunkt  $t_n \in T$  hat nun die Form  $v(Pp,T,V)$  und die Masse  $m$  von  $p$  ist  $m(Pp,M)$ . Bei einer Trennung zwischen  $M_p$  und  $M$  ist das Modell noch einmal separat zu definieren:

```
1 modell(ksm,X):-  
2   X=[P,T,V,M],  
3   pot_modell(ksm,X),
```

Der Impuls  $v$  eines Teilchens  $p \in P$  zu  $t_1$  wird durch den Aufruf `impuls(P,t1,Impuls_t1)` berechnet. Davon ausgehend wird dann der Gesamtimpuls für alle  $p$  zu  $t_1$  berechnet und in `Ges_Impuls` ausgegeben.

```
1   impuls(P,t1,Impuls_t1),  
2   ges_impuls(Impuls_t1,Ges_Impuls),
```

Analog wird nun der Impuls zu  $t_2$  berechnet. Die Gleichung besteht nun darin, dass der Gesamtimpuls zu  $t_2$ , also `ges_impuls(Impuls_t2,Ges_Impuls)`, den gleichen Wert wie zu  $t_1$  haben muss.

```
1   impuls(P,t2,Impuls_t2),  
2   ges_impuls(Impuls_t2,Ges_Impuls).
```

## 1 Einleitung

Da die Berechnung des Impulses und des Gesamtimpulses keine Funktionen in Prolog sind, müssen diese auch definiert werden:

**Definition 7 (Impuls)** *Der Impuls ergibt sich aus der Masse  $m$  multipliziert dem Vektor der Geschwindigkeiten in die gegebenen Dimensionen  $\vec{v}$ :  $\vec{p} = m \cdot \vec{v}$*

```
1 impuls([P|Rest],T,[Ergebnis|Rest2]):-
2   m(P,M),
3   v(P,T,V),
4   n_v_multi(M,V,Ergebnis),
5   impuls(Rest,T,Rest2).
6 impuls([],T,[]).
```

**Definition 8 (Gesamtimpuls)** *Addiert man die Einzelimpulse des Systems zu einem Zeitpunkt ergibt sich der Gesamtimpuls.*

```
1 ges_impuls([Imp|Rest],Ergebnis):-
2   ges_impuls(Imp,Rest,Ergebnis).
3
4 ges_impuls(V1,[V2|Rest],Ergebnis):-
5   v_v_add(V1,V2,E),!,
6   ges_impuls(E,Rest,Ergebnis).
7 ges_impuls(X,[],X).
```

Die Notwendigen Operationen sind wie folgend definiert: `v_v_add(X,Y,Z)` addiert zwei Vektoren.

```
1 v_v_add([],[],[]).
2 v_v_add([A|Rest1],[B|Rest2],[Neu|Liste]):-
3   Neu is A+B,
4   v_v_add(Rest1,Rest2,Liste).
```

Mit `n_v_multi` werden zwei Vektoren multipliziert.

```
1 n_v_multi(Zahl,Vektor,Ergebnis):-
2   n_v_multi(Zahl,Vektor,Temp,Ergebnis).
3
```



## 1 Einleitung

```
4 n_v_multi(Multi, [X|Rest], Temp, Ergebnis):-  
5   Zw is Multi*X,  
6   add(Zw, Temp, Neu_L),!,  
7   n_v_multi(Multi, Rest, Neu_L, Ergebnis).  
8 n_v_multi(Multi, [], Temp, Ergebnis):-Ergebnis = Temp.
```

Und add fügt ein Element an eine gegebene Liste an.

```
1 add(Element, [], [Element]).  
2 add(Element, [Kopf|Rest], [Kopf|Neueliste]):-  
3   add(Element, Rest, Neueliste).
```

Ein Modell für die *KSM* hat nun in Prolog die Form  $[P, T, V, M]$ . Somit ist die Übersetzung von *x ist ein Modell der klassischen Stoßmechanik* in `modell(ksm, X)` abgeschlossen.

## 2 Die Simulation

In diesem Kapitel wird der Ablauf der Simulation mit der Umsetzung in Prolog beschrieben. Zunächst werde ich die zentralen Konzepte anhand der Agenten und Handlungen beschreiben. Darauf aufbauend folgen der Aufbau der Simulation und eine Funktionsbeschreibung der einzelnen Module.

### 2.1 Die Agenten

In der vorgestellten Konfiguration gibt es zwei Gruppen von Agenten: Zum einen die Studenten und zum anderen die Wissenschaftler. Neben generischen Handlungen, wie etwa der *Generierung von Daten*, stehen jeder Gruppe spezifische Handlungen zur Verfügung.

<sup>34</sup>

1. Die erste Gruppe beinhaltet die *Studenten*. Handlungen dieser Gruppe sind etwa *die Teilnahme an Seminaren* oder *das Verfassen einer Arbeiten*.
2. In der zweiten Gruppe befinden sich *die Professoren*. Professoren *halten Seminare* oder *überprüfen Fakten*.

Der Agententyp der jeweiligen Agenten ist in der Datei `agents.pl` als Fakt der Form `fact(Lauf, Zeitpunkt, agent_type(Typ, Agent))` hinterlegt. Hierbei werden bei den vorgestellten Regeln Lauf und Zeitpunkt nicht beachtet. Die Agenten ändern ihren Typ während der Laufzeit nicht, doch besteht hier die Möglichkeit die Simulation auch dahingehend zu ändern. Ein Agent muss in der Simulation nicht als eigenständiges Objekt hinterlegt werden. Seine Existenz folgt aus dem Vorkommen in den Überzeugungssätzen und Fakten. Soll ein neuer Agent generiert werden, reicht also dessen Verwendung in den Faktenbasen.

---

<sup>34</sup>Auf die genaue Definition der Handlungsmöglichkeiten wird später noch genauer eingegangen.

## 2.2 Die Hypothesen

Die Hypothesen, mit denen die Wissenschaftler in der Simulation arbeiten, sind vorgegeben. Diese sind in diesem Stadium noch von einer sehr einfachen Struktur und dienen als Beispiel.

### Hypothese H1

Es existieren zwei Grundmengen  $D_1$  und  $D_2$ . Etwa die Menge aller Punkte  $D_1$  und die Menge aller Geraden  $D_2$ .

1. Die Menge der Punkte  $D_1$  enthält mindestens drei Elemente
2. Die Menge der Geraden  $D_2$  enthält mindestens ein Element
3. Für alle Punkte aus  $D_1$  gilt, sie liegen mindestens auf einer Geraden aus  $D_2$
4. Für alle Punkte gilt, liegt ein Punkt  $x_3 \in D_1$  zwischen zwei weiteren Punkten  $x_1$  und  $x_2$ , so liegt dieser auf der gleichen Geraden  $y_1 \in D_2$  wie die beiden umgebenden Punkte  $x_1$  und  $x_2$

### Beispiel 3 Ein Modell zu H1 in Prolog

Formal hat die Hypothese H1 folgende Form:

$$\forall x_1 \forall x_2 \forall x_3 \forall y_1 (\text{zwischen}(x_1, x_2, x_3) \wedge \text{auf}(x_1, y_1) \wedge \text{auf}(x_2, y_1) \rightarrow \text{auf}(x_3, y_1))$$

Die Relationen sind in

$$\text{zwischen}(D_1 \times D_1 \times D_1)$$

und

$$\text{auf}(D_1 \times D_2)$$

enthalten.

Als Beispiel sind nun vier Punkte  $p_1, p_2, p_3, p_4$  sind aus  $D_1$  auf einer Geraden  $g_1 \in D_2$  gegeben, wie in folgender Grafik dargestellt ist:

Abbildung 2.1: Modell für H1

## 2 Die Simulation

Daraus ergeben sich folgende Sätze:

$zwischen(p1, p3, p2), auf(p1, g1), auf(p2, g1), auf(p3, g1)$

$zwischen(p2, p4, p3), auf(p2, g1), auf(p4, g1), auf(p3, g1)$

Die Relation **zwischen** besteht aus Tupeln der folgenden Mengen: **zwischen** =  $\{\langle p1, p3, p2 \rangle, \langle p2, p4, p3 \rangle\}$   
Sowie **auf** mit **auf** =  $\{\langle p1, g1 \rangle, \langle p2, g1 \rangle, \langle p3, g1 \rangle, \langle p4, g1 \rangle, \}$ . In Prolog lässt sich dies folgend darstellen: Die Hypothese wird in das Prädikat **h1** überführt.

```
1 h1: -zwischen(X1, X2, X3), auf(X1, G1), auf(X2, G1) -> auf(X3, G1).
```

Folgende Instanz würden dann als Beispiel dieses Prädikat erfüllen und somit als mögliches Modell für H1 gelten:

```
2 zwischen(p1, p2, p3).
```

```
3 auf(p1, g1).
```

```
4 auf(p2, g1).
```

```
5 auf(p3, g1).
```

```
6 auf(p4, g1).
```

In der Simulation wird die Darstellung aus Praxisgründen etwas komplexer: Hier wird die Hypothese für die Passungsbedingung (siehe Abschnitt ??) in zwei Varianten aufgeteilt. Diese haben zwar inhaltlich die gleiche Aussage, doch laufen sie getrennt voneinander ab. Durch diese Trennung kann im späteren Verlauf die Passung von der Hypothese zur Welt und von den Daten zur Hypothese einzeln betrachtet werden. Zuerst wird die Menge aller Tupel der Relation **zwischen(X1, X2, X3)** gebildet

```
1 findall([PX1, PX2, PX3], fact(1, _, world, zwischen(PX1, PX2, PX3)), L),
```

und danach werden alle Geraden ermittelt.

```
2 findall(PY1, fact(1, _, world, auf(_, PY1)), L2),
```

Nun wird zu allen Tupeln und allen Objekten der Zwischenrelation überprüft, ob Instantiationen gebildet werden können:

```
3 member(T, L),
```

```
4 ( T=[[X1, T1], [X2, T1], [X3, T1]],
```

```
5 fact(Run, _, world, zwischen([X1, T1], [X2, T1], [X3, T1])),
```

```

6     fact(Run,_,world,auf([X1,T1],[Y1,T2])),
7     fact(Run,_,world,auf([X2,T1],[Y1,T2]))->
8     fact(Run,_,world,auf([X3,T1],[Y1,T2]))

```

Die zweite Variante verläuft analog, jedoch werden entsprechend der Datenmengen andere Instantiationen gebildet.

## Hypothese H2

Die zweite Hypothese ist ein einfaches, abstraktes Beispiel: Es gibt nur eine Grundmenge  $D_1$ , welche mindestens zwei Objekte  $x_1$  und  $x_2$  beinhaltet. Für alle Objekte  $x_1$  gibt es mindestens ein Objekt  $x_2$ .

$$\forall x_1 \exists x_2 P x_1 \rightarrow Q x_2$$

In der Simulation werden zunächst die  $x$  aus  $D_1$  mit  $Px$  gesucht.

```

1     findall(PX1,fact(1,_,world,p(PX1)),L),

```

Danach wird zu jedem dieser Elemente ein Weiteres gesucht, um die Existenzbedingung zu erfüllen.

```

2     member(T,L),
3     ( T=[X1,T1],
4     fact(Run,_,world,p([X1,T1]))->
5     fact(Run,_,world,q([X2,T1]))
6     ).

```

## 2.3 Aufbau

Das Programm ist in mehrere Module unterteilt. Jedes dieser Module stellt Funktionalitäten für die Simulation bereit und kann unabhängig erweitert werden. In der Tabelle 2.1 ist eine Übersicht der Module und der beinhalteten Funktionen zu sehen.

Tabelle 2.1: Module

Modul	Beschreibung
actions.pl	Beinhaltet die Handlungsregeln.
agents.pl	In dieser Datei werden die Agentenkonfigurationen festgelegt.
fit.pl	Das Passungsprogramm, welches einen Passungsgrad zurückgibt.
config.pl	Die Startkonfigurationen der Simulation.
function.pl	Erweiterte Funktionen, welche in mehreren Modulen benötigt werden.
simulation.pl	Der Simulationskern mit Ablaufregeln und den Simulationsschleifen.
worldN.pl	Enthält die Definitionen für die simulierte Welt.
output.pl	Generiert die Ausgabe.
urban_cart.pl	Ein Programm von Josef Urban zur Berechnung des kartesischen Produkts <sup>35</sup> .

## 2.4 Ablauf

Die Simulation verläuft rundenbasierend. Die Menge  $T$  beschreibt die Gesamtheit der Zeiteinheiten  $T := \{t_1, \dots, t_{ende}\}$ . Der Verlauf ist folgend:  $t_1 \leq t_n \leq t_{ende}$  und  $n \in \mathbb{N}$ . Zu jedem Zeitschritt  $t_i$  haben die Agenten die Möglichkeit Handlungen durchzuführen. Während der Laufzeit können alle Fakten verändert werden und die Simulation so dynamisch ablaufen.

## 2.5 Die Schleifen

Der Ablauf gliedert sich zunächst in zwei Schleifen: Einer äußeren über die Anzahl der Simulationsläufe `number_of_runs(Wert)` und einer inneren über die einzelnen Zeiteinheiten `number_of_ticks_per_run(Wert)`. Die Aufteilung in diese beiden Schleifen hat folgenden Hintergrund: Die äußere Schleife setzt die Simulation zu jedem kompletten Durchlauf wieder zurück auf die Startkonfiguration und setzt einen neuen Simulationslauf in Gang. Die innere Schleife stellt nun einen Simulationslauf da. Hier wird dann die Simulationszeit von  $t_1$  bis  $t_{ende}$  durchlaufen. Innerhalb dieser Inneren Schleife gibt es noch weitere kleine Schleifen, beispielsweise über die Anzahl der Agenten, um alle Aktionen zu erfassen.

Die Startkonfiguration ist in der Datei `config.pl` hinterlegt. Die äußere Schleife hat in der Simulation folgende Form:

```
1 run(I,Ende) :-
```

## 2 Die Simulation

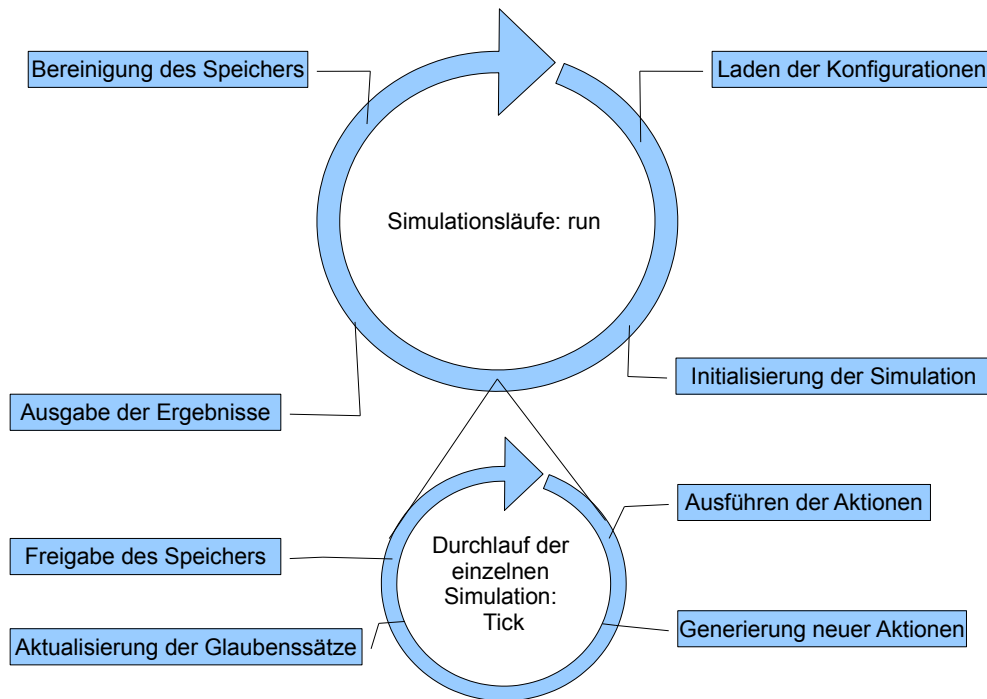


Abbildung 2.2: Schematischer Ablauf der Schleifen

```
2 I < Ende,  
3 retractall(sim_time(,_)),  
4 assert(sim_time(I,t)),  
5 number_of_ticks_per_run(Max_Ticks),  
6 tick(1,Max_Ticks),  
7 I1 is I + 1,  
8 run(I1,Ende).
```

Nach dem Start der Simulation wird die Schleife mittels `run(Startwert,Anzahl_der_L\"aufe)` gestartet. Die Anzahl der Simulationsläufe, also die die Angabe, wie Oft die Simulation durchlaufen werden soll, ist in `config.pl:number_of_runs(Wert)` hinterlegt. Zu jedem Simulationslauf wird einmal die Simulation von Zeiteinheit 1 bis zur definierten Abbruchbedingung durchlaufen. Die Abbruchbedingungen für die Schleife über die Simulationsläufe ist `run(Ende,Ende)`. Jeder Simulationslauf kann mit verschiedenen Parametern erfolgen. So werden in den Überzeugungen, den Fakten und Handlungen, der Simulationslauf als auch der Zeitpunkt hinterlegt, ab diese sie gültig sind. Die Simulationszeit wird als Fakt `sim_time(Lauf,Zeiteinheit)` gespeichert und

zu jedem Durchlauf aktualisiert. Auf diese Weise muss nicht zu jedem Aufruf und in jedem Prädikat die Zeiteinheit erneut übergeben werden, sondern kann von jeder Stelle aus einfach aus der Datenbank als dynamisches Prädikat abgefragt werden.

Analog zur äußeren Schleife, wird nun die innere Schleife, also die eigentliche Simulation über die Zeiteinheiten definiert:

```
1 tick(I,Ende) :-
2   I < Ende,
3   sim_time(Run,Time),
4   retractall(sim_time(_,_)),
5   assert(sim_time(Run,I)),
6   tick(I),
7   I1 is I + 1,
8   tick(I1,Ende).
```

Zu jedem Durchlauf wird hier ein Simulationsdurchlauf `tick(I)` ausgeführt und darauf die Zeit mit `I1 is I + 1` neu gesetzt. Der Ablauf eines Simulationsdurchlaufs wird zu jeder Zeiteinheit erneut in Gang gesetzt. Diese Folge stellt die eigentliche Simulation dar. Zu jedem  $t_n$  werden nun folgende Aktionen durchgeführt:

```
1 tick(Tick):-
2   actions(Tick),
3   retract_old_bel(Tick),
4   retract_old_actions(Tick).
```

Es werden zunächst mit `actions(Tick)` alle zu diesem Zeitpunkt  $t_i$  anstehenden Aktionen, also die Regeln und Handlungen durchgeführt. Danach werden (optional) die alten Daten, also die aktualisierten Überzeugungssätze mit `retract_old_bel(Tick)` und die bereits durchgeführten Handlungen mit `retract_old_actions(Tick)` aus dem Speicher entfernt. Die Abbildung 2.3 stellt den Ablauf mit zwei Aktionen schematisch dar. Es wird zu einem Tick eine Handlung gewählt, die Daten werden getestet und die Datenbank angepasst. Mit den Aktion geschieht nun folgendes:

### 2.5.1 Die Aktionen

Die Aktionen werden nun in zwei Schritten ausgeführt und aktualisiert:



## 2 Die Simulation

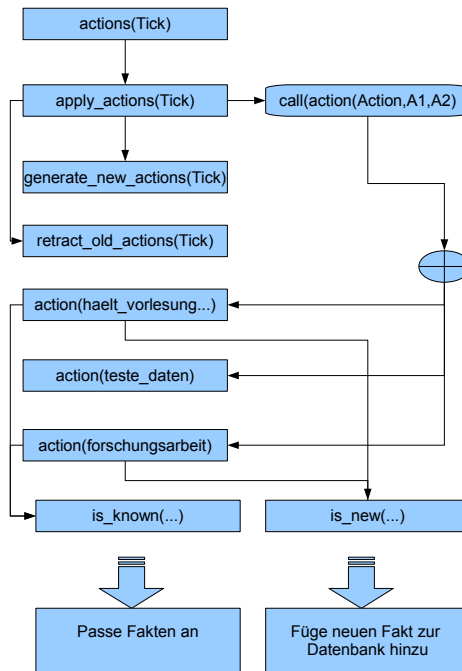


Abbildung 2.3: Ablauf

1. Zunächst werden die Aktionen von Agenten ausgeführt. Es werden also konkrete Handlungen, wie etwa das halten einer Vorlesung, die Überprüfung von Daten oder das Forschen an Daten durchgeführt. Um die Aktionen ausführen zu können werden zunächst alle Aktionen gesammelt, welche zu diesem Zeitpunkt ausgeführt werden sollen. Diese Aktionen werden in einer Liste zusammengefasst: `findAll(Action, fact(Run, Tick, Action), Actions)`. Jedes Element der entstandenen Liste wird nun einzeln selektiert `member(Action, Actions)`, validiert `Action=..[action, Act, Agent1, Agent2]` und aufgerufen `call(Action)`.

Eine Aktion ist nun folgend definiert:

`action(Name_der_Aktion, Ausf\ "uhrender_Agent, [Liste_der_beteiligten_Agenten ])`. Ist z.B. der Fakt `fact(1, 1, action(haelt_vorlesung, d1, [s1, s2]))`. gegeben, wird aus diesem Fakt, zum Simulationslauf 1 und Zeitpunkt 1, die Aktion `haelt_vorlesung(d1, [s1, s2])` für den Agenten `d1` und Angewendet auf die Agenten `s1` und `s2`.

2. Nach dem die Aktionen ausgeführt wurden, müssen neue Aktionen für die folgende Runde generiert werden. Diese Aktionen müssen natürlich zu der jeweili-

gen Sorte von Agenten passen. Agent der Gruppe 1, ein Student, hält z.B. keine Vorlesung. Das Generieren wird durch den Aufruf `generate_new_actions(Tick)` ausgelöst. Um die Aktionen zu verteilen, wird zuerst eine Liste aller Agenten zusammengestellt und zu jedem Agenten der Liste der entsprechende Typ bestimmt: `gfindall(Agents, fact(R, T, agent_type(AT, Agents)), Agent_Set)`. Diese Angaben werden dann an `random_action(Type, Action, Agent, Agent2)` übergeben. Hier wird nun mittels `random_element(Liste, Element)` eine zufällige Aktion aus der Liste der möglichen Aktionen `actions.pl:action_list(Typ, Liste_der_vorhandenen_Aktionen)` bestimmt.<sup>36</sup> Nach dem die Handlung ausgewählt ist, wird festgelegt, auf welche Agenten sich diese Aktion bezieht. Dies wird durch die Handlung selbst gesteuert. Also wird der Aufruf nun genau an diese weitergeben. Je nach selektierter Handlung werden nun die Agenten zurückgeben. Einige Handlungen geben Listen von Agenten zurück (wie etwa `haelt_vorlesung`) wohingegen z.B. die Handlung `fuehrt_messung_durch` von nur einem Agenten ausgeführt wird und keinen Wert zurückgibt. Nun ist klargestellt zwischen welchen Agenten die Handlung durchgeführt werden soll. Die Aktion kann nun für die nächste Zeiteinheit adressiert und zur Faktenbasis hinzugefügt werden. Diese wird nun in der nächsten Runde ausgeführt.

### 2.5.2 Ablauf der Handlungen

In der vorgestellten Konfiguration gibt es vier mögliche Handlungen: `haelt_vorlesung`, `testet_daten`, `fuehrt_messung_durch`, `forschungsarbeit`. Teilweise sind die Handlungen, wie beschrieben, auf Agentengruppen beschränkt und teilweise, werden sie nur von einem Agenten einzeln ausgeführt.

#### **Handlung: `haelt_vorlesung(Agent, Liste_von_Agenten)`**

Die Handlung `haelt_vorlesung(Agent, Liste_von_Agenten)` wird von einem Agenten der Gruppe 2 auf eine eine Liste von Agenten der Gruppe 1 angewendet.

Die Teilnehmer werden über `haelt_vorlesung(Dozent, List_of_all_students)` bestimmt. Hierfür wird eine Liste aller Agenten der Gruppe 1, also der Studenten gebildet

---

<sup>36</sup>Dieser Auswahlprozess kann durch weitere Regeln beeinflusst werden, ist jedoch in dem beschränkten Rahmen dieser Arbeit über einen Zufallsgenerator gesteuert.

und diese zurückgegeben.

Die eigentliche Handlung wird nun über `action(haelt_vorlesung,A1,A2)` aufgerufen. Bei der Ausführung der Handlung werden Fakten aus den Glaubenssätzen des Ausführenden Agenten an die Teilnehmenden übertragen:

```

1 action(haelt_vorlesung,Dozent,Audience_list):-
2   bel([Order_Nr,Dozent,Strength,D_Fact]),
3   findall(Facts,bel([Order_Nr,Dozent,Strength,Facts]),List_of_facts),
4   random_element(List_of_facts,Fact),

```

Zunächst wird eine Liste der Fakten aus den Glaubenssätzen des Agenten gebildet. Daraus wird mittels `lstinlinerandom_element(List_of_facts, Fact)` ein zufälliges Element ausgewählt.

```

5   sim_time(Run,Tick),
6   ( member(Agent,Audience_list),
7   ( is_known(Order_Nr,Agent,Strength,Fact,Tick) ;
8     is_new(Order_Nr,Agent,Strength,Fact,Tick) ) ),fail>true.

```

Nun wird die Liste der Teilnehmende Agenten mittels `member(Agent,Audience_list)` aufgliedert. Es wird jeder der Agenten der Liste auf sein Wissen über den selektierten Fakt getestet:

```

1 is_known(Order_Nr,Agent,Strength,Fact,Tick):-
2   Fact=..[T,Run,Time,Data],
3   sim_time(Run,Tick),
4   N_Time is Tick+1,
5   N_Fact=..[T,Run,N_Time,Data],
6   bel([ON,Agent,Str_d,N_Fact]),
7   retract(bel([Order_Nr,Agent,Str_d,N_Fact])),
8   X is random(5), Multi is (X+5)/10,
9   Str is Str_d + 0.2,
10  New_Strength is Str * Multi,
11  output_double_fact(Fact,Agent,N_Fact),
12  retract(bel([Order_Nr,Agent,Strength,Fact])),
13  assert(bel([Order_Nr,Agent,New_Strength,N_Fact])).

```

Dabei wird der Fakt zunächst in seine Bestandteile aufgliedert, dann die neue Si-

mulationszeit generiert und verglichen ob der Fakt bereits zu diesem Zeitpunkt in der Überzeugungsbasis des Agenten existiert. Trifft dies zu, wird sein *Glaube* an diesen Fakt gestärkt, also der entsprechende Wert erhöht. Sollte der Fakt noch nicht bekannt sein, wird dieser zur Datenbasis des Agenten hinzugefügt.

```
1 is_new(Order_Nr,Agent,Strength,Fact,Tick):-
2     not(bel([Order_Nr,Agent,Strength,Fact])),
3     not(is_known(Order_Nr,Agent,Strength,Fact,Tick)),
4     X is random(5), Multi is (X+5)/10,
5     New_Strength is Strength * Multi,
6     Fact=.. [T,Run,Time,Data],
7     N_Time is Tick+1,
8     N_Fact=.. [T,Run,N_Time,Data],
9     not(bel([ON,Agent,Str_d,N_Fact])),
10    output_new_fact(Fact,Agent,N_Fact),
11    assert(bel([Order_Nr,Agent,New_Strength,N_Fact])).
```

Die Stärke des Glaubenssatzes des Dozenten wird hierbei geringfügig abgeschwächt.

### Handlung: forschungsarbeit

Mit der Handlung `forschungsarbeit` können Fakten von Agenten der Gruppe 1 an die der Gruppe 2 übertragen werden. Jedoch kann hier, anders als bei `haelt_vorlesung` die Handlung immer nur auf einen Agenten, nicht auf eine Liste von Agenten angewendet werden.

Wieder wird über den Aufruf der Handlung (in diese Fall `forschungsarbeit(A,B)`) der betreffende Agent selektiert, an den die Fakten übertragen werden.

Analog zu `haelt_vorlesung` wird nun ein Fakt selektiert und zu den Überzeugungssätzen des zweiten Agenten hinzugefügt. Ein Unterschied besteht jedoch darin, dass nur neue Fakten akzeptiert werden. Sollte der Fakt bereits bekannt sein, wird nicht die Galaubensstärke angepasst.

### Handlung: fuehrt\_messung\_durch

Mit dieser Handlung können Fakten aus der Simulierten Welt in die Überzeugungsbasen übertragen werden. Die Handlung kann von beiden Gruppen ausgeführt werden und die Daten über das halten der Vorlesung oder das Verfassen von Forschungsarbeiten weitergegeben werden. Diese Handlung wird entsprechend nur von einem Agenten ausgeführt und gibt keinen Agenten zurück `fuehrt_messung_durch(_, _)`.

Die eigentliche Aktion selektiert nun einen Fakt der simulierten Welt aus und fügt ihn den Glaubenssätzen des ausführenden Agenten hinzu. Hierbei muss beachtet werden, dass die Daten aus dem Richtigen Simulationslauf selektiert werden.

### Handlung: testet\_daten

Diese Handlung ist die, für die Simulation, interessanteste Regel. Hier wird die Passung zwischen den Fakten innerhalb der Überzeugungssätze der Agenten, den Hypothesen und der Umwelt berechnet. Ein Agent kann hier seine eigenen oder die Fakten eines anderen Agenten auf den Passungsgrad Testen. Die hier gezeigte Version ist stark reduziert und vereinfacht aber lässt dennoch die Verknüpfung zwischen Strukturebene und Sozialebene erkenne.

Zunächst wird der Agent selektiert, dessen Daten getestet werden sollen. Dies kann auch der ausführende Agent selbst sein. Zunächst werden die Hypothesen in den Überzeugungssätzen des zu testenden Agenten gesammelt:

`findall([S,H],bel([ON,Dozent,S,fact(_,_ ,hypothesis(H))]),List_of_Hyp)`. Darauf wird eine aus dieser Liste ausgewählt und es werden alle Überzeugungssätze des Agenten gesucht: `findall(bel([0,Agent,S,F]),bel([0,Agent,S,F]),Data)`. Diese Daten werden dann an das Passungsprogramm weitergegeben. Der hieraus resultierende Wert wird nun mit den Überzeugungssätzen verrechnet und zur eigenen Faktenbasis hinzugefügt. Diese Berechnung ist sehr einfach gelöst: Es gibt nun drei Möglichkeiten:

1. Der Wert der Passung  $p$  ist größer als die Stärke der Überzeugungssätze  $s$ . In diesem Fall wird die neue Stärke  $s'$  der Überzeugungssätze durch  $\frac{s}{p}$  berechnet. Dadurch nähert sich die Stärke dem Wert der Passung an.
2.  $p$  ist kleiner als  $s$ . In diesem Fall wird die neue Stärke  $s'$  der Überzeugungssätze durch  $\frac{p}{s}$  berechnet und der Glaube geschwächt.

- Ist  $p = s$ , so bleibt die Stärke unverändert.

### 2.5.3 Das Passungsprogramm

In der vorgestellten Version des Programms ist die Passung stark an die jeweilige Hypothese angepasst.

#### Passung zu $H_1$

Der erste Schritt zur Berechnung der Passung ist es, die Daten, welche getestet werden sollen in eine verwertbare Form zu bringen.

```
1 h1(Data,Value):-
2   build_temp(Data),
```

Hierzu wird eine temporäre Liste angelegt, welche die zu testenden Fakten enthält:

```
1 build_temp(Data):-
2   member(S,Data),
3   S=..[bel,List],
4   List=[OrderNr,Owner,Bel_level,fact(Run,Time,Fact)],
5   assert(temp(fact,Fact)),
6   fail;true.
```

Es werden die Fakten aus den Überzeugungssätzen extrahiert und in der Form `temp(fact,Fakt)` gespeichert. Nun kann aus dem übergeordneten Blick des Simulationsprogramms getestet werden, welche Daten der Welt, die Hypothese erfüllen:

```
3 findall(W,test_h1_world(W),WL),
```

Es wird nun geprüft welche Tupel  $\langle x_1, x_2, x_3, y_1 \rangle$  die Hypothese  $H_1$  erfüllen.

```
1 test_h1_world([X1,X2,X3,Y1]):-
2   %sim_time(Run,Time),
3   findall([PX1,PX2,PX3],fact(1,_,world,zwischen(PX1,PX2,PX3)),L),
4   member(T,L),
5   ( T=[[X1,T1],[X2,T1],[X3,T1]],
6     fact(Run,_,world,zwischen([X1,T1],[X2,T1],[X3,T1])),
```

```

7     fact(Run,_,world,auf([X1,T1],[Y1,T2])),
8     fact(Run,_,world,auf([X2,T1],[Y1,T2]))->
9     fact(Run,_,world,auf([X3,T1],[Y1,T2]))
10    ).

```

Daraus resultiert nun die Menge  $W = \{\langle x_1, x_2, x_3, y_1 \rangle, \dots\}$  welche genau diese Tupel enthält. Mit `findall(D,test_h1_data(D),DL)` werden nun die Tupel gesucht, welche durch die Daten des Agenten erfüllt werden und in der Menge  $D$  gesammelt. Sofern nun die Kardinalitäten der Mengen  $|W|$  und  $|D|$  größer 0 sind und  $LDL > 0$ ,  $LWL > 0$ , gilt, wird nun der Wert der Passung über `Value is LDL/LWL` gebildet und danach der Speicher von den temporären Daten befreit:

```

5     findall(D,test_h1_data(D),DL),
6     length(WL,LWL),
7     length(DL,LDL),
8     ( ( LDL > 0,
9         LWL > 0,
10        Value is LDL/LWL
11      );
12     Value is 0
13   ),
14   retractall(temp(X,Y)).

```

### 2.5.4 Die Urban Regel

An einigen Stellen der Simulation wird das kartesische Produkt  $D_1, \dots, D_n$  benötigt. Dies geschieht mit Hilfe der Urban Regel, welche nach ihrem entwickler Josef Urban benannt ist.<sup>37</sup> Die einfachste Variante ist das kartesische Produkt über zwei Mengen  $D \times D$ . Durch den Aufruf `cart([a,b,c],[1,2,3],Kartesisches_Produkt)` erhält man durch den Rückgabewert das zu erwartende Ergebnis:

```

Kartesisches_Produkt = [[a,1], [a,2], [a,3], [b,1], [b,2], [b,3], [c,1],
                        [c,2], [c,3]]

```

<sup>37</sup>Josef Urban hat dieses Modul entwickelt und unter: <http://www.lrz-muenchen.de/W.Balzer/urban-rule.pdf> zur Verfügung gestellt

## 2 Die Simulation

Das kartesische Produkt von  $D \times D$  wird nun folgender Weise berechnet: Zuerst wird das erste Element der ersten Liste, also das `[a]` durch den Aufruf `cart([A|ARest],BL,CL)` abgespalten und zusammen mit der zweiten Liste an `create_tuple(A,BL,TList)` übergeben. Hier wird zu an das gegebene `[a]` rekursiv jedes Element der zweiten Liste durch Aussonderung der Elemente der Elemente mit `create_tuple(A,[B|BRest],[[A,B]|TRest])` und `create_tuple(A,BRest,TRest)` in einer neuen Menge angehängt. Es entsteht die Liste `[[a,1],[a,2],[a,3]]`. Nachdem dies geschehen ist, wird das gleiche Verfahren nun mit jedem weiteren Element aus der ersten Liste durchgeführt: `cart(ARest,BL,CInterim,CL)`, und das Ergebnis der jeweiligen Durchläufe zusammengeführt `append(TList,CInterim,CL)`. Das Ganze für eine  $n$  Mengen ( $n > 2$ ), also  $D_1 \times D_2 \times \dots \times D_n$  ist etwas komplexer:

```
1 n_cart(NList,[]):- member(NList,[]),!.
2 n_cart(NList,undef):- length(NList,1),!.
3 n_cart([AL,BL|Rest],CL):-
4     cart(AL,BL,C),
5     extend_cart_tuple([C|Rest],CL).
6
7 extend_cart_tuple([C|[]],C).
8 extend_cart_tuple([TupleList,Next|Rest],CL):-
9     help_extend_cart_tuple(TupleList,Next,ETupleList),
10    extend_cart_tuple([ETupleList|Rest],CL).
11
12 help_extend_cart_tuple([],_,[]).
13 help_extend_cart_tuple([T|TupleList],List,ExtTupel):-
14    extend_tuple(T,List,EList),
15    help_extend_cart_tuple(TupleList,List,IList),
16    append(EList,IList,ExtTupel).
17
18 extend_tuple(_,[],[]).
19 extend_tuple(T,[E|Rest],[ExtT|ERest]) :-
20    append(T,[E],ExtT),
21    extend_tuple(T,Rest,ERest).
```



### 3 Simulationsläufe und Beispiele

In diesem Kapitel gebe ich ein Beispiel aus der Simulation. Die Simulation wurde hierbei mit 30 Ticks durchlaufen. Da es nur zwei Agent gibt, bildet Agent `s1` die Gruppe 1 und Agent `d1` die Gruppe 2. Die Welt enthält folgende Fakten:

```
1 fact(1,_,world,zwischen([p1,1],[p2,1],[p3,1])).
2 fact(1,_,world,auf([p1,1],[g1,2])).
3 fact(1,_,world,auf([p2,1],[g1,2])).
4 fact(1,_,world,auf([p3,1],[g1,2])).
5
6 fact(1,_,world,zwischen([p4,1],[p5,1],[p6,1])).
7 fact(1,_,world,auf([p4,1],[g1,2])).
8 fact(1,_,world,auf([p5,1],[g1,2])).
9 fact(1,_,world,auf([p6,1],[g1,2])).
10
11 fact(1,_,world,zwischen([p7,1],[p8,1],[p9,1])).
12 fact(1,_,world,auf([p7,1],[g2,2])).
13 fact(1,_,world,auf([p8,1],[g2,2])).
14 fact(1,_,world,auf([p9,1],[g2,2])).
```

Von diesen sind den Agenten zum Simulationstart diese bekannt:

```
1 bel([1,s1,0.1,fact(1,1,auf([p1,1],[g1,2]))]).
2 bel([1,s1,0.1,fact(1,1,auf([p2,1],[g1,2]))]).
3 bel([1,s1,0.1,fact(1,1,auf([p3,1],[g1,2]))]).
4 bel([1,s1,0.1,fact(1,1,zwischen([p1,1],[p2,1],[p3,1]))]).
5 bel([1,s1,0.1,fact(1,1,hypothesis(h1))]).
```

Und im Fall von `d1`:

```
1 bel([1,d1,0.1,fact(1,1,auf([p1,1],[g1,2]))]).
```

### 3 Simulationsläufe und Beispiele

```
2 bel([1,d1,0.1,fact(1,1,auf([p2,1],[g1,2]))]).
3 bel([1,d1,0.1,fact(1,1,auf([p3,1],[g1,2]))]).
4 bel([1,d1,0.1,fact(1,1,zwischen([p1,1],[p2,1],[p3,1]))]).
5 bel([1,d1,0.1,fact(1,1,hypothesis(h1))]).
```

Wie zu erwarten ist, ist die Überzeugung zu entsprechenden Fakten nach dem Durchlauf der 30 Ticks bereits deutlich stärker:

```
1 agents:bel([_, d1, 0.4, fact(1, 25, auf([p1, 1], [g1, 2]))]).
2 agents:bel([_, d1, 0.4, fact(1, 26, auf([p5, 1], [g1, 2]))]).
3 agents:bel([_, d1, 0.5, fact(1, 26, auf([p4, 1], [g1, 2]))]).
4 agents:bel([_, d1, 0.3, fact(1, 27, auf([p6, 1], [g1, 2]))]).
5 agents:bel([_, d1, 0.5, fact(1, 28, zwischen([p1, 1], [p2, 1], [p3, 1]))
  ]).
```

Der Agent hatte nun die Möglichkeit einen weiteren Satz (Zeile 1) in seine Datenbank aufzunehmen. Interessant ist hier, dass der Fakt aus Zeile 4 trotz des, in der Kombination, guten Passungsverhältnisses einen im Vergleich zu den anderen Fakten schlechten Wert hat.

# 4 Das Prologprogramm

## 4.1 simulation.pl

Das Modul *simulation.pl* bietet die grundlegenden Funktionen der Simulationen. Beispielsweise die Steuerungsmechanismen oder die in Kapitel zwei erläuterten Schleifen. Für den Start der Anwendung, wird das Prädikat `start.` aufgerufen. Als erstes werden die verwendeten Module geladen:

```
1 :- style_check(-singleton).           % Ignoriert 'Singleton Variablen'
2 :-use_module(['agents.pl',           % Definition der Agenten
3             'actions.pl',           % Aktionen und Regeln
4             'function.pl',           % Erweiterte Funktionen
5             'interface.pl',           % Bildschirmausgabe
6             'config.pl',             % Die Konfigurationsdatei
7             'fit.pl',                 % Passung
8             'output.pl',             % Datenausgabe
9             'urban_cart.pl',         % "Urban Regel" üfr das kartesische
           Produkt
10            'world1.pl'               % Konfigurationen üfr die Welten
11 ]).
```

Der Fakt `sim_time/2` hält die die globale Simulationszeit fest. Da sich diese bei jedem durchlauf ändert, wird der Fakt als dynamisch gekennzeichnet und ist somit veränderbar.

```
12 :-dynamic(sim_time/2).
```

Der Fakt `sim_time/0` setzt die Simulation in Gang. Zu Beginn der Simulation wird die Anzahl der Simulationsläufe mit `number_of_runs/1` abgefragt. Darauf wird die Anzahl der zu durchlaufenden Wiederholungen an die zentrale Zeitschleife `run/2` übergeben. Die Funktion `time/1` wird von SWI-Prolog zur Verfügung gestellt und gibt beim Pro-

## 4 Das Prologprogramm

grammende zusätzliche Details zur Laufzeit und den verwendeten CPU Ressourcen der Anwendung aus.

```
13 start:-  
14     number_of_runs(Runs),  
15     time(run(1,Runs)).           % Startet das Programm
```

Für die zentrale Schleife wird zunächst die Endbedingung für definiert.

```
16 run(Ende,Ende):-              % Endbedingung Ticks im Durchlauf  
17     nl,  
18     writeln('E_N_D_U_S_I_M').
```

Zu jedem Simulationslauf wird der Zähler zurückgesetzt, ein Simulationslauf durch `tick/2` durchlaufen und die Ergebnisse mittels `output_result/1` ausgegeben. Darauf werden die Parameter für die Welt und die Agenten aktualisiert und der nächste Durchlauf vorbereitet.

```
19 run(I,Ende) :-                % Schleife üfr alle Ticks pro  
    Druchlauf  
20     I < Ende,  
21     nl,  
22     write('R_U_N: '),  
23     writeln(I),  
24     retractall(sim_time(_, _)), % Entfernt alte Zeit  
25     assert(sim_time(I,t)),      % Setzt globale Zeit  
26     number_of_ticks_per_run(Max_Ticks),  
27     tick(1,Max_Ticks),         % üFhrt Simulation zu Tick I aus  
28     output_result(I),  
29     %retractall(bel(X)),  
30     %retractall(fact(X)),  
31     %retractall(fact(X,Y,Z)),  
32     consult('agents.pl'),  
33     consult('world1.pl'),  
34     I1 is I + 1,              % äNchster Tick wird vorbereitet  
35     run(I1,Ende).            % Rekursiver Schleifenaufruf
```

Die Schleife `tick/2` durchläuft alle Zeitpunkte für einen Simulationslauf. Durch `tick/1`

## 4 Das Prologprogramm

werden dann die einzelnen Simulationsläufe gesteuert.

```
36 tick(Ende,Ende):-                % Endbedingung Ticks im Durchlauf
37     nl,
38     sim_time(Run,Time),
39     write('E_N_D_R_U_N'),
40     writeln(Run).
41
42 tick(I,Ende) :-                  % Schleife üfr alle Ticks pro
    Durchlauf
43     I < Ende,
44     nl,
45     sim_time(Run,Time),
46     write('T_I_C_K:_'),
47     write(I),
48     write('R_U_N:_'),
49     write(Run),
50     retractall(sim_time(_,_)),    % Entfernt alte Zeit
51     assert(sim_time(Run,I)),      % Setzt globale Zeit
52     tick(I),                      % üFhrt Simulation zu Tick I aus
53     output(Run,Time),
54     I1 is I + 1,                  % äNchster Tick wird vorbereitet
55     tick(I1,Ende).                % Rekursiver Schleifenaufruf
```

Innerhalb eines jeden Zeitdurchlaufs werden in `time/1` die anstehenden Aktionen der Agenten durch `actions/1` in gang gesetzt und darauf die veralteten Überzeugungen mit Hilfe von `retract_old_bel/1` aus dem Speicher entfernt.

```
56 tick(Tick):-
57 % rules(Tick),
58     actions(Tick),
59     writeln('*_*_ret_old_bel_*_*_*'),
60     retract_old_bel(Tick).
61
62 output_action_msg(Tick):-
63     write('|_Aktionen_im_TICK:_'),
64     write(Tick),
65     writeln('|_').
```

Das Ausführen von Aktionen wird in zwei Schritte aufgeteilt:

1. Aktionen werden mittels `apply_actions/1` angewendet.
2. Neue Aktionen werden durch `generate_new_actions/1` vorbereitet.

```

65 actions(Tick):-
66     output_action_msg(Tick),
67     writeln('*_*_*_*Apply_*_*_*_*'),
68     apply_actions(Tick),
69     writeln('*_*_*_*gen_act_*_*_*_*'),
70     generate_new_actions(Tick).

71 output_apply_actions_list_msg(Actions):-
72     write(' | In folgende Fakten wird nach Aktionen gesucht: '),
73     write(Actions),
74     writeln(' | ').
75
76 output_apply_action_msg(Action):-
77     write(' | Es wird '),
78     write(Action),
79     writeln(' | üausgefñhrt. | ').

```

Die ersten inhaltlich interessanten Punkte sind in `apply_actions/1` zu finden. Zunächst werden alle in diesem Tick auszuführenden Aktionen gesammelt.

```

80 apply_actions(Tick):-
81     findall(Action, fact(Run, Tick, Action), Actions),
82     output_apply_actions_list_msg(Actions),
83     ( member(Action, Actions),
84       Action=..[action, Act, Agent1, Agent2],
85       output_apply_action_msg(Action),
86       call(Action),
87       fail; true
88     ).

```

Um die alten Überzeugungen aus der Datenbasis zu entfernen, werden zunächst alle Überzeugungen gesammelt und eine Tupel bestehend aus den Überzeugungen  $\times$  Überzeugungen gebildet. Nun wird verglichen, ob es neuere Überzeugungen zu bestimmten

Fakten gibt und darauf werden die veralteten Fakten aus dem Speicher entfernt.

```

89 % Sucht und entfernt ü"berholte" und doppelte Fakten
90 retract_old_bel(Akt_Tick):-
91     findall(Agent,bel([_,Agent,L,F]),Agent_List),
92     (member(Singel_Agent,Agent_List),
93     findall([Tick,Fact],bel([_,Single_Agent,_,fact(Run,Tick,Fact)]),FList)
94     ,
95     cart(FList,FList,Cart),
96     (member(Tupel,Cart),
97     Tupel=[[Tick1,Fact],[Tick2,Fact]],
98     ( ( Tick1<Tick2 ,retract(bel([_,Single_Agent,_,fact(Run,Tick1,Fact)]))
99     );
100     ( Tick2<Tick1 ,retract(bel([_,Single_Agent,_,fact(Run,Tick2,Fact)]))
101     )
102     ),fail>true),fail>true).
103
104 output_status_retract:-
105     write('Suche_und_entferne_alte_bel_momentane_äLnge:'),
106     findall(Bel,bel(Bel),Bel_List),
107     length(Bel_List,L1),
108     writeln(L1).

```

Nachdem Handlungen durchgeführt wurden, müssen neue Handlungen zugeordnet werden. Dieser Vorgang wird durch `generate_new_action/1` in Gang gesetzt. Das geschieht an dieser Stelle zufällig. Natürlich lassen sich hier auch komplexere Regeln angeben. Es wird jedoch beachtet, dass die Fakten auch zu dem Agententypen passen. Ein Student hält also beispielsweise keine Vorlesung.

```

106 generate_new_actions(Tick):-
107     findall(Agents,fact(R,T,agent_type(AT,Agents)),Agent_Set),
108     ( member(Agent,Agent_Set),
109     fact(Run,_,agent_type(Type,Agent)),
110     random_action(Type,Action,Agent,Agent2),
111     Adressed_time is Tick +1,
112     assert(fact(Run,Adressed_time,action(Action,Agent,Agent2))),
113     fail>true

```

```
114 ).
```

Damit die Aktionen auch zufällig gewählt werden können, wird durch `random_action/4` eine zufällige Aktion aus allen möglichen Aktionen bestimmt. Da diese Liste zu jedem Aufruf neu generiert wird, ist es auch möglich, neue Aktionen zur Laufzeit zu generieren oder zu entfernen.

```
115 random_action(Type,Action,Agent,Agent2):-
116     action_list(Type,AL),
117     random_element(AL,Action),
118     Term=.. [Action,Agent,Agent2],
119     call(Term).
```

Zum Schluss folgen noch einige Hilfsprädikate für den Umgang mit den diversen Listen.

```
120 loop_remove_element(E,E,L,NL).
121 loop_remove_element(C,E,List,New_list):-
122     C<E,
123     remove_random_element(List,New_list),
124     Ci is C+1,
125     loop_remove_element(Ci,E,New_list,X).
```

```
126 remove_random_element(List,NList):-
127     random_element(List,Element),
128
129     select(Element,List,NList).
```

```
130 random_element([],Element).
131 random_element(List,Element):-
132     length(List,Length),
133     Length>0,
134     Random is random(Length),
135     Element_Number is Random + 1, % random ist 0=< rnd <n !!
136     nth1(Element_Number,List,Element).
```

```
137 % Kann je nach Bedarf eingebunden werden :
138
139 % retract_old_actions(Tick):-
```



```

140 % writeln('Suche alte Fakten:'),
141 % fact(Run,T,action(Act,A1,A2)),
142 % T<Tick,
143 % PTerm=.. [fact,Run,T,action(Act,A1,A2)],
144 % write(' Entferne : '),
145 % writeln(PTerm),
146 % retract(fact(Run,T,action(Act,A1,A2))).

```

## 4.2 actions.pl

In dem Modul actions.pl werden die möglichen Aktionen definiert. Eine genauere Beschreibung der Funktionsweise ist im Kapitel 2 gegeben.

```

1 :-module(actions,[forschungsarbeit/2,haelt_vorlesung/2,
2     getestet_daten/2,action/3,
3     action_list/2,fuehrt_messung_durch/2]).

```

Zum erstellen von Aktionen sind im Prinzip drei Schritte notwendig:

1. Die neueAktion muss in die in `action_list` des jeweiligen Agententypen eingetragen werden.
2. Die Regeln der Aktion als `action(name_der_aktion)` definieren werden.
3. Unter dem Prädikat `name_der_aktion(Agenten,Agenten)` werden die Agenten selektiert, auf die sich die Aktion bezieht.

```

4 % Listen der zu verwendenden Aktionen üfr die Agententypen
5 action_list(2,[haelt_vorlesung, getestet_daten, fuehrt_messung_durch]).
6 action_list(1,[forschungsarbeit, fuehrt_messung_durch]).

7 action(haelt_vorlesung,Dozent,Audience_list):-
8     write('Handlung:␣Vorlesung␣von␣'),writeln(Dozent),
9     bel([Order_Nr,Dozent,Strength,D_Fact]),
10    findall(Facts,bel([Order_Nr,Dozent,Strength,Facts]),List_of_facts),
11    random_element(List_of_facts,Fact),
12    sim_time(Run,Tick),
13    ( member(Agent,Audience_list),

```

## 4 Das Prologprogramm

```
14 ( %is_known(Order_Nr,Agent,Strength,Fact,Tick) ;
15     is_known(Fact,Agent),!;
16     is_new(Order_Nr,Agent,Strength,Fact,Tick),! ) ),fail>true.
```

Die Prädikate `is_known/5` und `is_new/5` stellen fest, ob Fakten bereits bekannt oder entsprechend neu sind.

```
17 is_known(Fact,Agent):-
18     sim_time(Run,Time),
19     Fact=..[fact,Run,T,Data],
20     findall([Tick,S],bel([ON,Agent,S,fact(Run,Tick,Data)]),List),
21     List\=[],
22     sort(List,SList), % Setzt den letzten Eintrag an das Ende der Liste
23     length(SList,SList_Length), % Sucht das Ende
24     nth1(SList_Length,SList,LList), %Selektiert das letzte Element
25     LList=[LTick,Str],
26     writeln(Str),
27     writeln('DEBUG'),
28     NewStrength is Str + 0.2,
29     NTime is Time+1,
30     ( member(E,List),
31         retract(bel([_,Agent,_,fact(_,_,Data)])),
32         fail>true ),
33     assert(agents:bel([_,Agent,NewStrength,fact(Run,NTime,Data)])).
34
35 output_double_fact(Fact,Agent,N_Fact):-
36     write('|_Doppelter_Fakt_:_''),
37     write(Fact),
38     write('_Agent:_''),
39     write(Agent),
40     write('._Wird_zu_Fakt_'),
41     write(N_Fact),
42     writeln('_|').
43
44 output_new_fact(Fact,Agent,N_Fact):-
45     write('neuer_Fakt_üfr_Agent_'),
```

## 4 Das Prologprogramm

```
46 write(Agent), writeln(N_Fact).

47 is_new(Order_Nr,Agent,Strength,Fact,Tick):-
48     not(bel([Order_Nr,Agent,Strength,Fact])),
49     not(is_known(Fact,Agent)),
50     X is random(5), Multi is (X+5)/10,
51     New_Strength is Strength * Multi,
52     Fact=.. [T,Run,Time,Data],
53     N_Time is Tick+1,
54     N_Fact=.. [T,Run,N_Time,Data],
55     not(agents:bel([ON,Agent,Str_d,N_Fact])),
56     output_new_fact(Fact,Agent,N_Fact),
57     assert(agents:bel([Order_Nr,Agent,New_Strength,N_Fact])).
58 % Rating des Dozenten übergeben
59
60 % Vom Student zum Dozent
61 action(forschungsarbeit,Student,[Dozent]):-
62     write('Handlung:␣Froschungsarbeit␣von␣'),writeln(Student),
63     findall(Facts,bel([Order_Nr,Student,Strength,Facts]),List_of_facts),
64     random_element(List_of_facts,Fact),
65     bel([Order_Nr,Student,Strength,Fact]),
66     sim_time(Run,Tick),
67     X is random(5), Multi is (X+5)/10,
68     New_Strength is Strength * Multi,
69     Fact=.. [T,Run,Time,Data],
70     N_Time is Tick+1,
71     N_Fact=.. [T,Run,N_Time,Data],
72     ( not(agents:bel([Order_Nr,Dozent,New_Strength,N_Fact])),
73       not(agents:bel([Order_Nr,Dozent,New_Strength,Fact])),
74       assert(agents:bel([Order_Nr,Dozent,New_Strength,N_Fact]))).
75
76 % von Dozent zu allen
77 action(testet_daten,Dozent,Single_Agent_List):-
78     sim_time(Run,Tick),
79     member(Agent,Single_Agent_List),
80     write('Handlung:␣Teste␣Daten␣von␣Agent␣'),writeln(Agent),
```

## 4 Das Prologprogramm

```
81 % findall([Strength,Hypothesis],bel([Order_Nr,Dozent,Strength,fact(_,_ ,
    hypothesis(Hypothesis))]),List_of_Hyp),
82 % writeln(List_of_Hyp),
83 % length(List_of_Hyp,Length_LOH),
84 % sort(List_of_Hyp,Sorted_LOH),
85 % nth1(Length_LOH,Sorted_LOH,Selected_Hyp),
86 bel([ON,Agent,V,fact(Run,_,hypothesis(Hyp))]),
87 write('Hypothese␣:␣'),writeln(Hyp),
88 % Selected_Hyp=[Strength,Hyp],
89 findall(bel([0,Agent,S,F]),bel([0,Agent,S,F]),Data),
90 Test_Data=..[Hyp,Data,Value],
91 not(Test_Data=[]) ,
92 call(Test_Data),!,
93 writeln(Value),
94
95 member(SData,Data),
96 ( SData=..[bel,BC],
97   BC=[ON,Agent,S,Fact],
98
99   ( ( S < Value , NV is S / Value ); % Rechnet neue äGlaubensstrke aus
100   ( S > Value, NV is Value / S); % der Passung
101   ( S = Value, NV is Value)
102   ),
103
104   Fact=[Run,Time,C],
105   NT=Time+1,          % Datiert den neuen Fakt
106   NFact=[Run,NT,C],
107
108   NBC=[ON,Dozent,NV,NFact],
109   NSData=..[bel,NBC],
110   %retract(SData),
111   retract(bel(_,Dozent,_,fact(Run,_,C))),
112   assert(agents:NSData) % Aktualisierter Glaubenssatz
113
114 ),fail,true.
```

```

115
116
117 action(fuehrt_messung_durch,Agent,_):-
118     sim_time(Run,Tick),
119     findall(P,fact(Run,Tick,world,P),List),
120     random_element(List,Data),
121     Fact=..[fact,Run,Tick,Data],
122     N_Time is Tick+1,
123     N_Fact=..[fact,Run,N_Time,Data],
124
125     ( not(bel([Order_Nr,Agent,S,N_Fact])),
126       not(bel([Order_Nr,Agent,S,Fact])),
127       assert(agents:bel([Order_Nr,Agent,0.5,N_Fact]))).
128
129
130
131 % Aktionen
132 fuehrt_messung_durch(_,_).
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148

```

```

testet_daten(Dozent,[Agent]):-
    findall(Agents,fact(Run,Tick,agent_type(AT,Agents)),As),
    random_element(As,Agent).

forschungsarbeit(Student,[Dozent]):-
    findall(Agents,fact(Run,Tick,agent_type(2,Agents)),As),
    random_element(As,Dozent).

haelt_vorlesung(Dozent,List_of_all_students):-
    findall(Agents,fact(Run,Tick,agent_type(1,Agents)),
            List_of_all_students).

% Alternative Definition
% is_known(Order_Nr,Agent,Strength,Fact,Tick):-
%     Fact=..[T,Run,Time,Data],
%     sim_time(Run,Tick),

```

```

149 %   N_Time is Tick+1,
150 %   N_Fact=.. [T,Run,N_Time,Data] ,
151 %
152 %   bel([_,Agent,Str_d,N_Fact]),
153 %
154 %
155 %   retract(bel([Order_Nr,Agent,Str_d,N_Fact])),
156 %   X is random(5), Multi is (X+5)/10,
157 %   Str is Str_d + 0.2,
158 %   New_Strength is Str * Multi,
159 %   output_double_fact(Fact,Agent,N_Fact),
160 %   retract(bel([Order_Nr,Agent,Strength,Fact])),
161 %   assert(bel([Order_Nr,Agent,New_Strength,N_Fact])).

```

### 4.3 agents.pl

Das Modul `agents.pl` enthält die Konfigurationen für die Agenten. An dieser Stelle können auch neue Agenten in die Grundkonfiguration eingefügt werden.

```

1 :-module(agents, [bel/1,agent_type/2,fact/3]).
2 :-dynamic([bel/1,agent_type/2,fact/3]).

```

Der Agent `s1` vom Typ `Student` ist zu Beginn mit folgenden Fakten folgend definiert:

```

3 bel([1,s1,0.1,fact(1,1,auf([p1,1],[g1,2]))]).
4 bel([1,s1,0.1,fact(1,1,auf([p2,1],[g1,2]))]).
5 bel([1,s1,0.1,fact(1,1,auf([p3,1],[g1,2]))]).
6 bel([1,s1,0.1,fact(1,1,zwischen([p1,1],[p2,1],[p3,1]))]).
7 bel([1,s1,0.1,fact(1,1,hypothesis(h1))]).
8 \begin{lstlisting}[style=nummern,name=source-agents.pl]

```

Nun muss dieser Agent noch eine Gruppe zugeordnet werden. In diesem Fall der Gruppe `1`, den Studenten.

```

9 fact(1,1,agent_type(1,s1)).

```

Zu Beginn der Simulation wird dieser Agent die Handlung `testet_daten` durchführen.

```
10 fact(1,1,action(testet_daten,s1,[d1])).
```

Als Dozent wird der Agent *d1* auf gleiche Weise definiert.

```
11 bel([1,d1,0.1,fact(1,1,auf([p1,1],[g1,2]))]).
12 bel([1,d1,0.1,fact(1,1,auf([p2,1],[g1,2]))]).
13 bel([1,d1,0.1,fact(1,1,auf([p3,1],[g1,2]))]).
14 bel([1,d1,0.1,fact(1,1,zwischen([p1,1],[p2,1],[p3,1]))]).
15 bel([1,d1,0.1,fact(1,1,hypothesis(h1))]).
16 \begin{lstlisting}[style=nummern,name=source-agents.pl]
```

Entsprechend wird dieser in die Gruppe *d2*, mit der Handlung *haelt\_vorlesung* eingefügt.

```
17 fact(1,1,agent_type(2,d1)).
18 fact(1,1,action(haelt_vorlesung,d1,[s1])).
```

Hier folgen noch weitere Beispiele:

```
19 %
20 % bel([1,s2,0.5,fact(1,1,p([x12,1]))]).
21 % bel([1,s2,0.1,fact(1,1,q([x3,1],[y34,2]))]).
22 % fact(1,1,agent_type(1,s2)).
23 % fact(1,1,action(forschungsarbeit,s2,[d1])).
24
25 % bel([1,d1,0.5,fact(1,1,p([x21,1]))]).
26 % bel([1,d1,0.7,fact(1,1,q([x21,1],[y32,2]))]).
27 % bel([1,d1,0.7,fact(1,1,hypothesis(h1))]).
28 % bel([1,d1,0.3,fact(1,1,hypothesis(h2))]).
29
30 % fact(1,1,agent_type(2,d1)).
31 % fact(1,1,action(haelt_vorlesung,d1,[s1,s2])).
```

## 4.4 config.pl

Konfigurationen lassen sich in der Konfigurationsdatei `config.pl` anpassen. Dies betrifft etwa die zu verwendenden Verzeichnisse oder die Anzahl der zu durchlaufenden Runden.

## 4 Das Prologprogramm

```
1 :-module(config,[number_of_runs/1,  
2             number_of_ticks_per_run/1,  
3             number_of_runs/1,  
4             output_file_suffix/1]).
```

Es muss das Arbeitsverzeichnis und das Verzeichnis für die Ausgabedateien angegeben werden. Hierbei sollte bei der Angabe auf die jeweiligen Eigenheiten des Betriebssystems geachtet werden. In jedem Fall werden jedoch die Angaben in der Form `Verzeichnis/Verzeichnis/...` verlangt. Konkret bedeutet das, dass etwa auch unter Microsoft Windows die Angabe in folgender Form erfolgen muss: `C:/projekte/simulation`

```
6 dir('D:/magister/aktuell').           % Das Arbeitsverzeichnis  
7 output_dir('D:/magister/aktuell/out'). % Das Verzeichnis üfr die Ausgabe
```

Die weiteren Angaben sollten nicht verändert werden, sofern nicht die Grundkonfiguration vollständig geändert wird:

```
8 visual_file('visual.dot').           % Die Ausgabedatei üfr Dot (optional)  
9 output_file_suffix('-result.pl').     % Die Erweiterung üfr die Ausgabe-  
   Datei  
10 agent_config_file('agents.pl').      % Die Agentenkonfigurationen  
11 actions_config_file('actions.pl').    % Die Aktionen und Regeln  
12 simulation_main_file('simulation.pl'). % Der Simulationskern  
13 \begin{lstlisting}[style=nummern,,name=source-config.pl]  
14  
15  
16 :-dir(Dir),working_directory(X,Dir).  % Öffnet das Arbeitsverzeichnis
```

Durch `number_of_runs/1` und `number_of_ticks_per_run/1` können die Anzahl der Durchläufe und Zeiteinheiten bestimmt werden.

```
17 number_of_runs(2).                   % Die Anzahl der äDurchlufe (Run)  
18 number_of_ticks_per_run(30).         % Die Anzahl der Zeiteinheiten (Tick)
```

Mit `output_file/1` wird die Ausgabedatei vorbereitet:

```
19 output_file(Run):-  
20   telling(Old),                         % Aktueller Stream wird in Old  
   gemerkt
```



```

21  get_time(X),                % Generiert einen Zeitstempel
22  stamp_date_time(X,Date,-3600), % Übersetzt den Zeitstempel
23  output_file_suffix(Suffix), % Suffix aus dem Speicher geladen
24  concat(Run,Suffix,Filename), % Dateiname wird gebildet
25  tell(Filename),            % Öffnet Ausgabedatei im Stream
26  writeln(Date),             % Schreibt den Zeitstempel in die
    Datei
27  told,                      % Beendet Stream
28  tell(Old).                 % Stream wird üzurckgesetzt

```

## 4.5 world1.pl

Dieses Modul definiert die Konfiguration der möglichen Welten. Es können zu jedem Simulationslauf eigene Konfigurationen angegeben werden.

```

1  :-module(world1,[fact/4]).
2  % Run 1
3  fact(1,_,world,zwischen([p1,1],[p2,1],[p3,1])).
4  fact(1,_,world,auf([p1,1],[g1,2])).
5  fact(1,_,world,auf([p2,1],[g1,2])).
6  fact(1,_,world,auf([p3,1],[g1,2])).
7
8  fact(1,_,world,zwischen([p4,1],[p5,1],[p6,1])).
9  fact(1,_,world,auf([p4,1],[g1,2])).
10 fact(1,_,world,auf([p5,1],[g1,2])).
11 fact(1,_,world,auf([p6,1],[g1,2])).
12
13 % fact(1,_,world,zwischen([p7,1],[p8,1],[p9,1])).
14 % fact(1,_,world,auf([p7,1],[g2,2])).
15 % fact(1,_,world,auf([p8,1],[g2,2])).
16 % fact(1,_,world,auf([p9,1],[g2,2])).
17 %
18 % Run 2
19 fact(2,_,world,zwischen([p1,1],[p2,1],[p3,1])).
20 fact(2,_,world,auf([p1,1],[g1,1])).

```

```

21
22
23 \begin{lstlisting}[style=nummern,name=source-world1.pl]
24
25 fact(2,_,world,auf([p2,1],[g1,1])).
26 fact(2,_,world,auf([p3,1],[g1,1])).
27
28 fact(2,_,world,zwischen([p4,1],[p5,1],[p6,1])).
29 fact(2,_,world,auf([p4,1],[g1,1])).
30 fact(2,_,world,auf([p5,1],[g1,1])).
31 fact(2,_,world,auf([p6,1],[g1,1])).
32
33 fact(2,_,world,zwischen([p7,1],[p8,1],[p9,1])).
34 fact(2,_,world,auf([p7,1],[g2,1])).
35 fact(2,_,world,auf([p8,1],[g2,1])).
36 fact(2,_,world,auf([p9,1],[g2,1])).
37
38 fact(3,_,world,zwischen([p7,1],[p8,1],[p9,1])).
39 fact(3,_,world,auf([p7,1],[g2,1])).
40 fact(3,_,world,auf([p8,1],[g2,1])).
41 fact(3,_,world,auf([p9,1],[g2,1])).
42
43 fact(4,_,world,zwischen([p7,1],[p8,1],[p9,1])).
44 fact(4,_,world,auf([p7,1],[g2,1])).
45 fact(4,_,world,auf([p8,1],[g2,1])).
46 fact(4,_,world,auf([p9,1],[g2,1])).

```

## 4.6 function.pl

In diesem Modul befinden sich diverse Hilfsfunktionen.

```

1 :-module(functions,[v_v_add/3,n_v_multi/3,add/3,ges_impuls/3,impuls/3,
   load_agent_facts/1]).
2 :-use_module('agents.pl').

```

## 4 Das Prologprogramm

```
3 %:-dynamic([knowledge/2,belive/2,agent_type/2,publication_level/2,  
    group_member/2])
```

Berechnet die Summe über einen gegebene Länge.

```
4 % sigma(+Startwert,+Länge,-Ausgabe).  
5 sigma(A,A,A).  
6 sigma(A,B,N):-  
7     B>A,  
8     A1 is A+1,  
9     sigma(A1,B,N1),  
10    N is A+N1.
```

Durch `v_v_add/3` werden zwei Listen elementweise miteinander Addiert.

```
11 % v_v_add(+Vektor,+Vektor,-Vektor).  
12 v_v_add([],[],[]).  
13 v_v_add([A|Rest1],[B|Rest2],[Neu|Liste]):-  
14     Neu is A+B,  
15     v_v_add(Rest1,Rest2,Liste).
```

`n_v_multi/3` Multipliziert eine Liste mit einem gegebenen Wert.

```
16 % n_v_multi(+Zahl,+Vektor,-Vektor).  
17 n_v_multi(Zahl,Vektor,Ergebnis):-  
18     n_v_multi(Zahl,Vektor,Temp,Ergebnis).  
19 n_v_multi(Multi,[X|Rest],Temp,Ergebnis):-  
20     Zw is Multi*X,  
21     add(Zw,Temp,Neu_L),!,  
22     n_v_multi(Multi,Rest,Neu_L,Ergebnis).  
23 n_v_multi(Multi,[],Temp,Ergebnis):-Ergebnis = Temp.
```

`add/3` fügt ein Element an eine Liste an.

```
24 % add(+Element,+Alte_Liste,-Neue_Liste).  
25 add(Element,[],[Element]).  
26 add(Element,[Kopf|Rest],[Kopf|Neueliste]):-  
27     add(Element,Rest,Neueliste).
```

Die weiteren Funktionen werden in Kapitel 2 genauer erläutert.

## 4 Das Prologprogramm

```
28 % AUFRUF impuls([p1,p2,p3],Impuls).
29 % impuls(+Liste_von_Partikeln,Impuls).
30 impuls([P|Rest],T,[Ergebnis|Rest2]):-
31     m(P,M),
32     v(P,T,V),
33     n_v_multi(M,V,Ergebnis),
34     impuls(Rest,T,Rest2).
35 impuls([],T,[]).

36 % AUFRUF ges_impuls([[2,2,2],[3,3,3]],E)
37 % ges_impuls(+Liste_der_Impulse,-Gesamt_Impuls).
38 ges_impuls([Imp|Rest],Ergebnis):-
39     ges_impuls(Imp,Rest,Ergebnis).
40 ges_impuls(V1,[V2|Rest],Ergebnis):-
41     v_v_add(V1,V2,E),!,
42     ges_impuls(E,Rest,Ergebnis).
43 ges_impuls(X,[],X).

44 load_agent_facts(Agent):-
45     agent(Agent,Facts),
46     member(Fact,Facts),
47     assert(Fact),
48     fail;true.

49 group_knowledge(Agent,Group,List):-
50     group_member(Group,Agent),
51     findall(Knowledge,single_knowledge(Agent,Group,Knowledge),List).

52 single_knowledge(Agent,Group,Knowledge):-
53     functions:group_member(Group,Agent),
54     findall(Agents,group_member(Group,Agents),Agent_List),
55     member(Single_Agent,Agent_List),
56     collecting_knowledge(Single_Agent,Knowledge).

57 collecting_knowledge(Single_Agent,Knowledge):-
58     knowledge(Single_Agent,Single_Knowledge),
59     findall(Single_Knowledge,knowledge(Single_Agent,Single_Knowledge),
60         Knowledge).
```

## 4.7 fit.pl

Das Modul `fit.pl` enthält die Hypothesen und die Passung.

## 4.8 urban\_cart.pl

Dieses Modul wird für die Berechnung des kartesischen Produkts benötigt und wurde von Josef Urban unter <http://www.lrz-muenchen.de/~W.Balzer/urban-rule.pdf> zur Verfügung gestellt.

```

1  % Dieses Modul wurde von Josef Urban entwickelt und ist unter folgender
2  % Adresse zu finden:
3  % >> http://www.lrz-muenchen.de/~W.Balzer/urban-rule.pdf << (April 2010)
4  :-module(cart, [cart/3,n_cart/2]).
5  cart([],_,[]) :-!.
6  cart(_,[],[]) :-!.
7  cart([A|ARest],BL,CL):-
8      create_tuple(A,BL,TList),
9      cart(ARest,BL,CInterim),
10     append(TList,CInterim,CL).
11
12 create_tuple(_,[],[]).
13 create_tuple(A,[B|BRest],[[A,B]|TRest]) :-
14     create_tuple(A,BRest,TRest).
15
16 n_cart(NList,[]) :- member(NList,[]),!.
17 n_cart(NList,undef) :- length(NList,1),!.
18 n_cart([AL,BL|Rest],CL):-
19     cart(AL,BL,C),
20     extend_cart_tuple([C|Rest],CL).
21
22 extend_cart_tuple([C|[]],C).
23 extend_cart_tuple([TupleList,Next|Rest],CL):-
24     help_extend_cart_tuple(TupleList,Next,ETupleList),

```

```

25     extend_cart_tuple([ETupleList|Rest],CL).
26
27 help_extend_cart_tuple([],_,[]).
28 help_extend_cart_tuple([T|TupleList],List,ExtTupel):-
29     extend_tuple(T,List,EList),
30     help_extend_cart_tuple(TupleList,List,IList),
31     append(EList,IList,ExtTupel).
32
33 extend_tuple(_, [], []).
34 extend_tuple(T, [E|Rest], [ExtT|ERest]) :-
35     append(T, [E], ExtT),
36     extend_tuple(T, Rest, ERest).

```

## 4.9 Das Beispiel KSM

Mit dem Modul `modell_KSM.pl` ist der kompletten Quelltext zu dem Beispiel 1.5.1 aus Kapitel 1 gegeben.

```

1 :-use_module(funktionen).
2 :-use_module(daten).
3
4
5 pot_modell(ksm,X):-
6     X=[P,T,V,M],
7     not(P = []),
8     member(T,[t1,t2]),
9     member(Pp,P),
10    v(Pp,T,V),
11    m(Pp,M),
12    true,
13    M>0.
14
15 % AUFRUF modell(ksm,[[p1,p2,p3],T,V,M]). - P4 ist DUMMY
16 modell(ksm,X):-
17     X=[P,T,V,M],

```

#### 4 Das Prologprogramm

```
18 pot_modell(ksm,X),
19
20 % Ges.Impuls zu t1 ...
21 impuls(P,t1,Impuls_t1),
22 ges_impuls(Impuls_t1,Ges_Impuls),
23
24 % ... ist gleich zu Ges.Impuls zu t2
25 impuls(P,t2,Impuls_t2),
26 ges_impuls(Impuls_t2,Ges_Impuls).
```

## 5 Fazit

Die mit dieser Arbeit skizzierte Simulation demonstriert einen Ansatz, die Dynamik von Wissenschaftsgemeinschaften in einer Computersimulation darzustellen. Durch die Verknüpfung der Multiagentensimulation mit der Analyse der Passung auf der Strukturebene wird die Verknüpfung zwischen der sozialen und der strukturellen Ebene deutlich. Doch können inhaltliche Aussagen mit dieser Simulation in diesem Stadium noch nicht getroffen werden. So ist vorgestellte Simulation als Machbarkeitsstudie zu verstehen und nicht als Endprodukt.

Es sind viele sinnvolle Erweiterungen und Präzisierungen denkbar, welche jedoch in dieser Arbeit nicht berücksichtigt werden konnten. Das Überzeugungssystem, welches hier nur sehr eingeschränkt Verwendung findet, bedarf weiterer Ausarbeitung. Eine große Zahl der Zufallsvariablen sollte durch begründete Funktionen und Konstanten ersetzt werden können. Generell ist zu sagen, dass an den Stellen, an denen einfache Konstanten oder Zufallsvariablen zum Einsatz kommen, komplexere Regelwerke die Simulation erweitern können. Eine stärkere Betrachtung von sozialen Strukturen würde interessantere und reichhaltigere Modelle erlauben.

Die Darstellung der Hypothesen ist ebenfalls noch sehr eingeschränkt und kann noch stark erweitert werden. Auf die formalen Hintergründe wird hier kein Bezug genommen. Entsprechend ist auch der Passungsbegriff, welcher im Kern der Simulation liegt, stark vereinfacht und für die Simulation optimiert. Doch gerade dieser würde ebenfalls durch eine stärkere Ausarbeitung, die Simulation deutlich interessanter gestalten.

Das Kernprogramm bietet noch viele Stellen für Optimierungen. So finden viele unnötig komplizierte und ineffiziente Listenoperationen statt, welche durch weitere Optimierungen eliminiert werden können. Auch sind noch einige Fehler in den Operationen vorhanden, welche zwar den Ablauf nicht behindern, jedoch keinen optimalen Durchlauf zulassen. Die Version ist auch stark vom Interpreter abhängig und nicht vollständig ISO kompatibel.



## 5 Fazit

Das Thema der Gruppenbindung von Forschergruppen wird hier gänzlich ausgeklammert. Doch das Verhalten und Agieren von Gruppen intern und miteinander ist ein wichtiger Bestandteil und Teil der Selbstorganisation. Das Verhalten von Intentionen und Gruppenhandlungen im Simulationsbezug wurde bereits in Hofmann, *Dynamik sozialer Praktiken und ihrer zu Grunde liegenden Einstellungen - Modellierung und Simulation* untersucht. Eine Adaption auf die gezeigte Simulation würde ebenfalls eine deutliche Erweiterung darstellen. Bei der Konzeption der Simulation habe ich viel Wert auf ein offenes erweiterbares System gelegt. So können etwa auch in späteren Erweiterungen gegenseitige Beeinflussungen von Messung und simulierter Welt berücksichtigt werden.

Doch als Fazit lässt sich feststellen, dass sich die Verbindung zwischen der Theorieebene und Sozialebene in einer Simulation zum Wissenschaftsprozess darstellen lässt und durchaus Potential für weitere Untersuchungen bietet.

# Literatur

- Balzer, W. *Die Wissenschaft und ihre Methoden*. Freiburg, München: Verlag Karl Alber, 1997.
- Balzer, W. und C. U. Moulines. "Introduction". In: *Structuralist Knowledge Representation: Paradigmatic Examples* 75 (2000), S. 5–18.
- Balzer, W., C. U. Moulines und J. D. Sneed. *An Architectonic for Science : the structuralist program*. Dordrecht: Reidel, 1987.
- Bratko, I. *Prolog - Programmierung für künstliche Intelligenzen*. Bonn: Addison-Wesley, 1986.
- Fagin, R. u. a. *Reasoning about knowledge*. Cambridge, Massachusetts, London: The MIT Press, 1995.
- Hofmann, S. *Dynamik sozialer Praktiken und ihrer zu Grunde liegenden Einstellungen - Modellierung und Simulation*. Wiesbaden: VS Verlag für Sozialwissenschaften, 2009.
- King, R. D. u. a. "The Automation of Science". In: *Science, Vol. 324, no. 5923* (2009), S. 85–89.
- Krohn, W. und G. Küppers. *Die Selbstorganisation der Wissenschaft - Wissenschaftsforschung Report 33 Science Studies*. 1987.
- Hrsg. *Selbstorganisation: Aspekte einer wissenschaftlichen Revolution (Wissenschaftstheorie, Wissenschaft und Philosophie 29)*. Braunschweig / Wiesbaden, 1990.
- Kuhn, T. S. *Die Struktur wissenschaftlicher Revolutionen, zweite revidierte Auflage*. Berlin: Shurekamp, 1976.
- Lakatos, I., A. Musgrave u. a. *Kritik und Erkenntnisfortschritt - Wissenschaftstheorie Wissenschaft und Philosophie Band 9*. Hrsg. von Imre Lakatos und Alan Musgrave. Braunschweig-Wiesbaden: Vieweg, 1974.
- Langley, P. u. a. *Scientific Discovery: Computational Explorations of the Creative Processes*. Cambridge, Massachusetts, London: The MIT Press, 1987.
- Lauth, B. und J. Sareiter. *Wissenschaftliche Erkenntnis*. Paderborn: Mentis, 2005.

## Literatur

- Mason, S. *Die Geschichte der Naturwissenschaften, zweite Auflage*. Stuttgart: Kröner, 1974.
- Masterman, M. "Die Natur eines Paradigmas". In: *I. Lakatis u.a.: Kritik und Erkenntnisfortschritt*. 1965, S. 59–88.
- Querenburg, B. v. *Mengentheoretische Topologie, zweite neubearbeitete und erweiterte Auflage*. Berlin, Heidelberg, New York: Springer, 1979.
- Schmidt und Lipson. "Distilling Free-Form Natural Laws from Experimental Data". In: *Science, Vol. 324, no. 5923* (2009), S. 81–85.
- Stegmüller, W. *Hauptströmungen der Gegenwartsphilosophie - Band II 6., erweiterte Auflage*. Stuttgart: Kröner, 1979.
- *Probleme und Resultate der Wissenschaftstheorie und Analytischen Philosophie Band II - Theorie und Erfahrung*. Berlin: Springer, 1970.
- Urban, J. "Zusammenspiel von Problemösungsstrategien und Theorie-Entwicklung: Ein Ansatz zur Simulation". Magisterarb. Magisterarbeit an der Ludwig-Maximilians-Universität München, 1995.
- Wielemaker, J. und A. Anjewierden. *Programming in XPCE/Prolog*. University of Amsterdam. 2005.