Grundzüge der Computermodellierung

Klaus Manhart

Überarbeitete Fassung aus meiner Dissertation

„KI-Modelle in den Sozialwissenschaften“,
Oldenbourg-Verlag, München, 1995

www.klaus-manhart.de
mail@klaus-manhart.de

München, September 2007

1. Einleitung

Computermodelle sind eine echte Teilmenge formaler Modelle, und Computermodellierung ist die Konstruktion und Ausführung von Modellen auf dem Computer. Computermodelle haben zusätzlich zu den Charakteristika formaler mathematischer Modelle die Eigenschaft, dass sie *von einem Rechner interpretiert und ausgeführt* werden können.[1] Das Repräsentationsmedium des Bildbereichs - die formale Sprache - bilden in diesem Fall Programmiersprachen oder eigens entwickelte Modellbeschreibungssprachen (Möhring 1990: 11). Abb. 1 zeigt eine Differentialgleichung als formales Modell und die programmiersprachliche Repräsentation als Computermodell.

Formales Modell (Differentialgleichung):

$$\ddot{x} = -2zw\,\dot{x} - w^2 x + w^2 f$$

mit $\dot{x}(0) = x(0) = 0$

Computermodell (FORTRAN-ähnliche Sprache):

```
X2D  = -2.0*ZETA*OMEGA*X1D-X*OMEGA**2+OMEGA**2*F
X1D  = INTGRL(0.0, X2D)
X    = INTGRL(0.0, X1D)
```

Abb. 1: Formales Modell und Computermodell (Kheir, nach Möhring 1990: 12)

Mit *Computersimulation* ist meist ein dynamischer Aspekt gemeint. Sie betrifft Modelle, in denen die zeitliche Entwicklung eines Systems repräsentiert ist. Troitzsch (1990: 178) fasst den Simulationsbegriff weiter, nämlich als „Experimentieren mit Modellen“, Kreutzer (1986: 5) als „experimental technique for exploring 'possible worlds' through computational processes“ und für Möhring (1990: 13) ist Simulation konkret „Beobachtung von Modellverhalten über einen längeren Zeitraum“.[2]

In Erweiterung des Schemas von Abb. 1 lassen sich die Relationen zwischen empirischem System, Modell und Computerprogramm grafisch in der folgenden Abbildung verdeutlichen.

---

[1] Computermodelle haben im Gegensatz zu mathematischen Modellen aber auch die Eigenschaft, dass sie Teile enthalten, die für das eigentliche Modell völlig irrelevant sind (vgl. Kap. 3.1.2).

[2] Der Simulationsbegriff von Troitzsch, Kreutzer und Möhring ist insofern „weiter“, als er Simulation nicht einengt auf dynamische Modelle. Zwar haben die meisten Computermodelle dynamischen Charakter, aber es gibt auch nicht-dynamische, statische Modelle (vgl. Tab. 2.1, S.35).

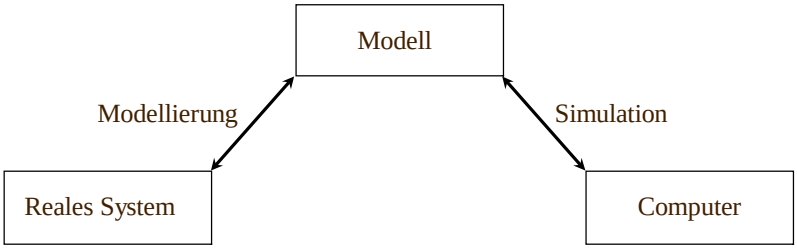

Abb. 2: Modellbildung und Simulation (nach Zeigler 1976: 4)

Dieses Schema ist allerdings insofern kritisch zu betrachten, als es zum einen möglich ist, dass es zu einem Computerprogramm kein verbales oder mathematisches Gegenstück gibt, so dass das Programm unmittelbar das Modell ist. Zum andern lässt das Schema die Relation von Modell und Theorie außer acht und es ist denkbar, dass eine Theorie direkt in einem Programm ausgedrückt wird. Auf diese Aspekte wird weiter unten eingegangen.[3]

2. Elementare Eigenschaften von Computermodellen[4]

Ein grundlegendes Merkmal von Computermodellen - oder allgemeiner: Computerprogrammen - ist, dass sie Eingabedaten entgegennehmen, diese verarbeiten und Ausgabedaten produzieren. Das Programm verändert die eingegebenen Daten gemäß bestimmter Regeln und gibt Daten an der Computerperipherie (Bildschirm, Drucker etc.) aus. Die Programmregeln sind so flexibel, dass sie auf eine ganze Menge von Eingaben angewandt werden können. Wenn wir vorerst das Computermodell M als Black-Box betrachten, dann nimmt M also eine Menge von ***Eingabedaten (Input)*** entgegen und liefert eine Menge von ***Ausgabedaten (Output)***. Wir bezeichnen die Menge der möglichen Modellinputs mit I_M und die Menge der möglichen Outputs mit O_M.[5] Die $i_M \in I_M$ brauchen vorerst nicht näher spezifiziert werden: es können Zahlen (z.B. Vektoren, Matrizen) oder beliebige andere Zeichenketten sein - im Extremfall auch umgangssprachliche Sätze (vgl. Colby 1973, 1975). Die Daten können ferner von realen empirischen Systemen stammen, es kann sich aber auch um künstliche oder hypothetische Daten handeln, wie dies besonders bei sozialwissenschaftlichen Simulationen oft der Fall ist.

I_M kann leer sein, nämlich dann, wenn M keine Eingabewerte verlangt - was allerdings nicht der Regelfall ist. Analog ist es denkbar, aber wenig sinnvoll, dass O_M leer ist, nämlich dann, wenn M keine Outputdaten erzeugt.
Die Anwendung des Modells M auf i_M liefert o_M, so dass wir folgendes ***Input-Output-Schema*** erhalten.

---

[3] Andere Schemata, auf die wir hier nicht weiter eingehen, diskutiert Troitzsch (1990).
[4] Vgl. zu diesem Abschnitt insbesondere Zeigler (1976) und Kobsa (1984).
[5] Wir lassen hier einfachheitshalber den Zeitaspekt außer acht.

Abb. 3: Input-Output-Schema für ein Modell M

Wir nennen das Paar $\langle i_M, o_M \rangle$ ein *Input-Output-Paar* oder *I/O-Paar*. Die Wiederholung dieses Vorgangs mit verschiedenen i_M ergibt eine Menge solcher I/O-Paare, die wir als *I/O-Relation* R_M bezeichnen: $R_M \subseteq I_M \times O_M$.

R_M ist also die Menge aller I/O-Paare. Ist die I/O-Relation rechtseindeutig, d.h. bildet sie einen beliebigen Eingabewert auf genau einen Ausgabewert ab, so sprechen wir von einer *I/O-Funktion*. Dies ist der Fall bei deterministischen Modellen, in denen keine Zufallsereignisse vorkommen. In stochastischen Modellen ist es möglich, dass es für ein bestimmtes i_M unterschiedliche o_M gibt. Wir beschränken uns im folgenden auf deterministische Modelle, die I/O-Funktionen liefern.
In diesem Fall kann das Modell als Funktion f_M von der Menge der Inputdaten in die Menge der Outputdaten betrachtet werden (Claus 1986: 378-379):
f_M: $I_M \rightarrow O_M$

Mit *Modell-* oder *I/O-Verhalten* ist die Umwandlung des Modellinputs in den Modelloutput gemeint, d.h. die Generierung von Outputdaten aus Inputdaten. Das vom Modell abgebildete empirische System S kann nun ebenfalls als Black-Box betrachtet werden, das aus Inputwerten $i_S \in I_S$ Outputwerte $o_S \in O_S$ generiert. Ziel des Modells ist zunächst, diesen Realitätsausschnitt S nachzubilden, so dass das Modellverhalten dem Verhalten des empirischen Systems gleich oder zumindest ähnlich ist. Wir vergleichen folglich das Verhalten des Modells mit dem Verhalten des empirischen Systems, was in Abb. 4 veranschaulicht wird (nach Harbordt 1974: 161).

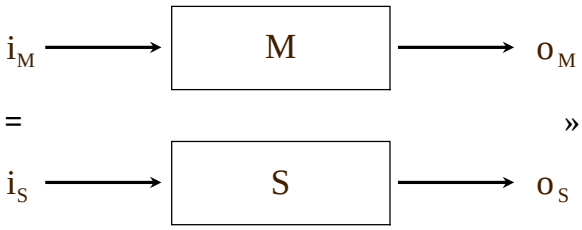

Abb. 4: Vergleich des I/O-Verhaltens von Modell M und empirischem System S.
„≈„ repräsentiert die Ähnlichkeitsrelation.

Bei der Modellkonstruktion strebt man in der Regel danach, dass die Beobachtung eines bestimmten I/O-Paares in S einem gleichen oder zumindest ähnlichen I/O-Paar in M korrespondieren soll. Umgekehrt sollte jedem I/O-Paar in M ein gleiches oder ähnliches Paar in S korrespondieren. Konkreter formuliert lautet letzteres: wir geben in das Modell M genau die Inputdaten, die für S zu einem bestimmten Zeitpunkt gültig sind (z.B. in ein Bevölkerungsmodell demographische Daten von 1960) und sollten von M gleiche oder zumindest ähnliche Werte zu S für einen späteren Zeitpunkt als Outputdaten erhalten (z.B.

die gleiche oder eine ähnliche Geschlechts-/Altersverteilung für 1994). Formal ausgedrückt sollte M idealerweise für alle I/O-Relationen sowohl vollständig als auch korrekt sein. M ist *vollständig*, wenn jede I/O-Relation des empirischen Systems die korrespondierende I/O-Relation in M zur Folge hat. Jede beliebige empirische I/O-Relation zwischen i_S und o_S müsste bei Vollständigkeit des Modells also die analoge I/O-Relation in M liefern, genauer (mit „≈„ als Symbol für die Ähnlichkeitsrelation):

für alle $i_S \in I_S$, $o_S \in O_S$: wenn i_S zu o_S führt, dann gibt es $i_M \in I_M$, $o_M \in O_M$, so dass $i_M = i_S$ und $o_M \approx o_S$.

Umgekehrt heißt M *korrekt*, wenn jede I/O-Relation des Modells, bei dem der Modellinput mit dem Input des empirischen Systems identisch ist, bei Eintreten des Systemoutputs einen ähnlichen Modelloutput liefert (vgl. hierzu auch Kobsa 1984: 53). Es müsste also genauer gelten:

für alle $i_M \in I_M$, $o_M \in O_M$: wenn i_M zu o_M führt und es gibt $i_S \in I_S$, so dass $i_M = i_S$, dann $o_M \approx o_S$.

Vollständigkeit und Korrektheit für *alle* I/O-Relationen sind für die Modellierung zu starke Forderungen. Sie würden eine vollständige Verhaltenskopie des empirischen Systems erzeugen. Die oben gelieferten Definitionen sind deshalb auf den modellierten Ausschnitt des empirischen Systems und die dort betrachteten Objekte und Relationen einzuschränken, so dass die *Vollständigkeits- und Korrektheitsforderung* nur *partiell* für diese Teilmenge gilt.

Vollständigkeit und Korrektheit beziehen sich auf die Adäquatheit, also Gültigkeit oder *Validität* von Modellen. Validität wird i.a. als *notwendig* angesehen, um von Modelleigenschaften auf Eigenschaften des empirischen Systems rückzuschließen. Validität im Sinne von *vollständiger Identität* der I/O-Paare von Modell und empirischem System ist in der Regel eine zu starke Forderung, die Outputs sollten aber zumindest *ähnlich* sein. Das Problem ist dann die Spezifizierung der Ähnlichkeitsrelation und es wird noch verschärft, wenn das Modell Wahrscheinlichkeitsfunktionen verwendet, die zu Zufallsschwankungen im Modellverhalten führen.

Zur Überprüfung der Validität bieten sich für numerische Modelle - d.h. für Modelle, deren Ein- und Ausgaben Zahlen darstellen - statistische Tests an: man lässt z.B. das Modell und das reale System Stichproben von Verhaltensprotokollen erzeugen[6] und vergleicht die entsprechenden I/O-Paare des Modells und des realen Systems mit den üblichen statistischen Verfahren (für einen Überblick über diese Verfahren vgl. Harbordt 1974). Eine andere Vergleichsvariante sind bereichsspezifische Turing-Tests, die vor allem bei nicht-numerischen Modellen verbreitet sind. Vereinfacht gesagt besteht ein Modell M den Turing-Test genau dann, „wenn ein kompetenter Beobachter nicht in der Lage ist, eine

[6] Da das reale System oft nicht manipulierbar ist, muss man meist von diesem ausgehen und die empirisch aufgetretenen Eingabewerte dann zum Modellinput machen.

Verhaltenssequenz des Urbildes von der Verhaltenssequenz des Modells zu unterscheiden, wenn er also Verhaltenssequenzen aus diesen beiden verschiedenen Quellen verwechselt" (Dörner 1984: 347).[7] Der Turing-Test eignet sich aufgrund mehrerer Eigenschaften nur bedingt für Modelltests (z.B. wer bestimmt die Kompetenz des Beobachters, welches Kriterium zieht er für sein Unterscheidbarkeits/Nicht-Unterscheidbarkeits-Urteil heran?). Abelson (1968: 318-320) stellt einen erweiterten Turing-Test vor, und in der Literatur finden sich weitere Varianten dieses Tests. Mehrere dieser Modifikationen wendet z.B. Colby (1975) in einem bekannten psychologischen Modell an, welches paranoides Verhalten simuliert. Das Modell nimmt als Inputs umgangssprachliche Sätze entgegen und produziert solche als Outputs, z.B. (Colby 1975: 78):

Input Interviewer: `HAVE YOU BEEN ACTIVELY TRYING TO AVOID THE UNDERWORLD?`
Output Modell: `NO ONE HAS ANY POWER OVER GANGSTERS.`

Mehrere modifizierte Turing-Tests wurden von dem Modell „bestanden". Bei einem dieser Tests sollten z.B. ausgewählte und angesehene Psychiater Interviewtranskripte, die von dem Modell stammten und solche, die von einem wirklichen paranoid Kranken herrührten, identifizieren. Die Gutachter nahmen zu 51% die richtige und zu 49% die falsche Identifikation vor, was statistisch interpretiert auf dem 95%-Konfidenzintervall einem Zufallsergebnis entspricht. Mit anderen Worten: die Experten konnten nicht unterscheiden, welche Antworten von dem Menschen und welche von der Maschine kamen.

Fragen der Gültigkeit von Modellen stehen nicht im Mittelpunkt dieser Arbeit und sollen hier nicht weiter behandelt werden. Die Gültigkeit spielt zudem bei vielen Computermodellen keine entscheidend große Rolle, da diese oft die Funktion von Ideenfindern und „formalisierten Gedankenexperimenten" haben (Ziegler 1972). „Das Gedankenexperiment dient dazu, die potentielle Erklärungskraft von Theorien zu überprüfen, d.h. festzustellen, ob sich bestimmte Aussagen tatsächlich aus gegebenen Prämissen ableiten lassen. Dagegen kommt ihm keine empirische Beweiskraft zu. Man wird jedoch Zeit und Kosten für theoretisch fruchtlose Untersuchungen sparen, wenn man von vornherein logisch nicht schlüssige 'Erklärungen' aussondert und nicht versucht, sie empirisch 'zu überprüfen'„ (Ziegler 1972: 89). Computersimulationen können deshalb als moderne Form des Gedankenexperiments betrachtet werden, in der Theorien oder Hypothesen konkret und unter kontrollierten Bedingungen „durchgerechnet" werden können, ohne sich zunächst um empirische Fragestellungen oder Validität zu kümmern.

Wir haben das Modell bislang als Black-Box betrachtet im Sinne einer Funktion, die Inputwerte nimmt und Outputwerte liefert:

$f_M(i_M)=o_M$.

---

[7] Der Mathematiker Alan Turing überlegte sich die allgemeine Fassung dieses Tests 1950 im Zusammenhang mit der Frage, ob Maschinen denken können. Grob gesprochen, ist auf diese Fragestellung nach Turing positiv zu reagieren, wenn das Antwortverhalten der Maschine in einem Frage-Antwort-Spiel von dem eines Menschen nicht zu unterscheiden ist. Für nähere Details vgl. Boden (1987) oder Abelson (1968).

Wird das Modell nur nach seinem I/O-Verhalten beurteilt und bildet es lediglich die I/O-Relationen des empirischen Systems ab, so ist das Modell bestenfalls ein *Verhaltensmodell*. Ein Verhaltensmodell ist seinem Urbild allenfalls gleich in der Art und Weise, wie es auf Inputs reagiert. Das gleiche I/O-Verhalten kann aber von einem ganz anderen Modell ebenfalls erzeugt werden, unter Umständen gibt es unendlich viele Modelle $M_1,...,M_n,...$ die sich völlig gleichartig verhalten. Man nennt Modelle, die das gleiche I/O-Verhalten produzieren, *funktional äquivalent* (Abb. 5). Allgemein heißen zwei Prozesse funktional äquivalent, wenn sie im mathematischen Sinn dieselbe Funktion realisieren (Görz 1988: 73).[8]

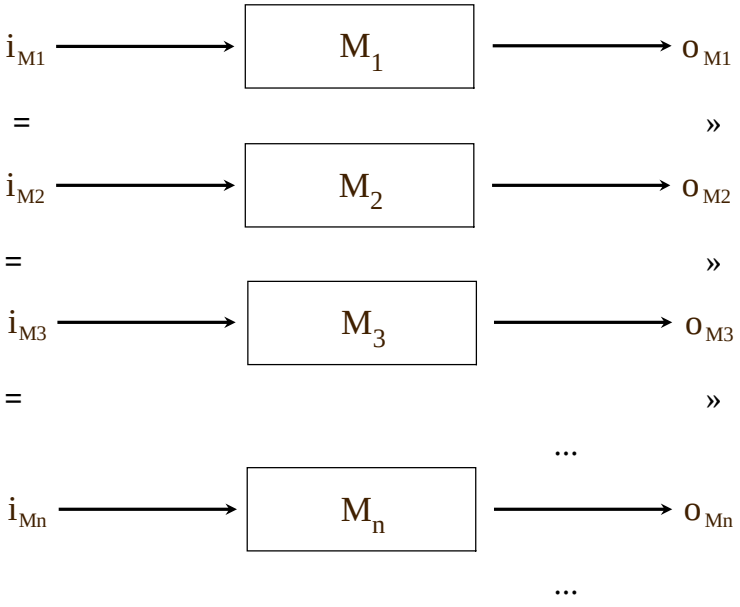

Abb. 5: Funktional äquivalente Modelle $M_1,...,M_n...$

Während ein Verhaltensmodell also seinem Urbild nur gleich ist oder gleich sein sollte im Hinblick auf die Art und Weise, wie es auf äußere Reize reagiert, ist ein *Strukturmodell* seinem Urbild zusätzlich auch gleich hinsichtlich der betrachteten Elemente und Relationen, d.h. es bildet die betrachtete Form und das Gefüge des Urbilds ab. Ein Strukturmodell ist damit immer auch ein Verhaltensmodell, aber nicht umgekehrt (Dörner 1984: 342).

Unter Verwendung der aus der mathematischen Modelltheorie stammenden Begriffe Homomorphie und Isomorphie lässt sich die Modellrelation für Strukturmodelle spezifizieren. Harbordt (1974: 58) führt die beiden Begriffe informell wie folgt ein:

Ein Modell M ist zu einem empirischen System S *isomorph* genau dann, wenn (gdw) gilt:
(1) Jedem Modellelement $m \in M$ ist ein $s \in S$ eindeutig zugeordnet und umgekehrt;

---

[8] Görz (1988: 73) verweist auf ein anschauliches Beispiel von Colby: Alle Mausefallen sind funktional äquivalent. Es gibt zwar verschiedene Mechanismen, um Mäuse zu fangen, doch die Bezeichnung „Mausefalle“ besagt, was ihnen gemeinsam ist: jede hat als Input eine lebendige Maus und als Output eine tote.

(2) Jeder Relation in S ist eine Relation in M eindeutig zugeordnet und umgekehrt;
(3) Einander zugeordnete Relationen enthalten nur einander zugeordnete Elemente.

Ein Modell M ist zu einem empirischen System S *homomorph* gdw gilt:
(1) Jedem Modellelement $m \in M$ ist ein $s \in S$ eindeutig zugeordnet, aber nicht umgekehrt;
(2) Jeder Relation in M ist eine Relation in S eindeutig zugeordnet, aber nicht umgekehrt;
(3) Einander nach (2) zugeordnete Relationen enthalten nur einander nach (1) zugeordnete Elemente.

Die Isomorphie-Relation gilt in ***beide*** Richtungen, Homomorphie nur vom Modell zum Urbild. In einer isomorphen Modellrelation würden sich also alle Elemente und Relationen des Urbilds im Modell vollständig wieder finden und umgekehrt. In einer homomorphen Modellrelation hingegen sind nicht alle Elemente und Relationen des Urbilds im Modell repräsentiert. Harbordt (1974: 59) verweist darauf, dass analog wie uneingeschränkte Vollständigkeit und Korrektheit auch Isomorphie eine ungeeignete Zielvorstellung ist, da das Modell die gleiche Komplexität annehmen würde wie das empirische System. Dies widerspricht dem Vereinfachungscharakter von Modellen. Man begnügt sich deshalb in der Regel mit strukturähnlichen, homomorphen Modellen (Harbordt 1974: 59).

Fasst man die Modellrelation als Homomorphismus auf, kann man andererseits aber nicht mehr von Merkmalen des Urbilds auf Modellmerkmale schließen, womit die Tatsache verloren geht, dass empirische Systeme in Modelle abgebildet werden. Dörner (1984) bezeichnet die Modellrelation in Strukturmodellen deshalb besser als „***partiellen Isomorphismus***“, bei dem die Isomorphierelation nur „bezüglich bestimmter, ausgewählter Merkmale“ gilt. Bezüglich dieser vom Modellierer ausgewählten Merkmale herrscht dann „eine umkehrbar eindeutige Abbildung, bei der alle Relationen erhalten bleiben“ (Tack, nach Dörner 1984: 337).

In einer so definierten Modellrelation steht im übrigen ***jedes*** der Systeme Urbild und Bild in Modellrelation zum anderen. Die Modellrelation ist also symmetrisch, so dass nicht unbedingt festgelegt werden muss, welches als Modell für das andere dient. Die Symmetrie der Modellrelation macht es überflüssig, Bild und Urbild zu identifizieren.

Die Teilmenge des Urbildbereichs, deren Elemente keine Bilder im Modell haben, bezeichnet man als ***Präteritionsklasse*** des Modells. Bei der Modelleisenbahn wäre dies z.B. der Dampfkessel einer Lokomotive. Die Teilmenge des Modells, deren Elemente nicht Bilder von Elementen des empirischen Systems sind, bezeichnet man als ***Abundanzklasse*** des Modells. Der Schleifbügel des Stromabnehmers bei der Modelleisenbahn hat z.B. kein Äquivalent im empirischen System (Troitzsch 1990: 14; Dörner 1984: 338). Abb. 2.6 veranschaulicht diese Beziehung grafisch.

Bei der Frage nach der Gültigkeit von Strukturmodellen kommt zu der Prüfung des I/O-Verhaltens noch die Prüfung auf Strukturgleichheit oder zumindest -ähnlichkeit von Modell und Urbild hinzu. Auf diese zum Teil recht diffizilen und kontrovers diskutierten

Möglichkeiten kann hier nicht eingegangen werden. Einige davon werden besprochen in Dörner (1984) oder Harbordt (1974).

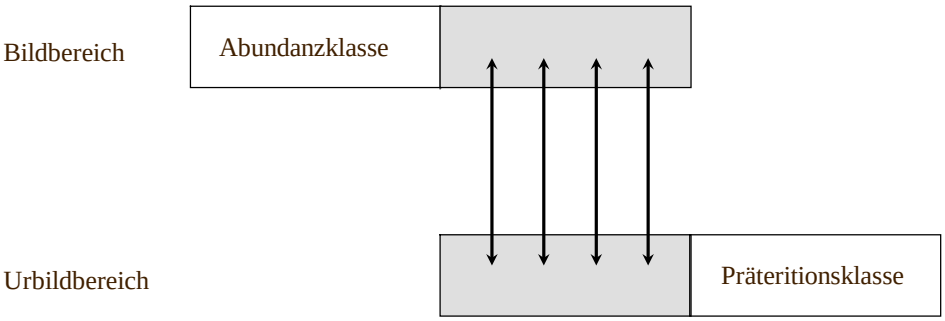

Abb. 6: Partieller Isomorphismus beim Strukturmodell mit Abundanz- und Präteritionsklasse (nach Troitzsch 1990: 14). Im abgedunkelten Bereich liegen die einem partiellen Isomorphismus genügenden Elemente und Relationen.

Damit sind die für unsere Zwecke wichtigsten Grundmerkmale der Computermodellierung angesprochen. Ausführlicher wird dieses Thema behandelt in Harbordt (1974), Kreutzer (1986) und Zeigler (1976). Zeigler (1976) entwickelt eine formale Modellierungs- und Simulationstheorie.

3. Effektive Verfahren und Algorithmen[9]

Das I/O-Verhalten von Computermodellen wird durch den im Computer realisierten Algorithmus bestimmt. Ein *Algorithmus* oder *effektives Verfahren* ist eine mechanische Verarbeitungsvorschrift, die angibt, wie Eingabedaten schrittweise in Ausgabedaten umgewandelt werden.

Algorithmen besitzen drei Eigenschaften (Kleinknecht/Wüst 1976: 57-58):

1. *Determiniertheit*, d.h. die Verarbeitungsvorschrift ist bis in alle Einzelheiten eindeutig festgelegt, und Wiederholung mit gleichen Eingabewerten und Startbedingungen liefert das gleiche Ergebnis;
2. *Allgemeinheit*, d.h. ein Algorithmus dient zur Beantwortung einer ganzen Klasse von Fragestellungen und nicht nur eines speziellen Problems;
3. *Endlichkeit*, d.h. ein Algorithmus ist durch einen endlichen Text beschreibbar und bricht nach endlich vielen Schritten mit einem Ergebnis ab.

Die Idee, ein mechanisches Verfahren oder einen Algorithmus zur Durchführung einer Aufgabe zu haben, existiert schon seit tausenden von Jahren. Eine exakte Bestimmung des Algorithmusbegriffs gelang aber erst Anfang dieses Jahrhunderts im Rahmen mathematischer Grundlagenforschung. Anlass für diese Explikation war das um 1900

[9] Vgl. zu diesem Kapitel insbesondere das ausführliche Kap. 3 „Theorie der Algorithmen“ in Goldschlager/Lister (1990).

aufgeworfene Hilbertsche Entscheidungsproblem.[10] Die Frage von Hilbert bestand darin, ob es einen Algorithmus geben kann, der für eine beliebige Aussage in einem gegebenen formalen System entscheidet, ob die Aussage richtig oder falsch ist. Gäbe es einen solchen Algorithmus, dann ließe sich jedes wohl definierte Problem einfach durch eine Algorithmusausführung lösen. Das Beweisproblem bestand zunächst in der „Operationalisierung" des Algorithmusbegriffs. Einer der Versuche, „Algorithmus" präzise festzulegen, geht auf Alan M. Turing zurück. Turings Idee war es, den intuitiven Algorithmusbegriff mit Hilfe einer Menge von Anweisungen an eine einfache Maschine zu erfassen. Zur Lösung des Entscheidungsproblems konstruierte Turing (1937) ein gedankliches Maschinenmodell - das später als Turing-Maschine bezeichnet wurde - und stellte die folgende Behauptung auf.

TURING-THESE
Jeder Algorithmus im intuitiven Sinn kann auf einer Turing-Maschine ausgeführt werden.[11]

Die Turing-These ist kein Theorem, sondern der Vorschlag einer ***Begriffsexplikation***. Diese Begriffsexplikation von „algorithmisch berechenbar" durch „mit Turing-Maschinen berechenbar" konnte bislang nicht widerlegt werden in dem Sinn, dass es etwas gibt, was intuitiv algorithmisch ist und nicht auf einer Turing-Maschine ausführbar ist. Alle im Verlauf der Geschichte erdachten Algorithmen lassen sich durch die Turing-Maschine erfassen. Mathematiker sind von der Gültigkeit der Turing-These so sehr überzeugt, dass mathematische Beweise, die sich auf sie beziehen, akzeptiert werden (Claus 1986: 97).

In dem so festgelegten Sinn - dass es für jeden Algorithmus im intuitiven Sinn eine Turing-Maschine gibt - konnte Turing zeigen, dass kein Algorithmus von der Art existiert, nach dem Hilbert strebte. Church, Kleene, Post und Gödel kamen in den dreißiger Jahren zu dem gleichen Ergebnis, wobei jeder „Algorithmus" in unterschiedlicher Weise definierte. Diese unterschiedlichen Definitionen werden als gleichwertig aufgefasst (Church-Turing-These), und bis heute gibt es kein einleuchtendes Gegenbeispiel dafür, dass alle vernünftigen Algorithmusdefinitionen nicht gleichwertig und gleichbedeutend seien.

Turing-Maschinen sind theoretisch bedeutsam, haben praktisch aber keine Relevanz, da heutige Computer in der Regel als von-Neumann-Maschinen gebaut werden. Das charakteristische Prinzip des von-Neumann-Rechners ist, dass zur Lösung eines Problems von außen ein Programm eingegeben werden muss, das im selben Speicher liegt wie die Daten. Berechnungen werden schrittweise, eine nach der anderen, durchgeführt. Den Zusammenhang zwischen Turing- und von-Neumann-Maschinen kann man sich in etwa wie folgt vorstellen. Ein von-Neumann-Computer, der mit einem festen Programm

---

[10] Das Hilbertsche Entscheidungsproblem war Teil eines Programms der Neubegründung der Mathematik in Form kalkülisierter Axiomensysteme. Die axiomatische Fundierung der Mathematik wurde infolge der Entdeckung verschiedener Antinomien als notwendig erachtet. Einen guten Überblick über diese Thematik gibt Krämer (1988).

[11] Die Turing-Maschine kann hier nicht dargestellt werden. Eine genaue Beschreibung findet sich z.B. in Reischuk (1991) oder in Krämer (1988). Eine populärere Darstellung mit Hintergründen und Auswirkungen dieses abstrakten Modells gibt Hopcroft (1984).

arbeitet, ist durch eine Turing-Maschine beschreibbar. Umgekehrt gibt es zu jeder Turing-Maschine ein Computerprogramm, das diese simuliert. Turing-Maschinen und Computer mit potentiell unendlichem Speicher sind also bezüglich der durch sie erzeugten Funktionenklasse äquivalent (Claus 1986: 504). Dies bedeutet, dass ein Computer alle Turing-berechenbaren Probleme behandeln kann, so dass ***ein Computer praktisch alle realen Prozesse ausführen kann, die intuitiv als effektive Verfahren oder algorithmisch bezeichnet werden können.*** Umgekehrt ist alles, was auf dem Computer realisierbar ist, algorithmisch. Darüber hinaus kann jeder Algorithmus, der auf einem bestimmten Computer realisiert wird, auf jedem anderen Computer realisiert werden: jeder Computer ist allen anderen in dem Sinn gleichgestellt, dass alle im Prinzip die gleichen Aufgaben ausführen können (Universalität).

Die Bearbeitungsmöglichkeit algorithmisch erschließbarer Probleme durch Computer hat Konsequenzen für das Modellierungspotential des Computers. „Da der Mensch, die Natur und selbst die Gesellschaft mit Verfahren arbeiten, die zweifellos in der einen oder anderen Hinsicht 'effektiv' sind, folgt daraus, dass ein Computer zumindest den Menschen, die Natur und die Gesellschaft in allen verfahrensmäßigen Aspekten imitieren kann. Wir müssten also immer imstande sein, wenn wir ein Phänomen insofern zu verstehen glauben, als wir dessen Verhaltensregeln kennen, unser Verständnis in Form eines Computerprogramms auszudrücken“ (Weizenbaum 1978: 95). Ein guter Test, ob ein Phänomen - sei es physischer, psychischer oder sozialer Natur - verstanden wird, ist deshalb der Versuch, für dieses einen Algorithmus zu finden und in einem Programm nachzubauen.

Die Turing-These impliziert nicht, dass sich ***alles*** in Begriffen eines effektiven Verfahrens beschreiben lässt. Dies trifft sogar auf die ***wenigsten*** Problemstellungen zu.

- Bestimmte Probleme sind dadurch charakterisiert, dass wir nicht nur ***keinen Algorithmus kennen,*** sondern es lässt sich sogar beweisen, dass es ***keinen Algorithmus geben kann.*** Man spricht im diesem Fall auch von Nicht-Berechenbarkeit, im positiven Fall - also wenn es einen Algorithmus gibt - von Berechenbarkeit. Ein berühmtes Beispiel, das für die Arithmetik zugleich eine (negative) Antwort auf das Hilbertsche Entscheidungsproblem liefert, ist das Unvollständigkeitstheorem von Gödel 1931. Es besagt, dass es in einem beliebigen widerpruchsfreien arithmetischen System immer wahre arithmetische Sätze gibt, die aus diesem System nicht abgeleitet werden können (Krämer 1988: 146). Unter anderem bedeutet dies, dass es keinen Algorithmus gibt, der als Eingabe irgendeine Aussage über die natürlichen Zahlen enthält und dessen Ausgabe feststellt, ob diese Aussage wahr oder falsch ist. Was Gödel für den Bereich der Arithmetik gelang, wies Church 1936 für die Prädikatenlogik nach: es gibt keinen Algorithmus, der entscheiden kann, ob ein beliebiger Satz ein Theorem der Prädikatenlogik ist oder nicht (Unentscheidbarkeit der Prädikatenlogik). Ebenso gibt es keinen Algorithmus, der für jedes Computerprogramm entscheiden könnte, ob es jemals zum Halten kommt oder nicht (Halteproblem). Beispiele für weitere nicht-berechenbare Probleme finden sich in Goldschlager/Lister (1990). Die Frage, welche

Probleme durch ein algorithmisches Verfahren berechnet werden können, wird in der Informatik in der Theorie der Berechenbarkeit behandelt.

- Ist bekannt, dass es für ein Problem einen Algorithmus gibt, so ist nicht gesagt, dass dieser auch praktisch verwendbar ist, da die Ausführung infolge hoher Zeit- und Speicherressourcen realistischerweise oft nicht möglich ist. Ein effektives Verfahren kann eine Berechnung im Prinzip zwar durchführen können, dafür aber so lange brauchen, dass es praktisch wertlos ist. Mit diesen Fragen befasst sich die Komplexitätstheorie.
- Drittens - und dies dürfte in den Human- und Sozialwissenschaften der wichtigste Punkt sein - kann etwas nicht in ein effektives Verfahren gebracht werden, weil es (noch) nicht verstanden wird. Intuition lässt sich schlecht algorithmisieren (Weizenbaum 1978). Für eine Vielzahl von Problemen aus den Human- und Sozialwissenschaften dürfte mangelndes Verständnis der entscheidende Hinderungsgrund sein, sie einem algorithmischen Computermodell zuführen zu können. Eine Formalisierung ist dabei ein fundamentaler Schritt zum Verständnis eines Problems, „denn immer, wenn eine Problemlösung formalisierbar ist, und immer, wenn sie mechanisierbar ist, existiert ein Algorithmus, dessen Abarbeitung die Problemlösung ergibt“ (Krämer 1988: 139).

Abb. 7 zeigt in einer bildlichen Darstellung, wie verschwindend klein die berechen- *und* durchführbaren Aufgabenstellungen im Vergleich zu berechenbaren Aufgabenstellungen und dem Universum aller möglichen Aufgabenstellungen sind.

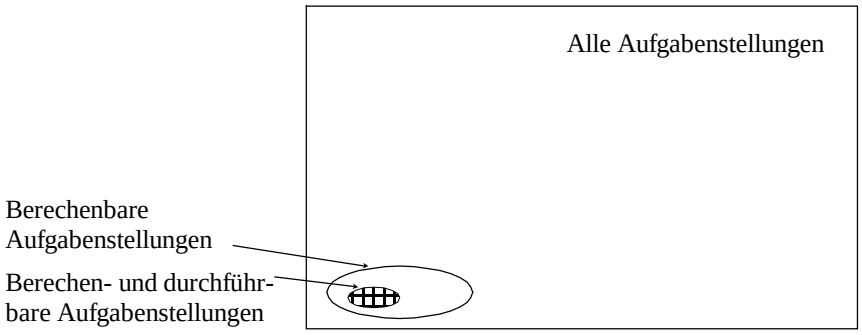

Abb. 7: Das Universum der Aufgabenstellungen (Goldschlager/Lister 1990: 94)

Kann ein Vorgang nicht durch einen Algorithmus beschrieben werden, dann gibt es somit keine Chance, ihn jemals durch einen Computer ausführen zu lassen - egal, wie groß oder schnell er ist. Die Rolle von Algorithmen ist damit grundlegend: ohne Algorithmus gibt es kein Programm und ohne Programm gibt es nichts auszuführen.

4. Programmiersprachen und Modellierungssoftware

Die Durchführung eines Prozesses auf einem Computer erfordert, dass
- ein Algorithmus entworfen wird, der beschreibt, wie der Prozess auszuführen ist,
- der Algorithmus als Programm in einer Programmiersprache formuliert wird,
- der Computer das Programm ausführt (Goldschlager/Lister 1990: 20).

Programmiersprachen sind Sprachen zur Spezifikation effektiver Verfahren. Eine Programmiersprache ist ein formales Notationssystem zur Beschreibung von Algorithmen, die von einem Computer ausgeführt werden sollen (Ghezzi/Jazayeri 1989: 375), und ein Programm kann als Implementation eines Algorithmus in einer bestimmten Programmiersprache angesehen werden (Winston 1984: 20).

Eine Programmiersprache ist prinzipiell
- eine formale Sprache (wie z.B. die der Prädikatenlogik 1. Stufe) und
- ein Medium, mit dem Computern Befehle erteilt werden können.

Programmiersprachen dienen also
- zum einen Menschen, die Programme schreiben und lesen, zur Beschreibung von Problemen,
- zum andern Computern, die Programme ausführen und Berechnungen durchführen.

Programmiersprachen lassen sich verschiedenen *Ebenen* zuordnen. Unmittelbar auf dem Computer ausführen lassen sich nur Algorithmen in *Maschinencode*, verfasst als Folge von 0-1-Tupeln. Befehle auf dieser untersten Ebene der Maschinensprache werden direkt vom Computer abgearbeitet, haben aber den Nachteil, dass sie schwer lesbar und hardwareabhängig sind. Man ging deshalb dazu über, in sog. niederen Programmier- oder *Assembler-Sprachen* die 0-1-Befehle durch mnemotechnische Buchstabenkombinationen abzukürzen (z.B. 10101010 durch ADD) und schließlich ganze Befehlsfolgen in einer einzigen Anweisung zusammenzufassen. Auf diese Weise entstand im historischen Verlauf eine Familie von *höheren oder problemorientierten Programmiersprachen*, die hardwareunabhängig und leicht lesbar sind. Der in diesen Sprachen erstellte Programmcode kann aber nicht unmittelbar maschinell ausgeführt werden, sondern muss mit einem eigenen Programm - Interpreter oder Compiler - in Maschineninstruktionen übersetzt werden.

Da jeder Programmtext in Maschinencode transferiert werden muss, lässt sich alles, was in einer höheren Programmiersprache formulierbar ist, prinzipiell auch auf Maschinenebene formulieren. Höhere Programmiersprachen machen den Computer nicht leistungsfähiger, aber für Menschen besser handhabbar. Alle Programmiersprachen sind also insofern *äquivalent*, als jede algorithmisch lösbare Aufgabe im Prinzip in jeder Sprache gelöst werden kann. Sie sind insofern *unterschiedlich*, als sich bestimmte Probleme mit bestimmten Sprachen besser lösen lassen als mit anderen.

Höhere Programmiersprachen fasst man in Sprachfamilien zusammen. Jeder Sprachfamilie kann eine bestimmte virtuelle Maschine zugeordnet werden, die aus einer bestehenden Allzweckmaschine eine neue, spezielle Maschine entwirft. Eine virtuelle oder gedachte Maschine schottet den Programmierer wie eine Schale von der Maschinenebene ab und es wird der Eindruck erzeugt, als „verstünde" die Maschine die Programmiersprache.[12] Auf programmiersprachlicher Ebene übernimmt diese Abschottung der Interpreter oder Compiler, der den Programmcode in Maschineninstruktionen übersetzt.

Die wichtigsten Sprachfamilien sind imperative, funktionale und logische Sprachen. Bei den *imperativen Sprachen* entsprechen Konzepte der Programmiersprache deutlich Maschinenoperationen und dem Turing-Maschinen-Modell. Ein Programm in einer imperativen Programmiersprache besteht aus einer Folge von sequentiellen Anweisungen an den Computer. Typische imperative Sprachen sind FORTRAN, PASCAL, ADA, ALGOL und C. Bei *funktionalen Programmiersprachen* werden Programme als Funktionen von Mengen von Eingabewerten in Mengen von Ausgabewerten betrachtet (Claus 1986: 461). Komplexere Funktionen werden durch Zusammensetzung aus einfachen Funktionen gebildet, wobei eine bestimmte Menge an Grundfunktionen vorgegeben ist. Dem Idealbild einer funktionalen Sprache kommt LISP am nächsten (Winston/Horn 1987). Bei *prädikativen oder logischen Programmiersprachen* wird Programmieren als Beweisen in einem System von Tatsachen und Schlussfolgerungen aufgefasst, und die Problembeschreibung erfolgt in einem logischen Formalismus (Claus 1986: 461). Logische Programmiersprachen realisieren die höchste Abstraktionsebene, was bei den üblichen von-Neumann-Rechnern auf Kosten der Effizienz geht. Eine typische logische Programmiersprache ist PROLOG (Clocksin/Mellish 1987).

Jede Programmiersprache ist im Hinblick auf bestimmte Problemstellungen entwickelt worden. FORTRAN wurde beispielsweise für technisch-naturwissenschaftliche Aufgaben konzipiert und PASCAL für Ausbildungszwecke. Im allgemeinen sind imperative Sprachen eher numerisch orientiert, d.h. sie eignen sich in erster Linie für Operationen mit Zahlen. PROLOG und LISP werden hingegen eher für nicht-numerische Problemstellungen eingesetzt, bei denen weniger Zahlen als Symbole und Zeichenketten manipuliert werden. Ein Überblick über die verschiedenen Sprachfamilien und Einsatzmöglichkeiten findet sich in Goldschager/Lister (1990) und Ghezzi/Jazayeri (1989).

Die naive Annahme, Computerprogrammierung bestehe im Schreiben sequentieller Anweisungen (wie sie in der Simulationsliteratur weit verbreitet ist, z.B. Apter 1971, Dörner 1984), ist falsch. Diese Auffassung trifft nur auf das imperative Modell zu, welches allerdings im allgemeinen und bei den Computermodellierern im besonderen das populärste ist. Richtig ist vielmehr, dass für jedes Programmierproblem ein bestimmtes *Verarbeitungsmodell* zugrunde gelegt wird, die Idee einer bestimmten Maschine, nach deren Vorstellung wir Algorithmen entwerfen. „Dieses Verarbeitungsmodell bestimmt die

---

[12] Im weiteren Sinn versteht man unter einer virtuellen Maschine jede durch Software hervorgerufene Funktionsänderung einer Rechenmaschine. Ein Schachprogramm ändert z.B. eine reale Maschine in die virtuelle Maschine „Schachautomat" um, die sich wie ein Schachspieler verhält (Ludewig 1985: 28).

Konstruktion des Algorithmus und damit auch die sprachlichen Erfordernisse, Formen und Strukturen für seine Implementierung. Es prägt den Stil, in dem das Programm geschrieben ist, den Programmierstil“ (Görz 1988: 73).[13] Die *Vorstellung* von einem Verarbeitungsmodell begründet also den Programmierstil und dieser kann durch Programmiersprachen unterstützt bzw. behindert werden (Stoyan 1988: 21).

Der Aussage von Dörner (1984: 346), nach der die Charakteristik einer Sprache nur eine geringe Rolle für die Modellkonstruktion spielt und man bei genügendem Programmiergeschick mit jeder Sprache alles machen kann, können wir nicht zustimmen. Zwar kann man „im Prinzip“, wie oben erwähnt, mit jeder Sprache alles machen, aber es gibt für die Modellierung im allgemeinen sowie für bestimmte Problemstellungen im besonderen geeignete und weniger bis völlig ungeeignete Sprachen. Ein sozialpsychologisches Modell in Maschinensprache zu implementieren scheint ebenso wenig angebracht wie ein quantitatives Modell, das hochkomplexe numerisch-statistische Verfahren verwendet, in einer symbolisch orientierten Sprache zu formulieren. Ghezzi/Jazayeri (1989: 25-27) stellen Forderungen an Programmiersprachen auf wie Programmierbarkeit (Probleme in angemessener Weise formulieren), Einfachheit und Lesbarkeit, Strukturierung, Effizienz und Portabilität (Übertragbarkeit auf andere Rechnerfamilien). Für die Computermodellierung haben diese Forderungen unterschiedliches Gewicht. Effizienz spielt hier z.B. eine geringere Rolle als in kommerziellen Anwendungen, da die Modelle in der Regel nicht zeitkritisch sind. Hingegen haben Lesbarkeit, Strukturierung und Portabilität eine größere Bedeutung. Maschinennahe Sprachen sind damit völlig ungeeignete Modellsprachen, da sie zwar effizient sind, aber Lesbarkeit, Ausdruckskraft und Portabilität nicht hinreichend gegeben sind.

Höhere Programmiersprachen erlauben eine Programmierung, die sich nahe an die Alltagssprache anlehnt (Schnell 1990: 117), und sind grundsätzlich zu bevorzugen. Von den gebräuchlichen imperativen Sprachen C, FORTRAN, BASIC und PASCAL ist C populär, effizient und hardwarenah, allerdings schwer lesbar und schwierig zu lernen. C hat bislang keine große Basis als sozialwissenschaftliches Modellierungswerkzeug, obwohl es unter professionellen Programmierern die bevorzugte Sprache ist. FORTRAN hat als quantitativ ausgerichtetes Modellierungswerkzeug in den Sozialwissenschaften eine gewisse Tradition und Existenzberechtigung. Aufgrund der mangelnden Strukturierungsmöglichkeit (goto-Sprungbefehl) und der damit verknüpften schweren Lesbarkeit ist FORTRAN aber eher zu meiden (was für neuere Versionen nicht mehr in dem Maße gilt).

Aufgrund ihrer Einfachheit und Ausdruckskraft erscheint PASCAL als ideale imperative Programmiersprache: es unterstützt die disziplinierte Programmierung, ist leicht lesbar und

---

[13] Das Verarbeitungsmodell ist nicht unbedingt mit einer bestimmten Sprachfamilie verknüpft. Hat man z.B. die Vorstellung, dass Algorithmen immer sequentielle Anweisungen an die Maschine sind, so wird man in LISP oder PROLOG anweisungsorientiert programmieren wie in FORTRAN oder C - was durchaus möglich ist. Allerdings ist dies schlechter Programmierstil, da diesen Sprachen eine andere „Philosophie“ unterliegt (Stoyan 1988).

lernbar sowie weit verbreitet. Hingegen mangelt es PASCAL an gewissen numerischen Fähigkeiten. Schnell (1990) tritt insbesondere für PASCAL als Allzweck-Sprache ein: „Die elementaren Grundlagen einer Sprache wie PASCAL, die notwendig sind, um einfache Programme lesen zu können, sind in sehr kurzer Zeit erlernbar. Die notwendige Lerndauer liegt mit Sicherheit unterhalb der entsprechenden Dauer für den Erwerb der Fähigkeiten für das Verständnis von Differentialgleichungen oder phänomenologischen Texten“ (Schnell 1990: 116-117).

Von den nicht-imperativen Sprachen sind LISP und PROLOG die gebräuchlichsten. Beide Sprachen werden in der nicht-numerischen, „qualitativen“ Modellierung bevorzugt. LISP wurde z.B. in der theoretischen Biologie zur Modellierung von Sozialbeziehungen von Hummeln verwendet (Hogeweg/Hesper 1985), PROLOG bei der Modellierung von Schemata-Konzepten zur Erklärung sozialer Strukturen (Banerjee 1986). In PROLOG muss i.a. weniger Code geschrieben werden als in LISP, und die Konzentration auf die Natur des Problems scheint eher möglich zu sein. PROLOG-Code ist zudem auch für Nicht-Programmierer leichter lesbar als LISP-Code: „vermutlich kann auch jemand, der Prolog nicht kennt, die Bedeutung des Prolog-Textes 'erraten', während dies bei der Lösung in Lisp kaum möglich ist“ (Schnupp/Nguyen Huu 1987: 4). Die Bevorzugung von PROLOG vor LISP hat aber noch andere, wichtigere Gründe, auf die wir später eingehen. Die eben aufgestellten Behauptungen können leicht anhand des einfachen - allerdings numerischen - Beispiels in Abb. 8 verifiziert werden.

```
/* PROLOG */

bevoelkerung(usa, 203).
bevoelkerung(indien, 548).
flaeche(usa,3).
flaeche(indien, 548).

bev_dichte(Land, Dichte) :-
   bevoelkerung(Land, Bev),
   flaeche(Land, Fl),
   Dichte is Bev/Fl.

/* LISP */

(defun bevoelkerung land
   (cond ((eq land 'usa) 203)
         ((eq land 'indien) 548) ))

(defun flaeche land
   (cond ((eq land 'usa) 3)
         ((eq land 'indien) 3) ))

(defun bev-dichte land
   (div (bevoelkerung land) (flaeche land) ))
```

Abb. 8: Vergleich von PROLOG und LISP anhand des einfachen Problems der Ermittlung der Bevölkerungsdichte (modifiziert nach Schnupp/Nguyen Huu 1987: 3)

Neben Programmiersprachen werden in der Computermodellierung oft höhere Modellierungswerkzeuge eingesetzt, die bestimmte Hilfsmittel bereitstellen und/oder auf

spezielle Anwendungsprobleme zugeschnitten sind. Es gibt eine große Vielfalt solcher Werkzeuge, eine Einordnung und Bewertung findet sich z.B. in Zeigler (1976) oder Kreutzer (1986).

Möhring (1990: 20-21) unterscheidet vier verschiedene Typen von Modellierungssystemen.

1. Universelle Programmiersysteme sind die eben angesprochenen Programmiersprachen (PASCAL, C, FORTRAN etc.), bei denen der Benutzer ***keine Unterstützung*** bezüglich des speziellen Anwendungszwecks erhält. Dafür kann er flexible, an die eigenen Erfordernisse angepasste Simulationsprogramme erstellen.
2. Simulationsorientierte Simulationssysteme (z.B. SIMULA, SIMSCRIPT, GASP) unterscheiden sich in ihrer Mächtigkeit kaum von allgemeinen Programmiersprachen, bieten dem Benutzer aber Unterstützung für die Erstellung von Simulationsprogrammen, wie etwa Zufallszahlengeneratoren.
3. Modellierungsorientierte Simulationssysteme (z.B. GPSS, DYNAMO, CSMP) stellen bestimmte Modellkomponenten bereits zur Verfügung (z.B. Berechnung von Ableitungen, Integralfunktionen). Da immer nur bestimmte Modellklassen unterstützt werden, sind die Modellbeschreibungsmöglichkeiten hierbei eingeschränkt.
4. Anwendungsorientierte Simulationssysteme (z.B. RAILSIM, HOSPSIM,XCERT) zeichnen sich durch einen zusätzlichen Anwendungsbezug aus und schränken die Einsatzmöglichkeiten weiter ein. In diesen Systemen wird die entsprechende Fachterminologie berücksichtigt bis hin zur „naturgetreuen“ Abbildung und Visualisierung von Simulationsabläufen per Computeranimation.

Von (1) nach (4) bieten die Modellierungssysteme mehr und mehr Unterstützung und der Modellierungsaufwand nimmt ab, weil Mittel zur Verfügung gestellt werden, die zu übersichtlicheren und kürzeren Modellbeschreibungen führen. Gleichzeitig sinkt die Modellierungsflexibilität, d.h. die Möglichkeiten der Modellierung insgesamt werden mehr und mehr eingeschränkt.

Aufgrund ihrer Komplexität und der Vielfalt ihres Untersuchungsgegenstandes fordert Möhring (1990: 21) für die Sozialwissenschaften Werkzeuge mit einem breit gefächerten Modellierungsinstrumentarium, gleichzeitig aber auch größtmögliche Problemnähe des Beschreibungsformalismus. Möhring entwickelt eine eigene funktionale Modellbeschreibungssprache MIMOSE, deren Vorzug gegenüber klassischen Programmiersprachen allerdings nicht unmittelbar einleuchtet.

Solche Simulationssysteme werden in den Sozialwissenschaften - mit Ausnahme von DYNAMO - auch kaum genutzt. Vielmehr sind die meisten neueren sozialwissenschaftlichen Simulationen in allgemein verfügbaren Hochsprachen wie FORTRAN, PASCAL, LISP oder PROLOG geschrieben (Schnell 1990: 116). Diese Sprachen sind flexibel und allgemein genug für Modellierungszwecke, werden - im Gegensatz zu speziellen Simulationswerkzeugen - von einem breiten, formal geschulten Benutzerkreis verstanden und sind leicht zu lernen. Ein wichtiger Grund für die Verwendung von

Hochsprachen liegt darin, dass es kein Simulationsprogramm geben kann, das nicht mittels der Hochsprachen realisiert werden kann, während die Spezialsoftware die Klasse möglicher Modelle zu stark einschränkt (Schnell 1990: 116). Das bei höheren Programmiersprachen fehlende Modellierungsinstrumentarium lässt sich eventuell durch Programm-Bibliotheken und Hilfsprogramme ersetzen. Schnell (1990) weist z.B. auf eine vollständige DYNAMO-Bibliothek in PASCAL und auf eine Sammlung von PASCAL-Hilfsprogrammen für eine große Zahl unterschiedlicher Simulationstypen hin.

Im Gegensatz zu Möhring (1990) wird hier also die Meinung vertreten, dass die üblichen höheren Programmiersprachen ideale Modellierungssprachen sind und jeder formal geschulte Sozialwissenschaftler

- in der Lage sein müsste, eine solche zu erlernen und
- diese als Modellierungssprache einzusetzen.

Abschließend soll noch auf einen Aspekt hingewiesen werden, der im Lauf dieses Buchs immer wieder relevant wird und der den *Aufbau* von Computerprogrammen betrifft. Moderne Computerprogramme können als formale Texte betrachtet werden, die in hierarchisch strukturierte Einzelteile zerfallen. Die hierarchischen Strukturen sind eine Folge der schrittweisen Verfeinerung von Problemlösungen, was in der Informatik mit „teile-und-herrsche“ beschrieben wird. „Der Grundgedanke ist, den auszuführenden Prozess in zahlreiche Einzelschritte zu zerlegen, von denen jeder durch einen Algorithmus beschrieben werden kann, der weniger umfangreich und einfacher ist, als der für den ursprünglichen ganzen Prozess“ (Goldschlager/Lister 1990: 31-32). Auf diese Weise zerfällt ein Programm in *hierarchische Programmschichten*, so dass es einfache Teilprogramme (Prozeduren) gibt, welche primitive Operationen auf einer niedrigen Ebene ausführen und Prozeduren auf einer höheren Ebene, die die einfachen Prozeduren verwenden. Mit anderen Worten: Teilprogramme oder Prozeduren lösen bestimmte Aufgaben, wobei diese Teilprogramme wiederum Teilprogramme benutzen, die noch elementarere Aufgaben lösen usf. Dieses Aufbauprinzip ist in der Computerwissenschaft Ausdruck der bedeutenden Grundidee der *Prozedurabstraktion*, nach der der Prozess der Erzeugung neuer Prozeduren durch Kombination bereits bestehender erfolgt. Prozedurabstraktion befreit also den Modellierer von ablenkenden Details, indem vorhandene Prozeduren verwendet werden (Winston/Horn 1987: 45).

5. Klassifikationsschemata von Computermodellen

Bei der Klassifikation von Computermodellen hält man sich in der Literatur meist an eines der folgenden Prinzipien. Die Modelle werden entweder inhaltlich nach dem Gegenstands- bzw. Anwendungsbereich geordnet oder aber mit Hilfe eines Schemas nach formalen Merkmalen. Seinem klassischen Überblicksartikel „Simulation of Social Behavior“ legt Abelson (1968) z.B. folgende einfache inhaltliche Klassifikation zugrunde.

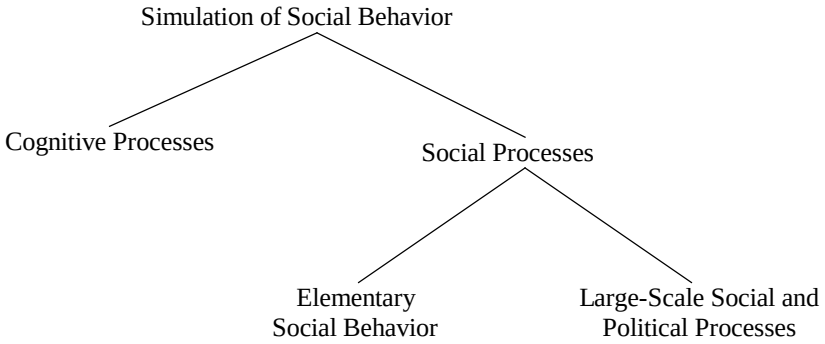

Abb. 9: Klassifikation sozialwissenschaftlicher Computermodelle nach inhaltlichen Kriterien (nach Abelson 1968)

Informationshaltiger und brauchbarer erscheinen formale Ordnungsschemata. Ein solches Klassifikationsschema nach Merkmalsklassen wie Verwendungszweck, Zeitabhängigkeit etc. liefert Troitzsch (1990: 12-21). Wir geben das Schema in der gerafften Form von Möhring (1990: 9-11) wieder, die Merkmalsklasse Bildbereich wurde in Tab. 1 bereits genannt.

Tab. 2: Klassifikation von Computermodellen nach formalen Kriterien (Möhring 1990)

Verwendungszweck	
Deskription:	Beschreibung von Zusammenhängen
Prognose:	Ableitung zukünftiger möglicher Entwicklungen
Entscheidung:	Ableitung von Prognosen durch Ausprobieren und Bewertung unterschiedlicher Strategien
Didaktik:	Vermittlung wissenschaftlicher Erkenntnisse
Zeitabhängigkeit	
Statisch:	Unabhängigkeit der Modelle von der Zeit
Dynamisch:	Zeitliche Veränderbarkeit von Modellstruktur und Modellverhalten
Zeitstruktur	
Diskret:	Eintreffen von Modellereignissen zu diskreten Zeitpunkten (z.B. Differenzengleichungen)
Kontinuierlich:	„Fließendes“ Eintreffen von Modellereignissen (z.B. Differentialgleichungen)
Grad der Determiniertheit	
Deterministisch:	Präzise Vorhersage von Modellereignissen
Stochastisch:	Wahrscheinlichkeitsaussagen zu zukünftigen Modellereignissen
Allgemeine Modellstruktur	
Makromodell:	Modelle bestehen aus einem Element

Mikro-/Mehrebenenmodell: verschiedenen Ebenen	Modelle enthalten mehrere Elemente auf

Allgemeine Struktur des Modellverhaltens

Linear:	Beziehungen zwischen Modellelementen sind durch lineare Funktionen beschreibbar
Nichtlinear:	Beziehungen zwischen Modellelementen sind durch allgemeine Funktionen beschreibbar

Wertebereich der Merkmale von Modellelementen

Qualitativ:	Formalisierung qualitativer Eigenschaften durch nominal- bzw. ordinalskalierte Wertebereiche
Quantitativ:	Ratioskalierte Wertebereiche

4. Zum Nutzen von Computermodellen

In der Literatur wird eine Vielzahl von Argumenten angeführt, die den positiven Einfluss der Verwendung von Computern auf die Theorien- und Modellbildung in den empirischen Wissenschaften explizit machen. Das folgende Schema versucht die wichtigsten Argumente in vier Punkten zusammenzufassen.[14]

1. Formalisierung ***und*** Ausführbarkeit

Da Programmiersprachen exakt definierte Kunstsprachen sind, bilden Programmiersprachen im Prinzip formale Sprachen wie die von Differentialgleichungen, Prädikatenlogik oder Graphentheorie. ***Computermodelle*** besitzen folglich die ***Eigenschaften formal-mathematischer Modelle*** und unterstützen damit den obengenannten Zwang zur Explikation von Theorien. Wie andere Formalisierungsformen ist Computermodellierung ein Mittel der Begriffsklärung, da in Computermodellen Begriffe ebenso präzisiert, einheitlich verwendet und Zusammenhänge explizit beschrieben werden müssen. Programmiersprachen haben aber im Vergleich zu anderen formalen Sprachen darüber hinaus den Vorteil, dass der ***formale Text von der Maschine bearbeitet*** werden kann, so dass das Verhalten des Rechners von dem Modell determiniert wird. Während ein verbales oder formales, nicht computerisiertes Modell eine abstrakte, undynamische Struktur darstellt, „macht" das Computermodell etwas: „Ein Simulationsmodell 'verhält sich'; es ist etwas Konkretes und lenkt unmittelbar den Blick auf merkwürdige, falsche oder unvollständige Konzeptualisierungen" (Dörner 1984: 346). Im Fall der Repräsentation theoretischer Aussagen führt der Rechner zu einer „Dynamisierung theoretischer Aussagen" (Harbordt 1974: 262); er zeigt das „Verhalten" der Theorie: „When that computer is programmed to represent one's theory, its processes are then synonymous with those of the theory, and pressing the GO button sets the ***theory*** in motion. Its processes begin literally to move, their consequences interact, evolve something new, and thus

[14] Vgl. zum folgenden insbesondere Gullahorn/Gullahorn (1965), Apter (1971), Ziegler (1972), Harbordt (1974), Dörner (1984), Lischka/Diederich (1987), Schnell (1990).

realize the theory's extended consequences. It makes a theory 'go somewhere'„ (McPhee 1960: 4, Hervorhebung McPhee).

2. Deduktive Mächtigkeit

Einer der zwingendsten Gründe für die Anwendung von Computern ist der, dass sie die blinden Flecke in unserem Denken bloßlegen (Weizenbaum 1978: 96). Im Gegensatz zum Menschen ist beim Computer die Ableitungsfähigkeit nicht beschränkt. Die Simulation ermöglicht schnelle und korrekte Ableitungen auch aus einem komplexen, für Menschen nur mehr schwer oder überhaupt nicht mehr überschaubaren Aussagensystem (Harbordt 1974: 270). Programme lassen sich durch eine Fülle von Daten nicht verwirren, sondern führen ***vollständige*** und ***korrekte*** Ableitungen durch. Da ein Simulationsmodell nur ablauffähig ist, wenn alle benötigten Informationen bereitgestellt werden, kann das Explizit-machen-müssen theoretischer Annahmen die Freilegung von Wissenslücken zur Folge haben. Beispielsweise lässt sich dadurch entdecken, dass bestimmte theoretische Konsequenzen nur bei Vorliegen bestimmter - bislang nicht explizit gemachter - Annahmen auftreten. Sylvan/Glassner (1985) entdeckten z.B., dass die Konflikt-Kohäsions-Behauptung von Simmel nicht aus seinen Annahmen ableitbar ist. Umgekehrt kann die Durchführung einer Simulation zur Entdeckung ***unerwarteter*** Effekte führen, wie sie etwa Schnell (1990) anhand einer Arbeit über die Bedeutung sozialer Netzwerke für die Entstehung kollektiver Bewegungen schildert.

3. Experimenteller Charakter

Die vordergründigste - und in den genannten Punkten bereits angeklungene - Eigenschaft von Computermodellen ist die ungehinderte Möglichkeit zum Experimentieren. Der Computer kann insofern ein neues Verhältnis zu Theorien eröffnen, als „er die Möglichkeit bietet, nicht nur natürliche, sondern auch künstliche Realisationen eines theoretischen Konzepts in ihrem Verhalten zu untersuchen und so Theorien zusätzlichen Prüfungen unterziehen zu können“ (Kobsa 1986: 123-124). Dies ist besonders wichtig in Wissenschaftsbereichen, die sich aus ethischen, physischen oder sonstigen Gründen der experimentellen Methode verschließen. Computermodelle ermöglichen beliebige Experimente, indem entweder die Inputs oder das Programm selbst (z.B. bestimmte Zusammenhänge) variiert und die Folgen beobachtet werden. Der experimentelle Charakter von Computermodellen hat im einzelnen etwa folgende positive Auswirkungen:

- Alternative Annahmen/Hypothesen und deren Konsequenzen können durchgespielt werden.
- Modelle können z.B. in allen Einzelheiten gleich gemacht werden außer in dem kritischen Aspekt, unter dem die Modelle verglichen werden sollen (Lindenberg 1971: 100).
- Der Gültigkeitsbereich eines Modells kann getestet werden, indem man Extrem-bedingungen untersucht (Lischka/Diederich 1987: 28).
- Die Robustheit des Modells und die Bedeutung einzelner Komponenten kann erfasst werden, indem man Teile entfernt oder hinzufügt (Lischka/Diederich 1987: 28).
- Unbegrenzte „mögliche Welten“ und „What-would-be-if“-Szenarios lassen sich realisieren.

4. Universalität

Kobsa (1986: 123) weist darauf hin, dass es derzeit kaum „Unmöglichkeitsbeweise“ gibt, also theoretische Grenzen der Anwendung des Computers als Hilfsmittel bei der Theoriebildung. Einzige Ausnahme dürfte dabei das oben genannte Gödelsche Unvollständigkeitstheorem sein, dessen Auswirkung auf das „High-Level-Verhalten“ des Computers aber noch sehr unklar ist. Verschiedene Computertheoretiker setzen sich z.B. mit der Frage auseinander, ob das Gödel-Theorem intelligente Maschinen prinzipiell unmöglich macht (vgl. hierzu die Literaturhinweise in Kobsa 1986). Solche Fragen sind in unserem Kontext aber nicht weiter relevant. Aus unserer Sicht ist ein Computer ein Allzweck-Werkzeug, das prinzipiell programmiert werden kann, um das Verhalten jedes anderen Systems nachzumachen (Gullahorn/Gullahorn 1965: 433). Eine prinzipielle Beschränkung ergibt sich weder für bestimmte Wissenschaftsgebiete (Sozial- versus Naturwissenschaften), noch für bestimmte Modellarten (z.B. quantitative versus qualitative Modelle).

LITERATUR

Abelson, R.P. (1968). Simulation of Social Behavior. In G. Lindzey/E. Aronson (eds.), *The Handbook of Social Psychology* Vol. 2 (2.Aufl.). Reading, Mass: Addison-Wesley, S.274-356.

Apter, M.J. (1971). *The Computer Simulation of Behaviour.* New York: Harper & Row.

Banerjee, S. (1986). Reproduction of Social Structure. An Artificial Intelligence Model. *Journal of Conflict Resolution*, 30, 2, S. 221-252.

Claus, V. (1986). *Informatik.* Mannheim: Bibliographisches Institut.

Clocksin, W.F./Mellish, C.S. (1987). *Programming in Prolog* (3. Aufl.). Berlin: Springer.

Colby, K.M. (1973). Simulations of Belief Systems. In R.Schank/K.M.Colby (eds.), *Computer Models of Thought and Language*. San Francisco: Freeman, S. 251-286.

Colby, K.M. (1975). *Artificial Paranoia: A Computer Simulation of Paranoid Processes.* New York: Pergamon.

Dörner, D. (1984). Modellbildung und Simulation. In E.Roth (ed.), *Sozialwissenschaftliche Methoden*. München: Oldenbourg, S. 337-350.

Ghezzi, C./Jazayeri, M. (1989). *Konzepte der Programmiersprachen: begriffliche Grundlagen, Analyse und Bewertung*. München: Oldenbourg.

Goldschlager, L./Lister, A. (1990). *Informatik. Eine moderne Einführung* (3.Aufl.). München: Hanser.

Görz, G. (1988). *Strukturanalyse natürlicher Sprache*. Bonn: Addison-Wesley.

Gullahorn, J.T./Gullahorn, J.E. (1965). The Computer as a Tool for Theory Development. In D.Hymes (ed.), *The Use of Computers in Anthropology*. London: Mouton, S.427-448.

Harbordt, S. (1974). *Computersimulation in den Sozialwissenschaften* 2 Bände, Reinbek bei Hamburg: Rowohlt.

Hogeweg, P./Hesper, B. (1985). Socioinformatic Processes: MIRROR Modelling Methodology. *Journal of Theoretical Biology*, 113, S.311-330.

Kleinknecht, R./Wüst, E. (1976). *Lehrbuch der elementaren Logik* 2 Bände. München: dtv.

Kobsa, A. (1984). What is Explained by AI Models. *Communication & Cognition*, 17, 2/3, S.49-65.

Kobsa, A. (1986). Artificial Intelligence und Kognitive Psychologie. In J.Retti et al (eds.), *Artificial Intelligence. Eine Einführung* (2.Aufl.). Stuttgart: Teubner, S.105-130.

Krämer, S. (1988). *Symbolische Maschinen. Die Idee der Formalisierung im geschichtlichen Abriß*. Darmstadt: Wissenschaftliche Buchgesellschaft.

Kreutzer, W. (1986). *System Simulation. Programming Styles and Languages*. Sidney: Addison-Wesley.

Lindenberg, S. (1971). Simulation und Theoriebildung. In H.Albert (ed.), *Sozialtheorie und soziale Praxis*. Meisenheim: Hain, S.78-113.

Lischka, C./Diederich, J. (1987). Gegenstand und Methode der Kognitionswissenschaft. *GMD-Spiegel*, 2/3, S.21-32.

Ludewig, J. (1985). *Sprachen für die Programmierung. Eine Übersicht*. Mannheim: Bibliographisches Institut.

McPhee, W.N. (1960). Computer Models and Social Theory: An Introduction, Paper Presented at the Annual Meetings of the American Sociological Association, New York.

Möhring, M. (1990). *Mimose. Eine funktionale Sprache zur Beschreibung und Simulation individuellen Verhaltens in interagierenden Populationen*. Dissertation, Universität Koblenz-Landau.

Schnell, R. (1990). Computersimulation und Theoriebildung in den Sozialwissenschaften. *Kölner Zeitschrift für Soziologie und Sozialpsychologie*, 42, 1, S.109-128.

Schnupp, P./Nguyen Huu, C.T. (1987). *Expertensystem-Praktikum*. Berlin: Springer.

Stoyan, H. (1988). *Programmiermethoden der Künstlichen Intelligenz* Band 1. Berlin: Springer.

Troitzsch, K. (1990). *Modellbildung und Simulation in den Sozialwissenschaften*. Opladen: Westdeutscher Verlag.

Weizenbaum, J. (1978). *Die Macht der Computer und die Ohnmacht der Vernunft*. Frankfurt a.M.: Suhrkamp (zuerst 1976: Computer Power and Human Reason. From Judgement to Calculation. San Francisco: Freeman).

Winston, P.H. (1984). *Artificial Intelligence* (2.Aufl.). Reading, Mass: Addison-Wesley.

Winston, P.H./Horn, B.K.P. (1987). *Lisp*. Bonn: Addison-Wesley (zuerst 1984: Lisp. Reading, Mass: Addison-Wesley).

Zeigler, B.P. (1976). *Theory of Modeling and Simulation*. New York: Wiley.

Ziegler, R. (1972). *Theorie und Modell. Der Beitrag der Formalisierung zur soziologischen Theoriebildung*. München: Oldenbourg.