Georg Trogemann

Experimentelle und spekulative Informatik

Es ist möglich, den Effekt einer Rechenmaschine zu erreichen, indem man eine Liste von Handlungsanweisungen niederschreibt und einen Menschen bittet, sie auszuführen. Eine derartige Kombination eines Menschen mit geschriebenen Instruktionen wird ›Papiermaschine‹ genannt. Ein Mensch, ausgestattet mit Papier, Bleistift und Radiergummi sowie strikter Disziplin unterworfen, ist in der Tat eine Universalmaschine.

Alan M. Turing[1]

Turings Feststellung erfaßt sehr präzise den Kern unseres klassischen Verständnisses von Algorithmen und Computern. Probleme der Berechenbarkeit lassen sich im Prinzip auch ohne Rechner, mit Papier und Bleistift am Küchentisch sitzend, lösen. Im Zentrum der klassischen Berechenbarkeitstheorie stehen klar spezifizierbare Aufgaben, deren Lösung durch Abfolgen von Handlungsanweisungen (Befehlen) beschrieben werden. Nicht nur, daß der Algorithmus komplett im Kopf des Programmierers entwickelt wird, er kann im Prinzip auch im Kopf ausgeführt werden. Ob das Verfahren am Ende auch noch auf einen realen Rechner implementiert wird, ist zweitrangig, von der Ausführung des Codes werden keine neuen Qualitäten erwartet. Wir wollen im Folgenden zeigen, daß bereits unsere aktuellen Anwendungen sich nicht mehr adäquat auf der Basis einer solchen passiven Maschine beschreiben lassen. Die inzwischen in verschiedenen Anwendungsbereichen formulierten Problemstellungen verlangen nicht nur nach vollkommen neuen Konzepten und Vorgehensweisen, sondern auch den radikalen Abschied von der Vorstellung, daß diese Systeme vollständig beherrschbar und in ihrem Verhalten vorhersagbar sind.

Computerpraxis – Drei Beispiele

Im Projekt *Talking Heads* von Luc Steels und Frédéric Kaplan versuchen visuell basierte und miteinander im Wechselwirkung stehende Roboter ohne vorhergehende Festlegung im Design des Programms oder durch menschliche Intervention eine eigene Sprache zu begründen und eine gemeinsame Ontologie zu entwickeln. Die beiden Roboter sind hierbei gleichzeitig Beobachter und Teilnehmer eines Kommunikationsprozesses. Das gemeinsame Vokabular der Roboterkommunikation über die Form, Farbe und Position von Objekten auf einer Tafel wird nicht von außen aufgeprägt, sondern entwickelt sich während der »Konversation« der Roboter. Kommunikation erzeugt hier neue Kommunikationsmöglichkeiten. Zwischen den Programmen findet kein Informations-

1. »Intelligent Machinery«, zit. nach: Alan M. Turing, *Intelligence Service*, Hg. B. Dotzler/F. Kittler, Berlin 1987, S. 91.

Talking Heads. Zwei steuerbare Kameras, die auf eine Tafel gerichtet sind und sich über das Gesehene in einer gemeinsam entwickelten Sprache austauschen.

austausch in herkömmlichen Sinne statt, da den Kommunikationspartnern nicht wie in der üblichen technischen Kommunikation von vornherein eine gemeinsame Semantik übergestülpt ist. Die Teilnehmer sind so gezwungen, eigene Interpretationen des Geschehens zu entwickeln. Mißverständnisse und Mißdeutungen im Kommunikationsprozeß sind hierbei keine Störungen im Sinne technischer Übertragungsfehler, sondern systemimmanente Voraussetzung für die Entwicklung der gemeinsamen Sprache.[2]

Im Computerspiel *The Sims* wird das normale Leben simuliert. Man lebt in einer Nachbarschaft mit anderen Familien und geht alltäglichen Beschäftigungen nach – Arbeiten, Fernsehen, Essen, Freunde treffen, usw. Es ist möglich, die autonomen Sims ihren Alltag selbst bewältigen zu lassen oder in ihr Verhalten einzugreifen. Der normale Spieler wird den Sims ihre Autonomie lassen, um nur gelegentlich steuernd einzugreifen. Da das Spiel keinerlei Gewalt vorsieht, ist es für Personen jeden Alters freigegeben. Wir lassen das Spiel einen achtjährigen Jungen spielen. In seiner Sims-Familie wird im Laufe des Spiels ein Baby geboren, um das er sich von nun an kümmern muß. Dies ist ihm fürchterlich lästig, wenn er aber nichts unternimmt, wird das Baby schreien, das Jugendamt oder

2. Luc Steels/Frédéric Kaplan u.a., »Crucial factors in the origins of word-meaning«, in: A. Wray (Hg.), *The Transition to Language*, Oxford 2002. [http://www.csl.sony.fr/downloads/papers/2001/steels-crucial-2001.pdf]; dies., »Situated grounded word semantics«, in: T. Dean (Hg.), *Proceedings of IJCAI 99*, San Francisco 1999, S. 862-867 [http://www.csl.sony.fr/downloads/papers/1999/ijcai99.pdf].

Painstation

die Polizei wird kommen und ihm Schwierigkeiten machen. Da es nun bei *Sims* unter anderem Möglichkeiten gibt, Häuser zu bauen und zu erweitern, hat der Junge eine Idee. Er entschließt sich, das Baby einfach einzumauern. Als dennoch die Polizei auftaucht, wird diese ebenfalls eingemauert. Solche Spielverläufe waren von den Spielentwicklern nie vorgesehen, sie emergieren aus der Komplexität und Offenheit des Spiels. Muß *The Sims* auf den Index der verbotenen Spiele?

Die *Painstation* ist ein Projekt von Volker Morawe und Tilman Reif. Das bekannte Videospiel *Pong* – eine einfache Tischtennissimulation – wird um eine Pain-Execution-Unit (PEU) erweitert. Über einen Drehknopf ist der Schläger zu steuern, mit dem der virtuelle Ball zurückspielt werden kann. Wird der Ball verfehlt, führt dies zu Strafpunkten. Die Überschreitung einer bestimmten Punktezahl führt zur körperlichen Bestrafung des Spielers. Die PEU fügt dann der linken Hand des Spielers durch Stromstöße, Peitschenhiebe oder Erhitzung reale Schmerzen zu. Wie bei jeder interaktiven Anwendung ist hier der steuernde Algorithmus lediglich eines der Glieder eines geschlossenen Mensch-Maschine-Zyklus. Das Programm berechnet Ausgabewerte, die dem Benutzer über das Interface angeboten werden. Der Benutzer reagiert auf das Gesehene/Gehörte/Gefühlte und erzeugt durch seine Aktionen seinerseits Eingabenwerte für das System. Diese Eingaben wiederum führen zu neuen Ausgaben der Maschine, usw. Bei allen derartigen interaktiven Handlungsprozessen muß eine geeignete Grundgeschwindigkeit in der Wiederholrate gewährleistet sein, damit

die Interaktion als Ganzes funktioniert. Ist die Wiederholrate zu gering, bricht die gesamte Interaktion zusammen und die Funktionsfähigkeit des Systems ist nicht mehr gegeben. Gleichzeitig hat diese Form der Interaktion eine vollkommen andere Qualität als numerische Ein-/Ausgabewerte. In einem Raum zu navigieren, Töne zu hören, Schmerzen zu spüren sind Wahrnehmungen. Ihre eigentliche Qualität, die sinnliche Erfahrung, kann nicht adäquat durch Zahlen, d.h. die Ein-/Ausgabewerte des Programms beschrieben werden. Am Beispiel der *Painstation* ist sehr leicht zu erkennen, daß die Qualität der Apparatur nicht in den einzelnen Komponenten – Software, Interface oder Spieler – zu suchen ist, sondern nur das Gesamtsystem im laufenden Betrieb seine Qualität entfaltet. Die durch das Interface wahrgenommenen Qualitäten sind oft im Programmcode nicht explizit repräsentiert und auch vom Entwickler nicht intendiert. Der Algorithmus ist zwar Kernstück eines Gesamtsystems, für sich alleine ist er aber wenig von Interesse und kann isoliert nur bedingt einer sinnvollen Analyse unterzogen werden.

Dies sind nur einige – mehr oder weniger willkürliche – Beispiele, die zeigen sollen, daß avancierte Anwendungen nicht mehr mit den Anfängen des Rechnereinsatzes zu vergleichen sind. Softwareagenten, Würmer, Viren, etc., sind nicht mehr an eine zugewiesene Hardware gebunden, sie bewegen sich frei durch die Netze und treten hierbei von sich aus in Kommunikation mit anderen Algorithmen. Adaptive Algorithmen sind lernfähig und in der Lage sich selbst zu modifizieren, sie ändern ihren eigenen Code oder schreiben neue Programme und bringen diese zur Ausführung. Steuerungsalgorithmen von interaktiven Prozessen sind eingebettet in komplexe Interface-Umwelten und nicht mehr als einfache Problemlöser zu verstehen, sondern nur als Gesamtsysteme mit weitaus umfassenderen Qualitäten. Bei allen diesen Anwendungen zählt weniger das Errechnete als vielmehr das Erlebte. Diese Systeme können nicht mehr vollständig auf dem Papier nach dem klassischen Problemlösungsansatz entwickelt werden, sondern nur interaktiv, in iterativen Zyklen, im direkten Zusammenspiel mit dem Computer. Mit anderen Worten: die Entwicklung solcher Algorithmen setzt voraus, daß uns bereits eine interaktive Maschine zur Verfügung steht, auf deren Basis wir in iterativen Test- und Codierungsphasen zu neuen Anwendungen kommen. Werden neben Interfaces weitere Mechanismen, z.B. die später betrachteten Code-Reflexionen und Code-Heterarchien eingesetzt, dann resultieren Anwendungen, bei denen das Unbeabsichtigte eine wesentliche Rolle spielt. Da das Verhalten dieser so entwickelten Algorithmen nicht durch ausschließliche Analyse der Codes vorhersagbar ist, treten das reale Experiment und die Spekulation, sowie leistungsfähige Strategien des Testens und Korrigierens ins Zentrum der Systementwicklung. Bevor wir uns die Mechanismen, die zur Unbeherrschbarkeit und Unvorhersagbarkeit solcher Systeme führen, zuwenden, wollen wir zuvor noch einmal einen Blick auf die klassische Ausgangssituation der Informatik werfen.

Klassische Algorithmentheorie

Die klassische Theorie des Computers enthält keine Zukunftsdimension. Algorithmen sind endliche schrittweise Verfahren zur Berechnung gesuchter aus gegebenen Größen. Jeder einzelne Schritt besteht aus einer Anzahl ausführbarer deterministischer Operationen. Nach Ausführung eines Schrittes steht eindeutig fest, welcher Schritt als nächstes ausgeführt wird. Alle Berechnung folgt damit einer (endlichen) Kausalkette. Diesem Denken fehlt die Dimension des Werdens, die Gegenwart enthält bereits vollständig alle Zukunft. Die Verbindung zwischen Vergangenheit und Zukunft ist lediglich ein getakteter und gleichmäßig voranschreitender Maschinenzustand. Wird ein Zustand notiert und die Maschine später auf diesen Zustand zurückgesetzt, ist alles, was dazwischen passiert, vergessen, für die Maschine ist es nie passiert. Damit soll keinesfalls gesagt sein, daß auf der Basis des schrittweisen Vorangehens nicht Unvorhersehbares und Überraschendes realisierbar wäre – ganz im Gegenteil – aber ein Begriff des »Werdens« ist bei der Behandlung klassischer algorithmischer Fragestellungen schlicht nicht erforderlich. Es wird von vielen sogar vehement bestritten, daß durch algorithmische Prozesse überhaupt Neues entstehen kann.

Computer waren zunächst als Hilfsmittel zur Entlastung des Menschen von repetitiven und mechanischen Rechenarbeiten gedacht. Große Mengen sich wiederholender Berechnungen sollten zuverlässig, schnell und effizient durchgeführt werden. Die Probleme, die mit dem klassischen Algorithmen-Begriff erfaßt werden sollten, zeichneten sich dadurch aus, daß das zu bearbeitende Problem klar beschrieben werden konnte und jeweils eindeutige Kriterien existierten, wann eine Lösung für ein Problem vorliegt. In der Theorie der Berechenbarkeit werden insbesondere zwei konkrete Formen von Problemlösungsprozessen unterschieden:

– die Berechnung einer Funktion,
– die Entscheidung über die Zugehörigkeit eines Elements zu einer Menge.

Da die Entscheidungsprobleme ebenfalls leicht als bedingte Funktionen darzustellen sind, können Fragen der Berechenbarkeit immer als Funktionen beschrieben werden. Die Theorie der Berechenbarkeit ist damit im Grunde ein bescheidener Ansatz. Sie fragt lediglich danach, ob es Funktionen gibt, die wir zwar exakt spezifizieren können, die aber dennoch von keinem schrittweisen endlichen Verfahren gelöst werden können. Die Entscheidung derartiger Fragen setzt die strenge Formalisierung des Begriffs »effektives Verfahren« (bzw. Algorithmus) voraus. Ein intuitiver Algorithmen-Begriff reicht aus, solange wir nur zeigen wollen, daß ein bestimmtes Problem algorithmisch lösbar ist. Die hinreichend detaillierte Skizzierung eines Verfahrens wird in den meisten Fällen genügen, um uns zu überzeugen, daß eine algorithmische Lösung für das Problem existiert. Um allerdings zu zeigen, daß gewisse Probleme prinzipiell nicht algorithmisch lösbar sind, müssen wir einem strikten Formalismus zugrunde legen. In diesen Fällen soll ja eine Aussage über alle denkbaren Verfahren getroffen werden.

Das unverrückbare Fundament der Informatik – die Theorie der Berechenbarkeit, die auf der Basis dieser Fragestellungen entwickelt wurde – hat sich seit Mitte der 1960er Jahre kaum mehr gewandelt. Zu den wichtigsten Ergebnissen der Berechenbarkeitstheorie gehören nicht nur verschiedene Formalismen zur Präzisierung des Algorithmenbegriffs, sondern auch allgemeine Aussagen über die Vollständigkeit, Widerspruchsfreiheit und Entscheidbarkeit formaler Systeme. Heutige Programmiersprachen und ihre Compiler basieren ebenso auf den Ergebnissen der Berechenbarkeitstheorie wie moderne Grundlagenforschungen auf dem Gebiet der Komplexitätstheorie. Zu den zentralen Konzepten gehört unter anderem die universelle Turingmaschine, eine Maschine, die dem Begriff des Algorithmus äquivalent ist und ihn gleichzeitig definiert. In Turings berühmten Artikel »On Computable Numbers, with an Application to the Entscheidungsproblem«[3] aus dem Jahre 1936 wird erstmals das Konzept der universellen Rechenmaschine eingeführt. Turing selbst sagt über seine Maschine:

Die Bedeutung der universalen Maschine ist klar. Wir brauchen nicht unzählige unterschiedliche Maschinen für unterschiedliche Aufgaben. Eine einzige wird genügen. Das technische Problem der Herstellung verschiedener Maschinen für verschiedene Zwecke ist ersetzt durch die Schreibarbeit, die Universalmaschine für diese Aufgabe zu programmieren.[4]

Aufgrund ihrer Programmierbarkeit kann diese Maschine alles berechnen, was überhaupt berechenbar ist, ohne für neue Aufgaben noch einmal in ihre innere Struktur eingreifen zu müssen. Jedes Programm ist die Beschreibung einer speziellen Maschine und die Universalmaschine ist in der Lage, die Beschreibung zu lesen und sich wie diese Maschine zu verhalten. Seit der realen Umsetzung dieses Prinzips – erstmals durch die Von-Neumann-Architektur – steht die Universalmaschine auch in der Praxis zur Verfügung. Alle berechenbaren und entscheidbaren Fragen können somit von ein und derselben Maschine berechnet und entschieden werden. Welche Zukünfte also soll der Computer vor diesem Hintergrund noch haben? Ist der Rest nicht pure Praxis, d.h. ewige, variierte Repetition der Grundprinzipien und vom Standpunkt der Theorie aus relativ langweilige Ausweitung der Anwendungsfelder? Wir betrachten die programmierbare universelle Maschine zunächst noch etwas eingehender. Die kurze Vergegenwärtigung ihrer elementaren Strukturen und ihrer Entwicklungsgeschichte wird es in der Folge erlauben, das Neue an den zukünftigen Praktiken der Anwendungsentwicklung besser zu verstehen.

3. Alan M. Turing, »On Computable Numbers, with an Application to the Entscheidungsproblem«, in: *Proc. London Math. Soc.* 2,42(1936), S. 230-267, hier zit. nach: Alan M. Turing, *Intelligence Service*, a.a.O., S. 17-60.

4. »Intelligent Machinery«, a.a.O., S. 88.

Die programmierbare Maschine – Ein historischer Abriß

Die programmierbare Universalmaschine wurde möglich durch die Zusammenführung dreier weitestgehend unabhängig verfolgter Forschungs- und Entwicklungslinien: der Formalisierung in der Mathematik, der Mechanisierung durch das Ingenieurwesen und der rationalistischen Tradition in der Philosophie und den modernen Wissenschaften generell.

Formalisierung, Mathematik
Einen naheliegenden Einstiegspunkt in der Geschichte der Formalisierung markiert der persisch-arabische Mathematiker Abu Dscha'far Muhammed ibn Musa Al-Khwarazmi (Muhammed, Vater des Dscha'far, Sohn des Musa, der Chorasmier), der um das Jahr 820 das Lehr- und Rechenbuch *Über die Indischen Zahlen* veröffentlicht. Dieses Buch, in dem die Grundrechenarten in unserem heute gebräuchlichen Zehnersystem beschrieben werden, leitet den Sieg des Ziffernrechnens über das bis dahin verbreitete Abacusrechnen ein. Die Rechenmeister und ihre Rechenbücher verdrängten zwar in der Folgezeit nach und nach die Rechenbretter, aber erst im 16. Jahrhundert setzen sich die Algorithmiker endgültig gegen die Abacisten durch. Doch das »Rechnen mit den Federn« anstatt »auf den Linien« (Abacus), ist nur der erste Schritt auf dem Weg zur Formalisierung der gesamten Mathematik. Die »Coß«, das Verbindungsglied zwischen bloßer Rechenkunst und dem Umgang mit Gleichungen und Unbekannten, ist ein weiterer. Auf dieser frühen Stufe der Algebraisierung im 15. Jahrhundert werden bereits erste Symbole und Kunstwörter verwendet. Die Verfasser entsprechender mathematischer Schriften werden Cossisten genannt. Das Wort »Coß« ist hergeleitet vom italienischen cosa (Sache) und steht für die unbekannte Größe in Gleichungen. Die Einführung der modernen Formelschreibweise wird wesentlich dem Franzosen Francois Viète (1540–1603) zugeschrieben. Mit der von ihm eingeführten Schreibweise lassen sich unter anderem erstmals Regeln für das Auflösen von Gleichungen allgemeingültig formulieren. Damit sind die uns heute geläufigen formalen Darstellungen in die Mathematik eingeführt und stehen im Prinzip auch zur Beschreibung von Algorithmen zur Verfügung. Vollendet wird das Programm der Formalisierung allerdings erst Ende des 19. und in der ersten Hälfte 20. Jahrhunderts. Der Göttinger Mathematiker David Hilbert (1862–1943) versuchte, das gesamte Gebäude der Mathematik auf einer vollständigen und widerspruchsfreien Formalisierung aufzubauen. Die mathematischen Symbole wurden hierbei jeder Bedeutung entledigt, die gesamte Mathematik sollte auf ein regelbasiertes Spiel syntaktischer Zeichen reduziert werden. Berühmt ist Hilberts Ausspruch: »Man muß jederzeit an Stelle von ›Punkten‹, ›Geraden‹, ›Ebenen‹, ›Tische‹, ›Stühle‹, ›Bierseidel‹ sagen können.« Die Hoffnung Hilberts hat sich – wie wir heute wissen – nicht erfüllt. So zeigte Kurt Gödel im Jahre 1931, daß die Widerspruchsfreiheit der Arithmetik innerhalb der Arithmetik nicht nachgewiesen werden kann. Alonso Church beweist 1936, daß es keinen Algorithmus gibt, der für jede mathematische Aussage entscheidet, ob sie wahr oder falsch ist. Alan M. Turing vollendet schließlich die Niederlage Hilberts mit

der Publikation seines bereits erwähnten Artikels »On Computable Numbers, with an Application to the Entscheidungsproblem«, in dem er zeigt, daß Hilberts Entscheidungsproblem keine Lösung haben kann. Da in diesem Beitrag gleichzeitig erstmals die universelle Turingmaschine definiert wird, markiert der Artikel nicht nur den Abschuß einer bedeutenden mathematischen Periode, sondern gleichzeitig den Beginn einer neuen Ära: des Computerzeitalters.

Mechanisierung, Ingenieurwesen

Die Turingmaschine ist eine Papiermaschine. Turings Ziel war die Lösung grundlegender formaler Fragen der Mathematik und sein Konzept der Maschine eine Möglichkeit, diese Grundprobleme zu behandeln. Einer der wichtigsten Schritte auf dem Weg zur gebauten Universellen Maschine ist die Idee der externen Programmierung. Der erste Rechner, der durch ein Programm, d.h. einen einzulesenden Code gesteuert werden sollte, die »Analytical Engine« von Charles Babbage (1791-1871), wurde in den 1830er Jahren in England konzipiert, aber zu Babbages Lebzeiten nie gebaut. Die genaueste und umfassendste Beschreibung dieser Maschine stammt von dem italienischen Militäringenieur und späteren Premierminister von Italien, L. F. Menabrea. Seine Veröffentlichung wird später von Ada Augusta Lovelace, der berühmten Freundin von Babbage und Tochter von Lord Byron, ins Englische übersetzt und mit zahlreichen Kommentaren und Anmerkungen ergänzt. Über diesen Umweg verfügen wir heute über eine detaillierte und relativ präzise Beschreibung der Funktionsweise der Maschine[5]. Das Programm der Analytischen Maschine sollte auf einer Reihe von Lochkarten gespeichert werden. Diese Idee hatte Babbage von Jacquards Webstühlen übernommen, bei denen die zu webenden Stoffmuster über Lochkarten kontrolliert wurden. Wie weit Babbages Überlegungen zur Programmierung wirklich gingen, konnte bisher nicht endgültig geklärt werden. Es gibt allerdings eine Passage in einem seiner Notizbücher, die eindeutig auf Überlegungen zur laufzeitabhängigen Steuerung von Prozessen hinweist: [6]

This day I had for the first time a general but very indistinct conception of the possibility of making an engine work out algebraic developments. I mean without any reference to the value of the letters. My notion is that as the cards (Jacquards) of the Calc. Engine direct a series of operations and then recommence with the first so it might perhaps be possible to cause the same cards to punch others equivalent to any given number of repetitions. But their holes might perhaps be small pieces of formulae previously made by the first card.

Hier stanzen Lochkarten andere Lochkarten, d.h. wir haben bereits das erste Beispiel für Codes, die andere Codes schreiben. Babbages weitsichtige Arbeiten gerieten vollkommen in Vergessenheit. Es dauerte noch einmal 100 Jahre, bis 1936 der deutsche Ingenieur Konrad Zuse (1910-1995) in der elterlichen Wohnung in Berlin den Z1 baut, den ersten mechanischen programmgesteuerten

5. Siehe dazu auch: Bernhard Dotzler (Hg.), *Babbages Rechen-Automate*, Wien 1996.

6. Zit. nach: Brian Randell, »Stored Program Electronic Computers«, in: *The Origin of Digital Computers*, Hg. B. Randell, New York 1982, S. 375-381.

Rechner. Aber auch Zuse ist als Konstrukteur des universellen Rechners umstritten, seine Maschinen bis hin zur Z4 sind schleifengesteuert und erlauben nicht die Berechnung allgemein rekursiver Funktionen.[7] Jedoch gehen die Ideen in seinen Aufzeichnungen und Notizen weiter. Bereits 1938 erwähnt er »lebendige Rechenpläne«, bei denen im Unterschied zu seinen bisherigen »starren Rechenplänen« nun die errechneten Daten und die Ausgangsdaten auch Einfluß auf den Ablauf der Berechnung haben. Die Realisierung der ersten Universalmaschine, die zumindest im Prinzip der universellen Turing-Maschine äquivalent ist, bleibt Eckert, Mauchly, Goldstine und von Neumann vorbehalten. Der durch von Neumann 1945 im *First Draft of a Report on the EDVAC* beschriebene Entwurf,[8] der später unter dem Namen von-Neumann-Architektur bekannt wird, ist die erste speicherprogrammierte Maschine, die der Papiermaschine Turings wirklich gleichmächtig ist. Interessant ist der Hinweis von F. L. Bauer, daß die Erfinder des Programmspeicherkonzeptes die Universalität ihres Ansatzes offensichtlich selbst nicht erkannten.[9] Externe Programmierbarkeit alleine reicht nicht aus, um die Universalität einer Maschine sicherzustellen. Es muß auf der Ebene der Maschinenbefehle ein weiterer Mechanismus – nämlich die bedingte Ausführung von Operationen abhängig von Zwischenergebnissen – realisiert werden, damit jede berechenbare Funktion auch wirklich von der Maschine berechnet werden kann. Da wir heute aber nicht mehr wie früher direkt in Maschinensprache programmieren, muß dieser Mechanismus auch in höheren Programmiersprachen in irgend einer Weise repräsentiert sein, ansonsten würde die prinzipielle Universalität auf der Ebene der Maschinenbefehle auf Softwareebene wieder verloren gehen. Neben dem von F. L. Bauer beschriebenen hardwarenahen Prinzip der ergebnisabhängigen Berechnung von Befehlsadressen, hat die theoretische Informatik eine Reihe weiterer Kontrollmechanismen untersucht und deren Äquivalenz im Hinblick auf Universalität nachgewiesen. In der ein oder anderen Form sind diese Kontrollstrukturen in allen höheren Programmiersprachen realisiert.

Rationalistisches Denken, Philosophie

Galileo Galilei (1564-1632) gilt als einer der frühesten Begründer der rationalistischen Tradition. Mit seiner Verbindung aus analytischer und synthetischer Methode begründet er einen neuen Ansatz in der Wissenschaft. Überall seien die verwickelten Erscheinungen der sinnlichen Beobachtungen über die »*metodo risolutivo*« in ihre einfachsten Komponenten zu zerlegen, um dann aus ihnen die Vorgänge über die »*metodo compositivo*«, die synthetische Methode, zu erklären. Als weitere Schlüsselfigur des rationalistischen Denkens gilt René Descartes (1596-1650). Er legt in seinem kleinen Buch *Le Discours de la Méthode* die vier Grundsätze rationalen Denkens nieder:[10]

7. Siehe Friedrich L. Bauer, »Konrad Zuse – Fakten und Legenden«, in: *Die Rechenmaschinen von Konrad Zuse*, Hg. R. Rojas, S. 5-22, Berlin 1998.
8. Nachdruck in: Randell a.a.O., S. 383-392.
9. Bauer, a.a.O.

1. Die erste besagte, niemals eine Sache als wahr anzuerkennen, von der ich nicht evidentermaßen erkenne, daß sie wahr ist: d.h. Übereilung und Vorurteile sorgfältig zu vermeiden und über nichts zu urteilen, was sich meinem Denken nicht so klar und deutlich darstellte, daß ich keinen Anlaß hätte, daran zu zweifeln.

2. Die zweite, jedes Problem, das ich untersuchen würde, in so viele Teile zu teilen, wie es angeht und es nötig ist, um es leichter zu lösen.

3. Die dritte, in der gehörigen Ordnung zu denken, d.h. mit den einfachsten und am leichtesten zu durchschauenden Dingen zu beginnen, um so nach und nach, gleichsam über Stufen, bis zur Erkenntnis der zusammengesetztesten aufzusteigen, ja selbst in Dinge Ordnung zu bringen, die natürlicherweise nicht aufeinander folgen.

4. Die letzte, überall so vollständige Aufzählungen und so allgemeine Übersichten aufzustellen, daß ich versichert wäre, nichts zu vergessen.

Die Form des Denkens, die hier erstmals formuliert wird, zieht sich in der Folge durch die gesamte abendländische Philosophie und findet in der Kybernetik, die Wissenschaft und Technik vereint, einen vorläufigen Abschuß. In moderner Fassung lauten die Handlungsanweisungen rationalistischen Denkens wie folgt:[11]

1. Beschreiben Sie die Situation in der Begrifflichkeit identifizierbarer Gegenstände mit wohldefinierten Eigenschaften

2. Suchen Sie dann nach allgemeingültigen Regeln, die sich auf Situationen in der Begrifflichkeit solcher Gegenstände und Eigenschaften anwenden lassen.

3. Wenden Sie schließlich diese Regeln logisch auf die betreffende Situation an und leiten Sie die nächsten, notwendigen Schritte daraus ab.

Algorithmische Problemlösungen stehen in der Tradition rationalistischer Denkformen. Genauso gehen wir vor, wenn wir Computerprogramme schreiben. Die rationale Vorgehensweise ist das leitende Paradigma bei jeder Entwicklung technischer Systeme auf der Ebene formaler Beschreibungen, damit auch beim Bau des Computers selbst und bei seiner Programmierung.

Formalisierung, Mechanisierung und Rationalismus gehen in der programmierbaren Maschine eine mächtige Verbindung ein. Am Ende dieser ersten Entwicklungsphase des Computers, die noch in der ersten Hälfte des 20. Jahrhunderts abgeschlossen wird, steht eine universelle Maschine, die nicht nur als theoretisches Konzept – als Papiermaschine – vorliegt, sondern als gebaute und funktionsfähige Technik. Diese erste Maschine ist aber lediglich eine Rechenmaschine, sie erhält als Eingaben Zeichen und Ziffern, manipuliert diese auf der Basis sequentieller Verarbeitungsschritte und gibt wieder Zeichen und Ziffern aus. Ihre Mächtigkeit ergibt sich aus der Trennung von Hardware und Software. Die Hardware ist der starre Mechanismus, der eine fest vorgegebene, relativ geringe Zahl von Instruktionen verarbeiten kann, sie stellt das Substrat und den Handlungsrahmen. Erst die einzelnen Programme legen fest, welche Funktion die Maschine tatsächlich ausführt. Mechanismus und Formalismus fallen bei der

10. Zit. nach René Descartes, *Von der Methode des richtigen Vernunftgebrauchs und der wissenschaftlichen Forschung*, Hamburg 1960. S. 15.

11. Terry Winograd/Fernando Flores, *Erkenntnis Maschinen Verstehen*, Berlin 1989, S. 37.

universellen Rechenmaschine in Eins. Der Widerstreit des Mittelalters zwischen Abacisten und Algorithmikern löst sich auf. Das »Rechnen mit den Federn« (mit Papier und Bleistift) und das »Rechnen auf den Linien« (mit dem Abacus) sind hier nicht länger Gegensatz, sie gehen eine neue Verbindung ein. Programme sind formale Symbolsysteme, die Mechanismen beschreiben. Sie sind einerseits lineare Zeichenketten, die aus einem vorgegebenen Alphabet von Zeichen aufgebaut sind, gleichzeitig sind sie aber Instruktionen für den Mechanismus der Maschine. Die tatsächliche Mächtigkeit dieses Konstruktes wurde aber erst in der Folge durch die langsame Entstehung einer Softwarekultur deutlich. Da nun sowohl die Größen, die durch die Maschinen verarbeitet werden, als auch die Maschinen selbst nichts anderes als Zeichenketten sind, werden vielfältige neue Konstellationen zwischen Zeichenketten möglich, aus der die heutige Praxis ihre eigentliche Vitalität bezieht. Zeichenketten können nämlich Zeichenketten erzeugen, manipulieren und verarbeiten, die selbst wieder Maschinenbeschreibungen sind, usw.

Die neue hybride Maschine

Die klassische programmierbare Universalmaschine ist von ihrer Idee und ihrem Wesen her eine Rechenmaschine. Die neue Maschine, die sich seit der zweiten Hälfte des 20. Jahrhunderts herausbildet, ist dagegen eine hybride Maschine. Sie ist hybrid in verschiedener Hinsicht. Zum ersten verwandeln die Interfaces den ursprünglichen Rechenknecht Computer wahlweise in eine Multimediamaschine, einen handelnden Roboter oder einen begehbaren virtuellen Raum. Die Anwendungen sind damit längst keine reinen Rechenanwendungen mehr, sondern erstrecken sich auf alle Bereiche des wirtschaftlichen, wissenschaftlichen und des kulturellen Alltags. Der Computer und seine spezifischen Anwendungen sind nicht mehr losgelöst von den Interfaces zu verstehen, sondern bestehen immer aus dem digitalen Rechner und dem analogen Interface, die Maschine wird zum raumgreifenden Apparat. Der rationale, zeichenbasierte Zugang zur Maschine tritt dabei zunehmend in den Hintergrund, und Beschäftigungen mit visuellen und akustischen Wahrnehmungsformen werden wichtiger. In diesem Zuge wird schmerzlich deutlich, daß Informatiker in der Regel nichts von Wahrnehmung verstehen. Die neue Maschine ist zum zweiten hybrid, weil der Rechner nicht mehr als isolierte Einheit in Erscheinung tritt, sondern nur noch als Knoten in komplexen Netzwerken. Das Internet, Funknetze oder die neuen Handynetze sind nur einige Beispiele für die Vielzahl heterogener, untereinander wieder in Verbindung stehender Netze aus rechnenden und informationsaustauschenden Grundeinheiten. Zum dritten ist die neue Maschine hybrid, weil die neuen Arbeitsprozesse am Computer grundsätzlich symbiotisch sind, d.h. das hybride Gesamtsystem aus Mensch und Maschine ist in der Lage, Aufgabenstellungen zu bearbeiten und neue Systeme zu entwickeln, die der Mensch alleine, ohne dieses Werkzeug, überhaupt nicht zustande bringen könnte.[12] Bei hybriden Maschinen verläuft die Praxis und die Entwicklung neuer Applikatio-

nen in einem neuartigen Wechselspiel von Mensch und Maschine und es erschließen sich damit neue Anwendungsfelder.

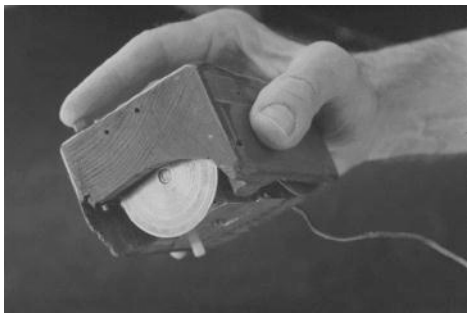
Ein wesentliches Kennzeichen der hybriden Maschine sind flottierende Codes, d.h. Programme, die nicht mehr an einen Ort gebunden sind, sondern sich frei durch die Netze bewegen und Hardware nur noch als wählbare Umgebung betrachten. Flottierende Codes sind aber auch Codes, die kontextabhängig als passive Daten behandelt werden, oder selbst aktive Prozesse sind, die also in einem Moment noch von einem anderen Programm als Datum manipuliert werden, im nächsten Moment umschlagen und selbst zum Prozeß werden und andere Codes bearbeiten. Ein aktuelles Leitbild der Informatik, »Pervasive Computing« – die vollständige Durchdringung der Alltagwelt mit vernetzten, »intelligenten« Gegenständen – macht den radikalen Anspruch hybrider Systeme deutlich. Die Vorstellung allgegenwärtiger, miniaturisierter, vernetzter und umgebungssensitiver Mikrochips in Kleidungsstücken, Brillen, Haushaltsgegenständen und nicht zuletzt direkt dem menschlichen Körper, läßt sofort erkennen, daß diese Technologien strikt dem Prinzip Verantwortung zu unterstellen sind. Der Grundgedanke einer experimentellen und spekulativen Informatik besteht nun allerdings – wie wir später noch sehen werden – gerade darin, daß derartige Systeme prinzipiell nicht beherrschbar und in ihrem Verhalten vorhersagbar sind, im gleichen Sinne, wie das Verhalten chaotischer Systeme in der Physik prinzipiell nicht über einen längeren Zeitraum vorhersagbar ist. Der Begriff »hybrid« soll damit nicht nur Sinne der lateinischen Übersetzung von »gemischt«, bzw. »von zweierlei Herkunft« verstanden werden, sondern durchaus auch im griechischen Ursprung von »vermessen« und »überheblich«. Die Dominanten der neuen hybriden Maschine sind nicht mehr Ziffern und Buchstaben, die man auch auf einem Blatt Papier ausrechnen könnte, sondern Interfaces, Code-Reflexionen und heterarchische Netze. Wir wollen diese Komponenten genauer betrachten.

Interfaces

Am 9. Dezember 1968 präsentieren Douglas C. Engelbart und eine Gruppe junger Wissenschaftler im Rahmen der »Fall Joint Computer Conference« ein vollkommen neuartiges o*N*-*L*ine *S*ystem (NLS). Engelbarts dreißigminütige Demonstration ist heute als »the mother of all demos« bekannt. Die von Douglas C. Engelbart und seiner Gruppe eingeführten Interaktionsparadigmen – das Positionieren, Zeigen, Auswählen und Manipulieren von Bildschirmobjekten – gehören inzwischen zum festen Bestandteil jeder Computeranwendung und haben sich als neue Kulturtechniken etabliert. Die erste Maus, die Engelbart und sein Team im Rahmen von NLS entwickelten, hatte zwei Räder auf der Unterseite, die durch die Bewegung der Maus über den Schreibtisch, Informationen über die x-y Koordinaten liefern. Die Abbildung zeigt das Gehäuse den ersten

12. Siehe dazu weiter unten: Douglas C. Engelbart, *Augmenting Human Intellect: A Conceptual Framework*, Summary Report AFOSR-3223, SRI Project 3578 for Air Force Office of Scientific Research, Stanford Research Institute, Menlo Park, Ca., October 1962.

Douglas Engelbart, Die erste Maus



Prototypen, das noch aus Holz war. Die Beschädigung an der Seite gibt den Blick auf eines der beiden Räder frei.

Die Maus ist allerdings nur eine von mehreren Neuerungen die im Rahmen des NLS Systems verwirklicht wurden, an dem die 17-köpfige Gruppe um Engelbart im »Augmentation Research Center« des Stanford Research Institutes in Menlo Park seit 1962 gearbeitet hatte. In der legendären Präsentation von 1968 im Convention Center in San Francisco werden unter anderem die Elemente moderner Textverarbeitung in einer 90-minütigen Live-Vorführung gezeigt. Von einer leeren »Seite« ausgehend, führt Engelbart die wesentlichen Elemente heutiger Textprozessoren vor, Texteingabe, cut & copy-Funktionen und das Anlegen von Dateien und Metadaten. Eine weitere bahnbrechende Innovation des NLS Systems ist die Realisierung einer kollaborativen Umgebung, in der zwei Benutzer an verschiedenen Orten über ein Netzwerk mit Audio- und Video-Verbindungen kommunizieren und dabei einen gemeinsamen Bildschirm (*shared screen*) manipulieren. Im Zentrum der Engelbart'schen Vision stand der Computer als Medium zur Erweiterung menschlicher Problemlösungskompetenz. Seine Forschungen zielten auf die Entwicklung neuer Werkzeuge und Interfaces für die Unterstützung des Benutzers bei der Bearbeitung dringender und schwieriger Probleme. Seine Hypothese war, daß durch die Verwendung geeigneter interaktiver Werkzeuge die Problemlösungskompetenz des Menschen enorm gesteigert werden kann. Dies ist der Grundgedanke jedes symbiotischen Systems: Das Gesamtsystem ist nicht nur quantitativ, sondern auch qualitativ leistungsfähiger als die einzelnen Teile. Bei der klassischen Turing'schen Maschine war das nicht gegeben, der Mensch konnte den Rechner noch vollständig ersetzen. Wir erinnern uns: »Ein Mensch, ausgestattet mit Papier, Bleistift und Radiergummi sowie strikter Disziplin unterworfen, ist in der Tat eine Universalmaschine.«

Ivan Sutherland, heute vor allem bekannt als Entwickler des *Sketchpad Systems*, ist ein weiterer Pionier des neuen erweiterten Computerdenkens. Er beschreibt 1965 in seinem Aufsatz »The Ultimate Display« Hard- und Software-Komponenten, die nach seiner Auffassung eines Tages zu einer computerbasierten »Reality Engine« zusammen gefügt würden.[13]

The ultimate display would, of course, be a room within which the computer can control the existence of matter. A chair displayed in such a room would be good enough to sit in.

Ivan Sutherland, *Head Mounted Display*

Malcolm Fowlers sich selbst nagelnder Hammer

Handcuffs displayed in such a room would be confining, and a bullet displayed in such room would be fatal. With appropriate programming such a display could literally be the Wonderland into which Alice walked.

Einen entscheidenden Schritt in Richtung der Realisierung seiner Vision unternimmt Sutherland selbst durch die Entwicklung eines »Head Mounted Display« (HMD), das er als Prototypen 1970 an der Universität Utah vorstellt.

Nach Dieter Daniels[14] erweist sich die Erkenntnis, daß jede Rezeption eines Kunstwerks die aktive Partizipation des Betrachters erfordert, als Leitmotiv der gesamten Moderne. Kein Kunstwerk kann demnach einem Betrachter genau das mitteilen, was der Künstler beabsichtigt. Marcel Duchamp nennt dies den »persönlichen Kunst-Koeffizienten«, der das Verhältnis zwischen dem Unausgedrückten-aber-Beabsichtigten und dem Unabsichtlich-Ausgedrückten beschreibt. In jeder ästhetischen Erfahrung kommt dem Betrachter selbst eine konstitutive Rolle zu. Wir können diese Einsicht aber auch auf unsere Computerinterfaces und die hybride Maschine übertragen, indem wir zunächst fragen, was Interfaces hier leisten. Eingabeseitig findet in Interface-Prozessen eine semantische Determinierung und syntaktische Codierung statt. Aus der Vielfalt der Umweltereignisse werden ganz bestimmte ausgewählt und mit Bedeutung belegt. Diese bedeutsamen Ereignisse werden numerisch codiert und können dann digital verarbeitet werden. Nehmen wir als einfaches Beispiel ein Thermo-

13. Vgl. Ivan Sutherland, »The Ultimate Display«, in: *Proceedings of the International Federation for Information Processing Congress*, 1965, S. 506-508.

14. Vgl. das Kapitel »Strategien der Interaktivität«, in: Dieter Daniels, *Vom Ready-Made zum Cyberspace, Kunst/Medien Interferenzen*, Stuttgart 2002 [http://www.hgb-leipzig.de/daniels/vom-readymade-zum-cyberspace/strategien_der_interaktivitaet.html].

meter. Aus der Vielfalt des Umweltgeschehens wird hier ein einziger numerischer Wert isoliert, die Temperatur am Meßfühler. In jedem Interface findet auf diese oder ähnliche Weise eine semantische Reduktion der komplexen Umwelt statt. Bei komplexeren Interfaces, etwa einer Kamera, können aber im Prozeß der Reduktion auch bereits wieder Nebeneffekte ins Spiel kommen. Das mit einer Kamera erzeugte Bild enthält niemals nur exakt die Informationen, die vom Kameramann intendiert waren. Ausgabeseitig haben wir es mit einem umgekehrten Prozeß zu tun. Numerische Werte und Zeichen werden in Wahrnehmungen umgesetzt, hierbei werden ebenfalls unvermeidlich Nebeneffekte generiert. Durch die Augmentation der Zeichen und der numerischen Werte in den Ausgabe-Interfaces kommt ebenfalls Unbeabsichtigtes ins Spiel, und es ergeben sich neue semantische Potentiale. Winograd und Flores bezeichnen diese Phänomene auch als »unbeabsichtigte Repräsentation«.[15] So können zum Beispiel sehr regelmäßige und dynamische Muster entstehen, wenn Inhalte von Datenstrukturen graphisch dargestellt werden. Etwa nimmt der Betrachter Kreise auf dem Bildschirm wahr, obwohl weder ein explizites Konstruktionsprinzip für Kreise im Code zu finden ist, noch der Programmierer in irgendeiner Phase des Programmentwicklung an Kreise gedacht hat. Festzuhalten ist: Interfaces können eingabeseitig semantische Potentiale reduzieren und sowohl eingabe- als auch ausgabeseitig neue semantische Potentiale generieren. Die Echtzeit-Interaktion mit einem Computer auf der Basis von Bildern, Tönen, Schmerzempfindungen, etc., löst unvermeidlich sowohl physiologisch als auch kulturell determinierte Wahrnehmungsprozesse aus und ist damit ein vollkommen anderer Prozeß als das Input-/Output-Verhalten einer zeichenbasierten Rechenmaschine.

Code-Reflexionen

In den 1980er Jahren gab es unter dem Begriff »Computational Reflection« ernsthafte wissenschaftliche Bemühungen, reflexive, d.h. selbstbezügliche Konstrukte in höheren Programmiersprachen zur Verfügung zu stellen.[16] Unter »Computational Reflection« werden alle Aktivitäten zusammengefaßt, die ein System durchführt, wenn es Berechnungen über seine eigenen Berechnungen anstellt. Ohne dies explizit als »Reflection« zu bezeichnen, führen Computer schon immer nicht nur Berechnungen im jeweiligen Problem- und Aufgabenbereich durch, sondern auch selbstbezogene. Beispiele sind: Performance-Statistiken über die Auslastung des Rechners und seiner Ressourcen, Sammlung von Informationen über den Debugging-Prozeß bei Compilern, Selbstmodifikationen innerhalb von Lernverfahren oder Berechnungen darüber, welche Befehle innerhalb eines laufenden Programms als nächstes ausgeführt werden sollen. Algorithmische Reflexionen stehen damit nicht nur im Kern der Berechenbarkeitstheorie (Antinomien, Cantorscher Diagonalisierungsbeweis, Unentscheidbarkeit, Halteproblem, etc.) sondern auch im Kern jeder Lern- und Evolutions-

15. Vgl. Winograd/Flores, a.a.O., S. 155
16. Vgl. Pattie Maes/Daniele Nardi, *Meta-Level Architectures and Reflection*, Amsterdam 1988.

theorie. Wir erinnern uns zwar dunkel, daß einst selbstreferentielle Strukturen (Antinomien) am gesamten Gebäude der klassischen Logik gerüttelt haben, glauben aber, daß die Reparaturen erfolgreich waren. Wir verdrängen bisher erfolgreich, daß nur notdürftig geflickt wurde und die Wunde jederzeit wieder aufreißen kann. Zu den bedeutendsten Mahnern gehört Gotthard Günther, der eine transklassische Logik und Maschinentheorie einfordert.[17] M. C. Eschers *upstairs-downstairs*, oder Malcolm Fowlers sich selbst nagelnder Hammer sind bildliche Ausdrucksformen des Prinzips der Selbstbezüglichkeit. An diesen visuellen Umsetzungen sehen wir bereits deutlich, daß hier etwas Gefährliches vorgeht, das unser übliches Denken herausfordert. Auch bei selbstbezüglichen Algorithmen sind Münchhausen-Strategien am Werke. Wir schreiben Codes, die über den Entwickler und ihre eigene Beschreibung hinauswachsen und sich am eigenen Schopf aus dem Sumpf ziehen.

Eng verbunden mit den Problemen der algorithmischen Reflexion ist das Konzept der Meta-Architekturen, d.h. die Verwendung hierarchisch höher stehender Systeme, die in der Lage sind, Auskunft über untergeordnete Subsysteme zu geben. Flottierende Codes, die sich frei durch die Netze bewegen und auf nicht vorher festgelegter Hardware zur Ausführung kommen, basieren ebenfalls auf solchen Formen der Code-Reflexion und der Meta-Architekturen. Nicht nur Viren und Würmer machen sich in unterschiedlich komplexen Varianten dieses Prinzips zu nutze, sondern bereits jeder Webbrowser wendet eine einfache Variante des Prinzips an. Programme übertragen andere Programme und bringen diese am neuen Ort zur Ausführung. Während Würmer noch auf die selbständige Verbreitung in Netzwerken angewiesen sind und lediglich Rechenzeit stehlen, sind Viren in der Lage, fremden Code zu infizieren. Sie schreiben sich selbst in andere Codes ein und versklaven diese. Zu den sicher bekanntesten Beispielen von Code-Reflexionen und Meta-Architekturen gehören Lernverfahren und Evolutionsstrategien. Die verschiedenen Varianten von Evolutionsstrategien versuchen, evolutionäre Prinzipien auf der Basis von Algorithmen zu realisieren. Aus Sicht von Mathematikern, Informatikern und Ingenieuren ist Evolution nicht mehr als eine interessante Möglichkeit, leistungsstarke Optimierungsverfahren zu entwickeln. Es geht nicht darum, die biologische Evolution zu verstehen oder gar nachzubilden, sondern es sollen lediglich bestimmte Aspekte der Evolution wie Selektion, Rekombination und Mutation so weit modelliert werden, daß sie auf Computern simuliert werden können und damit zur Lösung schwieriger Optimierungsprobleme beitragen können. Ein wichtiger Zweig Evolutionärer Algorithmen ist die von John Koza[18] entwickelte *Genetische Programmierung*. Die entscheidende Differenz zwischen den Genetischen Algorithmen und Genetischer Programmierung liegt in der Codierung der Individuen. Während Genetische Algorithmen die Individuen als passive Daten-

17. Vgl. Gotthard Günther, *Beiträge zur Grundlegung einer operationsfähigen Dialektik*, Band I-III, Hamburg 1976.
18. John R. Koza, *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, Cambridge, Mass. 1992.

strukturen – in der Regel gleichlange, binäre Zahlenketten – codieren, sind dagegen bei der Genetischen Programmierung die Individuen Computerprogramme variabler Länge und Struktur. Programme schreiben hier Programme.

Der Anthropologe und Kybernetiker Gregory Bateson, der den meisten als Urheber des *double-bind* bekannt ist, hat sich unter anderem mit den logischen Kategorien von Lernen und Kommunikation beschäftigt. Bateson überträgt dabei die logische Typenlehre Russells (die oben erwähnte Reparatur des klassischen Gebäudes der Logik), die dieser entwickelte, um Paradoxien in der formalen Logik zu vermeiden, auf den Begriff des Lernens. Die logische Typenlehre besagt, daß keine Menge in der formalen Logik Element ihrer selbst sein kann. Werden derartige Mengen nicht explizit verboten, können wir antinomische Strukturen wie die folgende konstruieren, die als Russells Antinomie in die Literatur eingegangen ist:

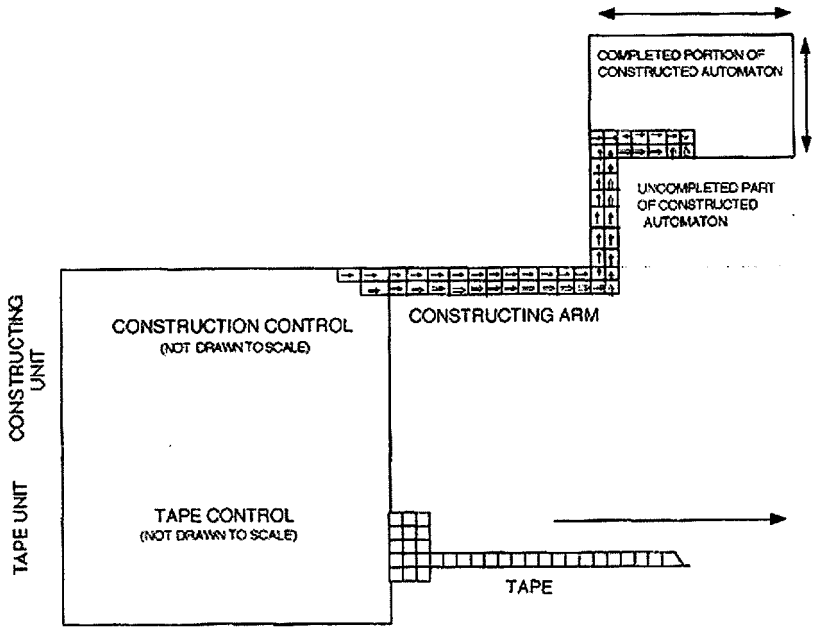
Sei R die Menge aller Mengen, die sich nicht selbst als Element enthalten, also
R := {X | X nicht Element X}.

Die Frage, ob die Menge R sich nun selbst als Element enthält oder nicht, führt zu einem unauflösbaren Widerspruch. Die logische Typenlehre geht deshalb von einer Folge von Hierarchieebenen von Mengen aus, die Bateson zur Klassifikation von Lernverfahren anwendet. Die Frage lautet für Bateson deshalb nicht, ob Maschinen lernen können, sondern welche Ebene des Lernens eine bestimmte Maschine erreicht. Bei Bateson werden ausführlich unterschiedliche Lernebenen vorgestellt, wobei die niedrigste Ebene als »Lernen null« bezeichnet wird.[19] Sein Schema kann so im Prinzip als unendliche Schichtung von Lernebenen fortgesetzt werden. Ein Lernverfahren der nächsthöheren Ebene beschreibt die Änderungen im Lernverfahren der darunterliegenden Schicht. Wenden wir die Bateson'sche Lernhierarchie auf die Lernmethoden und Algorithmen der Informatik an, stellen wir fest: Sowohl neuronale Netze, genetische Algorithmen als auch alle bekannten mathematischen Klassifikations- und Optimierungsverfahren gehören in die Kategorie »Lernen I«. Sie werden alle durch die Nachführung von Parametern realisiert. Lediglich »Genetische Programmierung« (auch »Emergente Evolutionäre Programmierung«) gehört zur Ebene »Lernen II«, da nicht nur die Parameter von Algorithmen optimiert werden, sondern Algorithmen neue Algorithmen entwickeln.

Als letztes Beispiel zur Code-Reflexion wollen wir selbstreproduzierende Systeme anführen. John von Neumann (1903-1957) arbeitete seit den späten 1940er Jahren an einer allgemeinen Theorie selbstreproduzierender Automaten.[20] Ausgangspunkt für von Neumanns Überlegungen zur Selbstreproduktion ist das hervorstechende Merkmal der Natur, daß komplizierte Organismen sich selbst fortpflanzen. Auf den ersten Blick erscheint diese Fähigkeit wie eine Antinomie, ein »circulus vitiosus« (Zirkelschluß, Teufelskreis), da wir erwarten würden, daß die Kompliziertheit von Systemen, die andere Systeme bauen, von

19. Vgl. Gregory Bateson, *Ökologie des Geistes*, Frankfurt/M. 1983. S. 362-399.
20. Vgl. John v. Neumann, *Theory of Self-Reproducing Automata*, edited and completed by Athur W. Burks, Urbana 1966.

Generation zu Generation abnimmt. Damit ein Automat A einen Automaten B bauen kann, muß er schließlich nicht nur eine vollständige Beschreibung von B enthalten, sondern auch noch verschiedene Vorrichtungen besitzen, um die Beschreibung interpretieren und die Bauarbeit ausführen zu können. Die plausibel erscheinende Annahme, die Kompliziertheit von sich selbst bauenden Automaten müßte von den Eltern zu den Nachkommen abnehmen, steht aber im Widerspruch zur offensichtlichen Selbsterhaltungsfähigkeit der Natur. Organismen pflanzen sich fort und produzieren neue Organismen, die mindestens genauso kompliziert sind wie sie selbst. Im Laufe langer Evolutionsperioden kann die Kompliziertheit, wie wir wissen, sogar zunehmen. Wie sehen die allgemeinen logischen Prinzipien aus, die Automaten zur Selbst-Fortpflanzung befähigen und sogar eine Steigerung an Kompliziertheit ermöglichen? Von Neumanns Schlußfolgerung lautet: Es gibt ein minimales Niveau von Kompliziertheit, auf dem Automaten möglich sind, die sich selbst fortpflanzen oder sogar höhere Gebilde bauen. Unterhalb dieses Niveaus sind Automaten degenerativ, d.h. Automaten, die andere Automaten bauen, sind nur in der Lage, einen weniger komplizierten zu erzeugen. Von Neumann beschreibt verschiedene Systeme zur Selbstreproduktion. Es ist sogar relativ einfach, Modelle anzugeben, die nicht nur in der Lage sind, sich selbst zu reproduzieren, sondern von Generation zu Generation auch noch die Leistungsfähigkeit zu steigern.



John von Neumann, Self-Reproducing Automata

Allen angeführten Beispielen ist gemein, daß ihre innere Konstruktion ein minimales Niveau von Kompliziertheit im Sinne von Neumanns voraussetzt.

Nur dann gelingen sie und können ihre Aufgabe erfüllen. Im Kern dieser Prinzipien stehen Code-Reflexionen. Codes interpretieren Codes und erzeugen nach bestimmten Vorschriften neue Codes. Oder, Codes kopieren andere Codes, bauen Variationen ein, übertragen diese neuen Codes an einen anderen Ort und bringen sie dort zur Ausführung. Diese wiederum machen sich ebenfalls sofort an die Arbeit, usw.

Heterarchische Netze

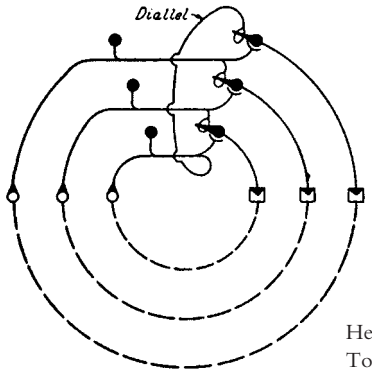
Computer und ihre Software gelten zweifellos als äußerst komplexe Systeme. Davon legt nicht zuletzt die anhaltende Softwarekrise Zeugnis ab, d.h. die verspätete Fertigstellung, eklatante Budgetüberschreitung und mangelnde Funktionsfähigkeit kommerzieller Software, bis hin zum kompletten Scheitern vieler Softwareentwicklungsprojekte. Das mächtigste Werkzeug, um Komplexität in den Griff zu bekommen und komplexe Systeme überhaupt zu beschreiben und verstehen zu können, sind Hierarchien. Der Nobelpreisträger Herbert Simon legt in einer Reihe von Arbeiten dar, daß viele komplexe Systeme eine zerlegbare hierarchische Struktur besitzen.[21] Die Architektur komplexer Systeme ist dabei immer eine Funktion ihrer Komponenten und deren hierarchischen Beziehungen untereinander. Der Wert solcher Systeme erwächst also aus dem Zusammenwirken der einzelnen Teile und nicht aus den Teilen selbst. Es gibt einige charakteristische Merkmale komplexer Systeme, die wir aus der Beobachtung natürlicher Systeme erkennen und für die Entwicklung künstlicher Systeme nutzbar machen können. Hierarchische Systeme setzen sich zum Beispiel typischerweise aus wenigen unterschiedlichen Arten von Subsystemen zusammen. Die Wahl und Anzahl unterschiedlicher Komponenten, aus denen sich das System zusammensetzt, ist allerdings relativ willkürlich und hängt von der Betrachtungsperspektive ab. Was für den einen bereits eine komplexe Komponente darstellt, kann für den anderen noch eine primitive Grundeinheit sein. Ein wichtiges Kriterium für die Zerlegung eines Systems in Subsysteme ist allerdings die Forderung, daß die Beziehungen innerhalb der Komponenten stärker sind als die Beziehungen zwischen den Komponenten. Für die Entwicklung von komplexen Systemen ist weiterhin die Beobachtung wichtig, daß sich Systeme, die funktionieren, mit Sicherheit aus einfacheren Systemen entwickelt haben, die ebenfalls funktionierten. Ein komplexes System, das von Grund auf neu zu entwerfen ist, kann nie zuverlässig arbeiten.

Bei der Entwicklung von Computern, sowohl der Hardware als auch der Software, kommen auf allen Ebenen und in allen Bereichen hierarchische Organisationsprinzipien zum Tragen. Es gibt letztlich niemanden mehr, der alle diese Ebenen der Maschine und alle ihre Subkomponenten im einzelnen verstehen würde, dennoch werden diese Maschinen gebaut und funktionieren – mehr oder weniger. Dies ist nur durch Hierarchisierung möglich. Während Hierarchien bewußt und extensiv in der Informatik eingesetzt werden, ist eine erweiterte Fassung dieses Organisationsprinzips bisher wenig untersucht, obwohl es in

21. Siehe z.B. Herbert Simon, *Die Wissenschaften von Künstlichen*, Wien 1994.

der Praxis bereits überall zu finden ist: die Heterarchie. Der Begriff Heterarchie geht im hier verwendeten Zusammenhang zurück auf Warren McCulloch. In seinem Artikel »A Heterachy of Values Determined by the Topology of Nervous Nets«[22] zeigt er die Notwendigkeit nicht-transitiver und heterarchicher Prinzipien, um neuronale Aktivitäten des Gehirns adäquat beschreiben zu können. Er macht explizit deutlich, daß bestimmte zielgerichtete Aktivitäten, bei denen zum Beispiel A den Vorzug vor B, B den Vorzug vor C, aber C den Vorzug vor A erhält, nicht in flachen hierarchischen Strukturen modelliert werden können. Solche Strukturen sind irreduzibel und können in der Ebene nicht ohne Diallele, eine die Fläche verlassende und übergreifende Struktur, dargestellt werden. McCulloch:[23]

Die einfachste Oberfläche, auf die sich dieses Netz topologisch (ohne Diallele) abbilden läßt, ist ein Torus. Zirkularitäten in der Präferenz zeigen nicht etwa Widersprüchlichkeiten an, sondern beweisen vielmehr Widerspruchsfreiheit einer höheren Ordnung, als sie unsere Philosophie sich je erträumen würde.



Heterarchie von Werten in der Topologie eines Nervennetzes

Hier wird ein einfacher Mechanismus beschrieben, der für die klassische und formale Logik in der Erscheinungsform von Zirkelschlüssen und Antinomien ein unüberwindbares Problem zu sein scheint. McCullochs Figur ist uns andererseits aus dem Alltag bekannt und wir gehen dort ganz selbstverständlich damit um. Das Spiel »Schere, Stein, Papier« basiert zum Beispiel auf einer derartigen nicht-transitiven Ordnung. Schere schneidet Papier, Papier wickelt Stein und Stein schlägt Schere. Diese einfache Form der Heterarchie ist inzwischen aber auch ein wichtiges Grundprinzip, um in Computerspielen »game balancing«, d.h. die Ausgewogenheit des Spiels zu gewährleisten. So gibt es in Kampfspielen beispielsweise nie Waffen, die stärker sind als alle anderen Waffen. In Zweikämpfen gibt es keine dominanten Schläge, sondern für jede Aktion gibt es mindestens eine andere, die diese neutralisiert. Computerspiele müssen also, damit sie interessant bleiben, heterarchisch organisiert sein. Zyklische Ordnungsstruk-

22. Die deutsche Fassung des Artikels ist zu finden in: Warren S. McCulloch, *Verkörperungen des Geistes*, Wien/New York 2000.
23. ebd.

Carlo Maria Mariana, *Die Hand unterwirft sich dem Intellekt*

turen sind die einfachste Form von Heterarchien, und natürlich können wir uns leicht erweiterte Formen der Heterarchie ausdenken. Unter Heterarchien – der Herrschaft des Anderen – wollen wir ganz allgemein sich selbst steuernde, verteilte Systeme verstehen, die über die Fähigkeit verfügen, spontan problemorientierte Ordnungs- und Kooperationsformen auszubilden und diese auch wieder aufzulösen. Während in hierarchischen Systemen die Ordnungsstrukturen starr bleiben, sind sie in heterarchischen labil und können situationsabhängig verändert werden. Die Heterarchie ist demnach kein Gegensatz zu Hierarchie, sondern dessen Erweiterung. So können zum Beispiel Heterarchien temporär Hierarchien ausbilden und diese nach einiger Zeit wieder auflösen. Wie bereits oben ausgeführt, ist ganz generell die Qualität komplexer Systeme immer eine Funktion seiner Komponenten und deren Beziehungen untereinander, sie entsteht durch das Zusammenwirken der einzelnen Teile und steckt nicht bereits in den Teilen selbst. Was bereits für hierarchische Organisationsformen gilt, gilt umso mehr für heterarchische Strukturen. Durch das Zusammenwirken der Teile erzeugt sich das System mit seinen spezifischen Qualitäten überhaupt erst selbst. Als bildhafte Darstellungen des heterarchischen Prinzips können M. C. Eschers *Drawing Hands* gelten, oder Carlo Maria Marianas Bild *Die Hand unterwirft sich dem Intellekt*. Heterarchische Systeme werden nicht von außen gesetzt, sondern erzeugen sich aufgrund ihrer inneren Organisation selbst.

Ein wichtiger Unterschied zwischen hierarchischen und heterarchischen Systemen besteht darin, daß sich heterarchische nicht mehr so einfach planen lassen wie hierarchische. Das eingangs erläuterte Beispiel zum Computerspiel *The Sims* zeigte, welche Phänomene bei der Etablierung von Nebenordnungen in der Praxis zu erwarten sind. Tatsächlich ist gerade der Computerspielmarkt ein hervorragendes Feld, um auch die Vorteile heterarchischer Organisationsformen zu zeigen. Den Schwierigkeiten in der Realisierung stehen Offenheiten gegenüber,

die mit herkömmlichen multilinearen Vorgehensweisen nicht zu erreichen sind. Bisher wurden Spiele auf der Basis vorgedachter Verzweigungsmöglichkeiten entwickelt. Der Spieldesigner definiert für die einzelnen Situationen eine Menge von möglichen Interaktionsformen, und der Spieler kann eine auswählen. Aufgrund der enormen Komplexität, die einige Spiele inzwischen erreicht haben, ist diese Vorgehensweise an ihre Grenzen gestoßen. In diesen komplexen Spielen treten zunehmend Situationen auf, in denen sich der Spieler eine plausible Strategie ausdenkt, der Entwickler diese aber einfach nicht vorgesehen hat. Spieler werden so gezwungen, zu überlegen, was der Entwickler sich gedacht haben könnte – ein Killerkriterium für jedes Spiel. Bei avancierten Computerspielen stellt sich deshalb die Frage, wie offen das Spiel für die Intentionen des Spielers ist und wie weit es sich zusammen mit den Strategien des Spielers entwickeln kann. Alle interaktiven Anwendungen, bei denen komplexe lebensnahe Situationen simuliert werden sollen, deren Ziel es ist, dem Nutzer das Gefühl zu vermitteln, vollkommen frei zu agieren (Computerspiele, Virtual Reality, Artificial Life, Digital Storytelling, etc.), verlangen nach dieser Form der Offenheit. In Virtuellen Welten zählt für den Besucher nicht ein Rechenergebnis, sondern die Erlebnisqualität. Derzeit konzentriert sich die Hoffnung der Entwickler auf Methoden der Künstlichen Intelligenz (KI), um die Restriktionen der Navigation auf prädeterminierten Pfaden in vollständig antizipierten Welten zu überwinden. Bisherige Methoden der KI beschränkten sich auf Pfad-Such-Probleme oder »Finite State Machines« für autonome Objekte. Zunehmend werden auch andere KI-Techniken wie BDI-Architekturen (*Belief-Desire-Intention*) und Lernmethoden verwendet, um offene Systeme und komplexes Verhalten zu erzeugen. Die Realisierung autonomer Objekte innerhalb virtueller Umgebungen verstärkt aber das Problem der Antizipation. Das Zusammenspiel der unabhängigen komplexen Einheiten läßt sich insbesondere bei lernenden, sich selbst modifizierenden Verfahren vom Autor nicht mehr vordenken. Diese prinzipielle Grenze der denkenden Antizipation des Autors muß deshalb in Zukunft Eingang in die allgemeinen Entwurfsprinzipen finden. Die im VR- und Spielebereich wichtigen »parallel hierarchical finite state machines« sind bereits ein erster Schritt in Richtung heterarchischer Modellierungsansätze.

Experiment und Spekulation

> *In the existing sciences whenever a phenomenon is encountered that seems complex it is taken almost for granted that the phenomenon must be the result of some underlying mechanism that is itself complex. But my discovery that simple programs can produce great complexity makes it clear that this is not in fact correct.* Stephen Wolfram[24]

In seinem Buch *A New Kind of Science* legt Stephen Wolfram eine der umfangreichsten Untersuchungen Neuronaler Netze vor. In über 20-jähriger Arbeit hat

24. Stephen Wolfram, *A New Kind of Science*, Wolfram Media Inc., Champaign 2002, S. 4.

er die Struktur zellulärer Automaten untersucht und auf eine Reihe fundamentaler wissenschaftlicher Probleme angewandt. Zu seinen grundlegenden Entdekkungen gehört, daß bereits einfachste Programme große Komplexität erzeugen können. Nach Wolframs Auffassung könnten eindimensionale zelluläre Automaten die einfachsten formalen Systeme sein, die zu komplexer Selbstorganisation fähig sind. Aber die Softwaresysteme, die wir im Bereich der Informatik mit gegenwärtigen Softwaretechniken entwickeln, tendieren dazu, in ihrer Struktur sehr komplex zu werden und dennoch nur einfaches Verhalten zu generieren, das mehr oder weniger vordefinierte Zwecke erfüllt. Wir benötigen also noch immer meist komplizierte Strukturen, um einfaches Verhalten zu erzeugen, selbst dieses einfache Verhalten bekommen wir aber nicht in den Griff. Die zurückliegenden Jahrzehnte der Computerentwicklung waren geprägt von dem Versuch, unsere Computersysteme beherrschbar zu machen. Mit mäßigem Erfolg, wie wir heute wissen. Der y2k-Bug, das Problem des Datumswechsels in den Softwaresystemen zum Jahr 2000, ist uns allen noch in lebhafter Erinnerung. Trotz enormer personeller und finanzieller Anstrengungen konnte niemand sagen, was wirklich passieren würde. Man hatte nicht das Gefühl, es mit einem sehr einfachen numerischen Problem zu tun zu haben, sondern vielmehr mit einem unvermeidlichen Naturereignis. Zwar weiß jeder, daß Computer streng deterministischen Gesetzen folgen, aber was hilft das, wenn wir sie auf dieser Basis offensichtlich weder beherrschen noch ihr Verhalten zuverlässig vorhersagen können? Diese Systeme haben eine Komplexität erreicht, die jenseits der Ebene deterministischer Befehlsfolgen liegt. Wir brauchen deshalb neue Methoden um brauchbare Laufzeitbeschreibungen von den Computersystemen anfertigen zu können. Da wir im Umgang mit Computern oft das Gefühl haben, launischen Naturphänomenen gegenüber zu stehen, sollten wir sie vielleicht auch einfach so behandeln. Die Physik, die ebenfalls theoretisch deterministische Systeme praktisch nie in den Griff bekommen hat, daraus aber gelernt hat, auf der Basis stochastischer Modelle viel exaktere Vorhersagen zu treffen, könnte hier Vorbild sein.

Von zukünftigen interaktiven Anwendungen wird – insbesondere auf dem Gebiet der so genannten Neuen Medien – eine größere Entwicklungsfähigkeit und Offenheit in der Interaktion verlangt, d.h. die Systeme sollen sich durch eine Dimension des »Werdens« auszeichnen. Wir begegnen hier dem alten Konflikt zwischen Kontrolle und Autonomie. Eigentlich wollen wir, daß diese Systeme uns überraschen, dieser Gewinn an Autonomie muß aber mit einem Verlust an Kontrolle erkauft werden. Es stellt sich also die Frage: Wie können wir solche Systeme entwickeln, die einerseits zuverlässig, robust und konsistent sind und andererseits interessantes und überraschendes Verhalten generieren? Da wir nach Jahrzehnten ernsthafter Anstrengungen erkennen, daß wir uns in der Praxis immer weiter vom Ziel der Kontrollierbarkeit entfernen, scheint es nun an der Zeit, auch der Überraschung eine Chance zu geben. Warum sollten wir nicht die Frage der Beherrschbarkeit – die wir offensichtlich sowieso nicht lösen können – bei diesen Anwendungen einmal hinten anstellen und die Frage der Autonomie ins Zentrum rücken.

Der gegenwärtige Entwicklungsprozeß für interaktive Anwendungen basiert aber auf dem Prinzip der Antizipation. Der Entwickler versucht die Strategien der Anwender vorherzusehen und entwirft das System so, daß es geeignet reagiert. Für eine Reihe praxisrelevanter Anwendungen ist diese Methode schon heute an ihre Grenzen gestoßen. Der Anwender wird sich vollkommen vernünftige Strategien ausdenken, aber die Anwendung wird nicht erlauben, diese umzusetzen. Die experimentelle und spekulative Informatik geht dagegen nach einem anderen Ansatz vor, ähnlich dem Prinzip das Stephen Wolfram für die Untersuchung neuronaler Netze angewendet hat. Es wird ein kleines Universum etabliert, d.h. es werden zuverlässige, robuste und konsistente Systeme mit parallelen (heterarchischen) und sich weiterentwickelnden (code-reflexiven) Aktivitätsträgern erzeugt. Das Verhalten des Systems wird dann, während das System läuft, aus dem Systementwurf emergieren. Die Beherrschbarkeit und analytische Vorhersagbarkeit dieser Systeme ist allerdings prinzipiell nicht mehr gegeben: 1. weil sich der Programmcode aufgrund von Code-Reflexionen während der Programmausführung selbst ändert, 2. aufgrund nichtvorhersehbarer Verbindungsmöglichkeiten zwischen den heterarchischen Elementen, 3. durch unbeabsichtigte Repräsentationen in den Interfaces. Diese Verbindungsmöglichkeiten hängen nicht zuletzt von der individuellen und kulturellen Vorprägung des Benutzers ab, wie auch von dem gewählten Interface. Kulturelles Wissen und Kenntnisse über Wahrnehmung werden für den Entwickler deshalb genauso wichtig, wie Wissen über formale Strukturen. Die einzige Möglichkeit herauszufinden, welches Verhalten tatsächlich erzeugt wird, besteht darin, die Systeme laufen zu lassen und sie dabei zu beobachten und zu analysieren. Man könnte diese Vorgehensweise deshalb auch als Performative Wissenschaft bezeichnen.