Informatik Spektrum (2020) 43:262–271
https://doi.org/10.1007/s00287-020-01294-z

HAUPTBEITRAG

Informatik – eine eigenständige Wissenschaft?

Wolfgang Reisig[1]

Online publiziert: 31. Juli 2020

Zusammenfassung
In diesem Beitrag frage ich nach einem Rahmen für eine umfassende Theorie der Informatik als eine formale Theorie der diskreten dynamischen Systeme, nach dem Vorbild der Theoriebildung in den Naturwissenschaften. An zahlreichen Beispielen zeige ich, dass dieses Unterfangen durchaus aussichtsreich ist und in isolierten Teilen schon vorliegt. Auf lange Sicht könnte die Informatik damit eine eigenständige Wissenschaft herausbilden, in Ergänzung zu ihren starken technologischen Aspekten, mit eigener theoretischer, mathematischer Basis, und auf Augenhöhe mit den Naturwissenschaften.

Einleitung

Hintergrund

Informatik wird gern erzählt als Erfolgsgeschichte, getrieben vom Moore'schen Gesetz, nach dem Informationstechnik in den vergangenen 60 Jahren exponentiell immer billiger, schneller und kleiner geworden ist [8]. Entsprechend wurde Informationstechnik für immer neue Aufgaben verwendet. Systematische Fehler im Verhalten von Geräten, gescheiterte Projekte, unbeherrschbares Verhalten rechnerintegrierter Systeme etc. werden als vermeintlich unvermeidbare Begleiterscheinungen hingenommen und gerechtfertigt als der Preis, den die rasante Entwicklung eben fordert.

Ob es auch anders oder besser hätte kommen können – oder vielleicht kommen sollen –, wird kaum diskutiert und schwerlich für möglich gehalten. Schon die Frage nach einer alternativen Entwicklung stößt auf Unmut, gerade bei jüngeren Wissenschaftlern, die das Bestehende so eifrig gelernt haben und die nun auf dieser vermeintlich soliden und unumstößlichen Basis aufbauen wollen, eingezwängt in Förderungs- und Belohnungssysteme, die die Beschäftigung mit fundamentalen Fragen an die Informatik kaum unterstützen. In dieser Situation erscheint es mir interessant, einige grundlegende Fragen zu stellen und Aspekte zu diskutieren, die mehr Systematik in die auseinanderstrebenden Teilgebiete der Informatik bringen könnten und die als Grundlage einer Wissenschaft der Informatik auch zukünftige Entwicklungen besser und schneller verständlich und nutzbar machen könnten.

✉ Wolfgang Reisig
reisig@informatik.hu-berlin.de

[1] Humboldt-Universität zu Berlin, Berlin, Deutschland

Informatik als Wissenschaft: historische Entwicklung

Kommerziell wurde Rechentechnik in den 1950er-Jahren vorwiegend für zwei Bereiche verwendet: für numerische Aufgaben der klassischen Ingenieurwissenschaften und für das Durchforsten größerer Datenbestände, die beispielsweise bei einer Volkszählung entstehen. Die beiden Programmiersprachen FORTRAN und COBOL waren darauf zugeschnitten; der Weg von einer Aufgabe über eine algorithmische Idee hin zu einem Programm war im Allgemeinen hinreichend beherrschbar. Dies änderte sich in den 1960er-Jahren: Mit wachsender Rechenleistung der Computer erschienen nun auch Aufgaben lösbar, die komplexere algorithmische Ideen und umfangreichere Programme erfordern. Allerdings wurden Programme zunehmend fehleranfällig und undurchschaubar und schließlich wurde 1968 bei einer berühmt gewordenen Konferenz [30] in Garmisch-Partenkirchen eine „Software-Krise" konstatiert, der mit einem „Software Engineering" begegnet werden sollte. Das führte zu neuen Programmiersprachen (PL/1, Pascal, ADA), neuen Programmierparadigmen (strukturiertes Programmieren, Objektorientierung) und später zu neuen Software-Architekturen (Komponenten, Serviceorientierung, Micro-Services). Zudem wurden Software-Entwicklungsmethoden wie das systematische Verfeinern und die Spezifikation von Schnittstellen propagiert.

In der Rückschau ist offensichtlich, dass die Garmisch-Partenkirchener Konferenz wichtige Fragen aufgeworfen und zielführende Antworten angestoßen hat. Aber alle dor-

tigen Vorschläge gestalten letztlich den langen Weg von einer Aufgabe über eine algorithmische Idee hin zu einem Programm nur im letzten Abschnitt etwas bequemer.

Die Denkweise der „Computer Science“ als Wissenschaft von Beginn bis in jüngste Zeit kann man wie in einem Brennglas an der „Memorial Lecture“ für den berühmten Informatiker E.W. Dijkstra aus dem Jahr 2010 studieren, wo der ebenfalls berühmte Informatiker Tony Hoare vier Kriterien für Wissenschaft formuliert und für die Gegenstände der Computer Science spezialisiert [21]:

- **Description:** Wissenschaft beschreibt Eigenschaften und Verhalten von Systemen; in der Computer Science kann eine solche Beschreibung als Spezifikation dienen, insbesondere als präzise Schnittstelle zwischen Kunde und Hersteller. Als Beschreibungsmethode schlägt Hoare logikbasierte Kalküle vor; als Beispiel weakest preconditions.
- **Analysis**: Die zentralen Gegenstände und ihr konzeptioneller Zusammenhang werden erläutert. Gegenstand der Computer Science sind Programme. Sie werden analysiert mit pre-/postconditions, beispielsweise Microsoft-Contracts.
- **Explanation:** Hier wird nach Begründungen für das Verhalten von Systemen gesucht. In der Computer Science liefert die Semantik von Programmiersprachen solche Begründungen.
- **Prediction:** Gute Wissenschaft kann Ereignisse vorhersagen. In der Computer Science heißt das, das korrekte Verhalten eines Programms vorherzusagen, es also zu verifizieren.

Ein Programm wird also als mathematisches Objekt aufgefasst, seine Eigenschaften sollen in der Sprache der Logik formuliert werden, seine Bedeutung soll formal gefasst und seine Korrektheit soll formal bewiesen werden. Das sind zweifellos wissenschaftliche Konzepte. In diesem Sinn betrachtet auch Donald Knuth in [18] das Programmieren als eine Wissenschaft.

Dieser Beitrag

In 14 Abschnitten formulieren wir einige Gedanken zu einer umfassenden Theorie der Informatik als eine formale wissenschaftliche Theorie der dynamischen Systeme. Der Gegenstand dieser Wissenschaft soll allerdings sehr viel breiter gefasst werden als der oben geschilderte von Dijkstra, Hoare, Gries und Knuth. Dennoch soll diese Theorie formal gefasst sein und nicht etwa lediglich die vorhandenen formalen Konzepte um sozialwissenschaftliche, damit informale, Konzepte ergänzen. Beispiele für formale Konzepte der Informationsverarbeitung, die nicht zur Implementierung vorgesehen sind, finden sich in der Wirtschaftsinformatik und in eingebetteten Systemen.

In diesem Beitrag werden Konzepte erläutert und Fragen aufgeworfen, die teilweise bisher in der Literatur wenig beachtet wurden. Einige vorgestellte Konzepte sind aber nicht wirklich neu; sie wurden isoliert schon vielfach diskutiert. Neu ist vielleicht ihre Zusammenstellung, verbunden mit dem Anspruch, geeignet kombiniert und um weitere Konzepte ergänzt, zu einer umfassenden Theorie der Informatik beizutragen.

Die hier vorgestellten Ideen beziehen sich aufeinander; wie genau sie zusammenhängen, bleibt offen. Vielleicht erweisen sich einige dieser Ideen später als wenig fruchtbar – und zu einer runden Theorie der Informatik fehlen vermutlich noch eine Reihe weiterer grundlegender Ideen.

In diesem Beitrag möchte ich lediglich zeigen, dass es sich lohnt, nach einer umfassenderen Theorie der Informatik zu suchen.

1. Erfolgreiche Modellbildung in der Wissenschaft

Gute Wissenschaft bildet Theorien, indem sie *Modelle* entwickelt. Ein Modell ist ein System von Begriffen und Bezügen zwischen den Begriffen, um die Realität besser zu verstehen. „Realität“ liegt dabei entweder in der Natur vor oder wird – wie im Fall der Informatik – in Teilen von Menschen konstruiert. Ein besonders eindrucksvolles Beispiel ist die Physik: Jahrhundertelang haben Physiker nach einem vereinheitlichenden Modell für alle Zweige der Physik gesucht und so ein eindrucksvolles Gebäude wissenschaftlicher Theoriebildung entwickelt. Insbesondere entstand eine erstaunliche Harmonie von Physik und Mathematik [26] mit sehr abstrakten, aber überaus nützlichen wissenschaftlichen Konzepten und Modellen. Ein typisches Beispiel ist der Begriff der *Energie*. Mit ihm können ganz unterschiedliche Phänomene aufeinander bezogen und quantitativ bestimmt werden, die beispielsweise auftreten, wenn ein stehendes Fahrzeug beschleunigt und an eine Wand gefahren wird: In diesem Prozess tritt eine bestimmte Menge an Energie auf, die erst im Benzin, dann in der Beschleunigung und schließlich in der Umformung von Blech steckt. Die Nützlichkeit des Begriffs *Energie* liegt in seiner quantifizierbaren Invarianz für dynamische Prozesse. Derartige Invarianten, oft als „Naturgesetze“ bezeichnet, sind das Gerüst von Wissenschaft. In den letzten Jahren sucht beispielsweise auch die „Systembiologie“ nach solchen Begriffen und Invarianten zum besseren Verständnis von Stoffwechselprozessen aller Art.

Informatik sollte von der Physik und anderen Wissenschaften lernen, eine Theorie für ihre Gesamtheit zu bilden. John McCarthy hat schon 1963 in [27] die Entwicklung einer mathematikbasierten „Science of Computation“ angemahnt angeregt, in der die wichtigen Eigenschaften und

Gegenstände dieser Wissenschaft aus wenigen grundlegenden Annahmen abgeleitet werden.

2. Modellbildung in der Informatik

Indem Modelle den Kern einer wissenschaftlichen Theorie ausmachen, stellt sich die Frage nach Modellen in der Informatik. Wie oft in Naturwissenschaften, geht es auch in der Informatik um die Beschreibung *dynamischer* Eigenschaften, allerdings mit einem generellen und fundamentalen Unterschied: Physik beschreibt Verhalten im Allgemeinen kontinuierlich, mit Funktionen über einer reellen Zeitachse. Damit kann man wunderbar Differenzialgleichungen und Integrale bilden und vieles mehr. Informatik beschreibt Verhalten in *diskreten Schritten*; damit müssen und können ganz andere Eigenschaften beschrieben und mit ganz anderen mathematischen Analysetechniken analysiert werden.

Modelle werden in der Informatik für verschiedene Zwecke verwendet:

- um Sachverhalte einer Domäne zu beschreiben, beispielsweise die Struktur der Buchhaltung einer Firma. Inwieweit diese Beschreibung „korrekt“ ist, bleibt zwangsläufig informal,
- um algorithmisches Verhalten zu beschreiben, beispielsweise das Beantragen einer Leistung einer Versicherung,
- um die Leistung einer Software zu beschreiben, entweder über ihre Eigenschaften oder ihr operationales Verhalten.

Ein Modell in der Informatik ist immer eine symbolische Beschreibung. Ein Modell kann auf der Ebene der symbolischen Beschreibung *ausführbar* sein und so auf einem Rechner simuliert werden. Es kann aber auch konzeptionell das Verhalten eines Systems in einer konkreten Domäne beschreiben, ohne implementiert zu werden. Mit einer wirklich nützlichen Theorie der Informatik wird es sich lohnen, Korrektheit auf der Ebene der Modelle zu verhandeln und dann systematisch korrekte Software abzuleiten.

Die nahtlose Integration von Rechentechnik mit ihrer organisatorischen oder technischen Umgebung ist für viele rechnerintegrierte Systeme notwendig, nicht zuletzt für das *Internet der Dinge*: Das ist nur beherrschbar mit formalen Modellen, die sowohl Rechentechnik als auch ihre Umgebung einschließen. Dazu gehört insbesondere auch die Modellierung technischer oder organisatorischer Komponenten, die gar nicht implementiert werden sollen.

Nun gibt es eine Reihe von Modellierungstechniken, die diesen Aspekt unterstützen; allen voran die Unified Modeling Language (UML) [6] und – primär für die Wirtschaftsinformatik – Business Process Model and Notation (BPMN) [1]. Diese Techniken schlagen eine Vielzahl grafischer Ausdrucksmittel vor, um spezielle, subtile, domänenspezifische Sachverhalte auszudrücken. Und beide Techniken (und eine Reihe weiterer, beispielsweise Statecharts) werden durchaus zur Modellierung genutzt. Aber alle diese Techniken bieten wenige Möglichkeiten, gewünschte Eigenschaften des modellierten Systems im Modell nachzuweisen. Selbst wenn man also mit diesen Techniken modelliert und daraus Software generiert, wird nicht auf Modellebene formal über Korrektheit argumentiert, sondern auf der Software-Ebene.

Daneben gibt es allerdings durchaus Modellierungstechniken mit Analyse- und Verifikationskonzepten. Dazu gehören beispielsweise ALLOY, B, Focus, Live Sequence Charts, Petrinetze, TLA und Z, neben vielen anderen. Einige davon sind allerdings eher Konzeptstudien für ganz spezielle Eigenschaften und Aspekte.

Eine Modellierungstechnik wird sich auf Dauer nur durchsetzen, wenn sie starke Analysetechniken anbietet.

3. Vertrauenswürdige Modelle

In einem generellen systematischen Aufbau von Prinzipien der Modellierung in der Informatik sollte es einem Modellierer möglich sein, mit einer passenden – formalen – Modellierungstechnik ein System vertrauenswürdig, nachvollziehbar und eindeutig zu beschreiben. Für ein gegebenes Beispiel haben wir normalerweise ein gutes Verständnis dafür, was eine angemessene Beschreibung ausmacht: Sie enthält alle Aspekte, die dem Modellierer wichtig sind, und vermeidet Aspekte, über die der Modellierer nicht reden möchte. Konkreter formuliert beschreibt ein vertrauenswürdiges Modell M eines diskreten Systems S:

- jedes elementare Objekt von S als elementares Objekt von M,
- jede elementare Operation von S als elementare Operation von M,
- jedes komponierte Objekt von S als entsprechend komponiertes Objekt von M,
- jede komponierte Operation von S als entsprechend komponierte Operation von M,
- jeden – lokalen – Zustand von S als Zustand von M,
- jeden – lokalen – Schritt von S als Schritt von M.

Zusammengefast: Elementare und zusammengesetzte Objekte und Operationen sowie Zustände und Schritte von S und M sollten sich bijektiv entsprechen. Intuitives und formal dargestelltes Verhalten sollen also so eng wie irgend möglich zusammenhängen.

Nun stellt sich die Frage nach einer Modellierungstechnik, die diese Forderungen erfüllt, zumindest für eine hinreichend große Klasse von Systemen. Es ist offensichtlich, dass solche Objekte, Operationen, Zustände und Schritte im klassischen Sinn nicht immer berechenbare Funktionen sind. Mehr oder weniger deutlich wurde dieses Problem im-

mer wieder angesprochen. Schon in seinem grundlegenden Werk *The Art of Computer Programming* [22] führt Donald Knuth mit *computational methods* (heutzutage oft als *Transitionssystem* bezeichnet) einen allgemeineren Begriff des *Algorithmus* ein: Eine computational method besteht aus einer Menge S von Zuständen und einer Next-state-Funktion F auf S, von der es heißt *„F might involve operations that mortal man can't always perform."* [22, S. 8]. Knuth nennt eine computational method *effektiv*, wenn F eine berechenbare Funktion ist. Durchaus selbstkritisch erklärt Robin Milner in seiner EATCS award lecture [29]: *„... we should have achieved a mathematical model of computation, perhaps highly abstract in contrast with the concrete nature of paper and register machines, but such that programming languages are merely executable fragments of the theory ...."* Es bleibt offen, welche Art nichtausführbarer Fragmente Milner meint.

Oft wird versucht, das klassische Berechnungskonzept aus Variablen, effektiven Objekten und Operationen über Zeichenketten und Programmen aus Wertzuweisung mit Sequenz, Alternative und bedingten Schleifen zu verallgemeinern auf freigewählte mathematische Objekte und Operationen, beispielsweise in [37]. Shepherdson in [33] entwickelt eine ähnliche Idee auf der Basis von Turingmaschinen. In einem etwas anderen Stil und aus rein logischer Perspektive betrachten Abstract State Machines [19] Programme über einer Signatur (einer Symbolmenge verschiedener Sorten); der Nutzer des Formalismus kann dann eine Signatur und seine gewünschte Struktur über dieser Signatur frei wählen und mit Termen über die Struktur reden.

Ein gutes Modell verwendet also Symbole, die in der modellierten Welt unmittelbar als Objekte oder Operationen interpretiert werden. Das unterscheidet Modelle von Programmen: Eine Programmiersprache fixiert die Interpretation der Symbole eines Programms.

4. Invarianten in der Informatik

Invarianz ist ein Eckpfeiler wissenschaftlicher Theoriebildung (Abschn. 1); damit stellt sich die Frage nach Invarianzbegriffen in der Informatik. Der bekannteste derartige Begriff ist sicherlich Hoares Invariantenkalkül [20] mit seinem Konzept der Schleifeninvarianten für klassische Programme [16] bezeichnet Schleifeninvarianz als eine der grundlegenden Ideen der Software-Konstruktion. Das mag zutreffen, wenn man die Korrektheit eines Systementwurfs ganz am Ende, bei der Programmerstellung in einer klassischen Programmiersprache ansiedelt. Eine Theorie der Informatik sollte allerdings *Korrektheit von Modellen* betrachten. Tatsächlich arbeiten viele Modellierungstechniken mit speziellen Invarianten: [36] schlägt beispielsweise Invarianten für Modelle verteilter Prozesse, verteilter Algorithmen und Kommunikationsprotokolle vor. Der Invariantenkalkül für Petrinetze ist besonders ausdrucksstark (weil Transitionen reversibel sind) [31].

Jede solche Invariante beschreibt zwischen Systemvariablen eines Systems einen Zusammenhang, der in allen erreichbaren Systemzuständen invariant bleibt. Diese Invarianzbegriffe bleiben damit sehr eng am gegebenen Systemmodell. Wie das Beispiel des Begriffs der Energie (Abschn. 1) zeigt, kennt die Physik weitaus tiefer liegende, weniger offensichtliche und dennoch (oder deshalb) äußerst nützliche Invarianten. Vergleichbar tief liegende Invarianzbegriffe sollte die Informatik auch anstreben. Solche Invarianten könnten vielleicht genau bestimmen, was invariant bleibt, wenn beispielsweise rechnergestützt:

- ein Reisebüro eine Reise bucht,
- ein Bankkunde an einem Automaten Bargeld abhebt,
- eine Lebensversicherung einen bestehenden Vertrag auf eine neue Hardware umstellt,
- eine Autoversicherung einen Schaden reguliert,
- ein Rechenzentrum das Wetter von morgen berechnet,
- ein Betriebssystem den Speicher aufräumt,
- ein Compiler ein Programm übersetzt.

Derzeit ist das alles Spekulation; es loht aber, systematisch nach Modellierungstechniken zu suchen, die im Vergleich mit bisherigen Konzepten weit abstraktere und weniger naheliegende, aber dennoch intuitiv verständliche Begriffe und Invarianten verwenden.

5. Ein fundamentaler Begriff (von „Information")

Wie das Beispiel des Energiebegriffs zeigt, kann Invarianz auf einem sehr abstrakten, dennoch intuitiv verständlichen Begriff beruhen. Wie sähe nun ein solcher Begriff der Informatik aus? Als Beispiel versuche ich es mit einem Begriff von „Information", mit der zentralen Invariante der Informationserhaltung in Schritten: Ein gegebener Zustand eines Systems enthält eine Menge „Information". Solange das System nicht mit seiner Umgebung kommuniziert, kann das System die Information in ganz verschiedener Weise wandeln, neu kombinieren, es kann etwas berechnen, auch kann ein Schritt Teile der Information unzugänglich machen. Aber die Information des Systems bleibt in ihrer Gesamtheit erhalten. Eine Berechnung, eine Sequenz von Schritten, modelliert dann einen *strikt organisierten Fluss* von Information.

Eine konstruktive Definition eines solchen Informationsbegriffs ist nicht in Sicht. Sie hätte aber eine Reihe überaus wertvoller Konsequenzen, beispielsweise sollten damit die in Abschn. 4 beschriebenen Invarianten formal fassbar sein. Man sollte präzise formulieren können, was genau sich än-

dert und was gleich bleibt beim Kopieren, Löschen oder Zusammenfügen von Informationen oder Dokumenten. Datenschutz, Schutz der Privatsphäre und verwandte Begriffe könnten eine sehr viel präzisere Bedeutung bekommen.

Eine weitere interessante Eigenschaft eines solchen Informationsbegriffs sind *informationserhaltende Operationen*: Eine solche Operation *f* verliert keine Informationen; ein Argument *a* für *f* kann also aus dem Resultat *f(a)* zurückberechnet werden. Beispiele sind die Negation der Aussagenlogik und die positive Wurzel aus positiven reellen Zahlen. Solche reversiblen Funktionen sind oft betrachtet worden, vorwiegend im Kontext von Aussagenlogik und von Schaltkreisen [39]. Die Idee reversiblen Rechnens könnte IT Sicherheit entscheidend voranbringen: Ein Angreifer könnte retrospektiv zurückverfolgt werden.

Vielleicht gelingt die Charakterisierung spezieller Informationsbegriffe über die Operationen, die mit ihnen in einem jeweils gegebenen Kontext möglich oder erlaubt sind.

6. Interaktive Komponenten

Klassische theoretische Informatik abstrahiert informationstechnische Prozesse zu berechenbaren Funktionen über Symbolketten. Mit den tief liegenden Resultaten zur Komplexitätstheorie und zu den Zusammenhängen zwischen Logik und Automatentheorie hat sich dieser Ansatz als theoretische Grundlage der Informatik etabliert. Das bekannteste operationale Modell dafür sind Turingmaschinen. Eine Turingmaschine mit einem Speicher und einem Prozessor kann als Abstraktion eines Rechners der 1960er-Jahre aufgefasst werden. Eine Multiprozessorarchitektur, die mehrere Prozessoren verwendet, um die Rechengeschwindigkeit zu erhöhen, lässt sich noch plausibel zu einer Architektur mit einem einzigen Prozessor abstrahieren. Ein System aus mehreren, miteinander kommunizierenden aktiven Komponenten kann auf einem 1-Prozessor-System zwar simuliert werden, lässt sich aber nur schlecht als 1-Prozessor-System auffassen, ohne den Sinn des Systems zu verfälschen.

Erste solche Systeme wurden in den 1980er-Jahren vorgeschlagen, insbesondere der Actor-Formalismus von Agha und Hewitt [2]. Ähnliche Ideen verfolgte beispielsweise die Sprache LINDA [11] und die Chemical Abstract Machine (*CHAM*) [5]. In dieser Metapher werden aktive Elemente als eine Art Moleküle aufgefasst, die in einer – metaphorischen – chemischen Lösung schwimmen und miteinander reagieren, sobald sie aufeinandertreffen und geeignete Reaktionsbedingungen bestehen. In diesem Rahmen sind zahlreiche Algorithmen formuliert worden. Ein anderes Beispiel ist der FOCUS-Formalismus von Broy [10], mit Komponenten, die stromverarbeitende Funktionen ausbilden. Schließlich gehören auch Petrinetze [31] in diese Reihe von Modellierungstechniken: jede Transition kann aufgefasst werden als elementare aktive Einheit.

Es stellt sich die Frage nach einer Theorie, die für derartige und viele weitere Modellierungstechniken eine angemessene abstrakte Grundlage bildet, in Analogie zu den berechenbaren Funktionen für sequenzielle Ein-/Ausgabealgorithmen.

In einer Reihe von Papieren schlägt Wegner dafür Verallgemeinerungen von Turingmaschinen vor, beispielsweise in [40] und in [41]. In [13] wird allerdings die Korrektheit von Wegners Argumentation bestritten.

Mir geht es hier nicht um die Frage, „was ein Computer im Prinzip kann“, sondern um die wesentlich breitere Fragestellung, wie Systeme angemessen modelliert und analysiert werden können, die in diskreten Schritten voranschreiten und mit ihrer Umgebung interagieren.

7. Unabhängige Schritte

Das Konzept diskreter Schritte beruht auf Zuständen: Ein Schritt beginnt in einem Zustand und endet in einem Zustand. Im klassischen Rahmen von Systembeschreibungen sind Zustände *global*: Jeder Schritt aktualisiert den gegenwärtigen Zustand. Ein einzelnes Verhalten, ein Ablauf, eine Berechnung, ist dann eine Sequenz s_0-t_1-s_1-t_2-s_2 ... von Zuständen s_i und Schritten s_{i-1}-t_i-s_i (i = 1,2, ...). Viele Systemmodelle konstruieren globale Zustände eines aus Komponenten C_1, ..., C_n komponierten Systems als kartesisches Produkt $S_1 \times ... \times S_n$ der Zustandsräume S_k der Komponenten C_k. Dabei entstehen für jeden Schritt einer Komponen-

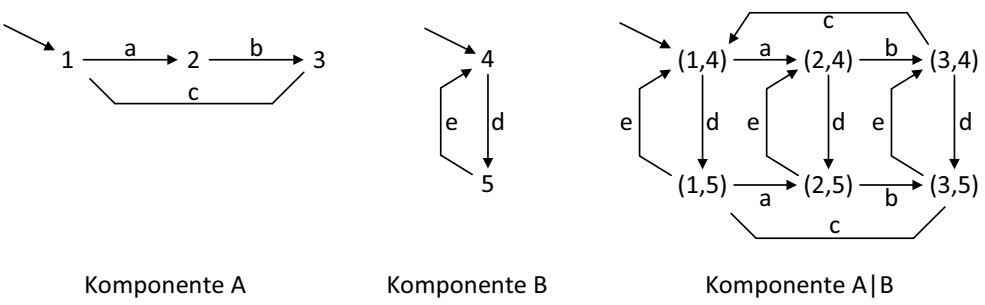

Abb. 1 Kartesisches Produkt A|B zweier Komponenten A und B

te mehrere Schritte des komponierten Systems. Voneinander unabhängige Schritte in verschiedenen Komponenten werden in diesem Modell nichtdeterministisch in beliebiger Reihenfolge notiert.

Abb. 1 zeigt ein kleines Beispiel: A und B seien zwei unabhängige Komponenten mit 3 bzw. 2 Zuständen und Schritten, die in Zyklen angeordnet sind. Jede der beiden Komponenten hat also genau einen unendlich langen Ablauf. Die Komposition A|B von A und B ist eine Komponente mit 6 Zuständen. In jedem Zustand beginnen 2 Schritte; damit entstehen unendlich viele unendlich lange Abläufe. Diese Auffassung von Verhalten liegt intuitiv nahe und wird von vielen Analyseverfahren verwendet, insbesondere von Verfahren des Model Checking.

Diese Auffassung hat aber auch Nachteile: Die beiden in einem Zustand beginnenden Schritte sehen aus wie Alternativen; tatsächlich aber treten sie beide unabhängig voneinander ein! Alternativ daran erscheint die – auf Uhren außerhalb des Systems gemessene – zeitliche Reihenfolge der Schritte. [24] diskutiert Einzelheiten derartiger Annahmen über zeitliche Reihenfolgen und benötigt dafür perfekt genau gehende Uhren. Um solche Annahmen zu vermeiden, kann man ausnutzen, dass unabhängige Schritte in *disjunkten lokalen* Zuständen beginnen und enden. Damit kann man sie als *ungeordnet* notieren. Ein Ablauf ist dann nicht eine Sequenz, sondern eine *Halbordnung* von Schritten. Ordnung bezeichnet dann nicht das Voranschreiten in der Zeit, sondern nur einen „Vorher-Nachher"-Zusammenhang. Das System A|B aus Abb. 1 hat in dieser Auffassung dann wiederum nur einen Ablauf, der die beiden Abläufe von A und B vereinigt und dessen Ordnung die Ordnungen der beiden Abläufe vereinigt. Mit der zusätzlichen Forderung, dass B niemals mehr lokale Schritte ausführt als A, hat das System immer noch genau einen Ablauf, den Abb. 2 zeigt. Petrinetze verfolgen diesen Vorschlag mit dem Konzept des „verteilten Ablaufs" [31].

Einen weiteren Nachteil der Auffassung eines einzelnen Ablaufs eines Systems als eine Sequenz von Schritten verdeutlicht Lamports Beispiel einer Stundenuhr, die von einer vollen Stunde zur nächsten in *einem* Schritt übergeht. Eine Stunden-und-Minuten-Uhr braucht dafür *sechzig* Schritte, ist also streng genommen nicht als Stundenuhr verwendbar. Lamport schlägt deshalb in seinem TLA-Formalismus [25] eine Logik vor, die einzelne Schritt konzeptuell logisch mit „beliebig viele Schritte" gleichsetzt.

Beide Beispiele zeigen, dass die konzeptuelle Auffassung eines einzelnen Ablaufs eines Systems als Sequenz von Schritten unschöne Konsequenzen hat, wenn Systeme komponiert oder verfeinert werden. Deshalb lohnt es sich, unabhängige Ereignisse mit ihren lokalen Ursachen und Wirkungen ernst zu nehmen, und von Nichtdeterminismus zu unterscheiden. Das gilt insbesondere für große Systeme, die ja meistens systematisch als Komposition oder Verfeinerung kleinerer Systeme konstruiert werden. [23] schlägt dafür eine spezifische Logik vor.

8. Beschränkte Ausdruckskraft der Wertzuweisung

Nicht nur Programmiersprachen, sondern auch viele Modellierungssprachen beschreiben Schritte als Wertzuweisung an Variablen. In vielen Fällen ist das angemessen oder zumindest akzeptabel, kann aber auch zu wenig überzeugenden Modellen führen. Ein Beispiel ist das „Pebble Game", das Dijkstra in einem Video einer anspruchsvollen Serie der Stanford-Universität beschreibt [15]: Eine Urne enthält endlich viele schwarze und weiße Steine. So lange wie möglich werden wiederholt zwei Steine a und b entnommen und ein Stein c zurückgelegt. Dabei ist c weiß, wenn a und b verschieden gefärbt sind, sonst ist c schwarz. Abb. 3 zeigt Dijkstras Modell: ein nichtdeterministisches Programm aus bedingten Wertzuweisungen mit arithmetischen Operationen auf vier Integer-Variablen. Abb. 4 modelliert das Spiel als Petrinetz: *PEBBLE* ist ein Konstantensymbol, das mit schwarzen und weißen Steinen initialisiert wird. Jede Transition modelliert einen Schritt des Systems, bestehend aus zwei entnommenen und einem zurückgelegten Stein. Die Beschriftung der Pfeile jeder Transition zeigen die Färbung der beteiligten Steine. In diesem Modell wird „entnehmen" und „zurücklegen" unmittelbar modelliert. Der Um-

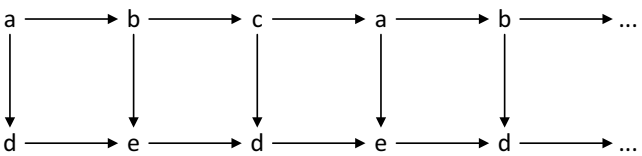

Abb. 2 Verteilter Ablauf von A|B mit der zusätzlichen Forderung, dass B niemals mehr lokale Schritt ausführt als A

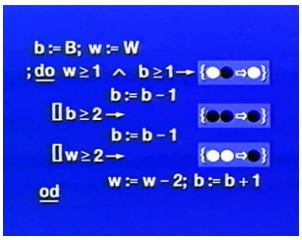

Abb. 3 Dijkstras Lösung [15] des Pebble-Spiels

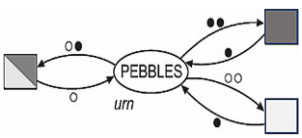

Abb. 4 Petrinetz-Lösung des Pebble-Spiels

weg über die Anzahl der beteiligten schwarzen und weißen Steine wird im Modell vermieden. Mit einer Invariante zeigt Dijkstra, dass nach endlich vielen Schritten genau ein Stein übrig bleibt. Der ist genau dann weiß, wenn die Anzahl weißer Steine anfangs ungerade ist. Eine entsprechende Invariante gibt es allerdings auch für das Petrinetz [31].

Variablen und Wertzuweisung sind auch wenig geeignet zur Modellierung verteilter Systeme, Kommunikationsprotokoll, etc. Ein Beispiel ist das TLA-Modell eines asynchronen Interfaces aus [25]. Letztendlich ist dort ein Scheduler nötig, der den Zugriff mehrerer Komponenten auf die gemeinsam benutzten Variablen regelt. Die Komponenten sind also nur unter weitreichenden weiteren Annahmen wirklich verteilt implementierbar.

Eine Theorie der Informatik wird bei der Beschreibung von Schritten nicht nur Wertzuweisung verwenden, sondern eine Vielzahl anderer, ggf. abstrakterer Konzepte. Ein Beispiel sind Petrinetze: Die Bedeutung einer Transition ist gegeben durch die Änderungen der Markierung der Plätze in ihrer unmittelbaren Umgebung.

9. Die Metapher des lebenden Organismus

Neue Systeme können durch Verfeinern und Komponieren gegebener Systeme gebildet werden. Es stellt sich die Frage nach weiteren Methoden zur systematischen Konstruktion neuer Systeme aus gegebenen. Zu den wenigen Vorschlägen dafür gehört die Metapher des 3D-Druckens, sowie die „Living-organism"-Metapher nach [14]. Dort können mehrere „Zellen" gemeinsam selbstständig „Kreaturen" bilden, die ganz neue Eigenschaften haben. Generell stellt sich die Frage nach der Grenze solcher Konstruktionen und nach dem „Nichtkonstruierbaren", in Analogie zur Grenze der Berechenbarkeit bei klassischem Rechnen.

10. Korrektheit, Verifikation

Wissenschaftliche Theorien leben von Modellen, die interessante Konsequenzen zeigen. Ein Modell ist also erst dann wirklich nützlich, wenn man daraus etwas Interessantes ableiten kann. Der rein beschreibende Charakter von UML, BPMN, ASM und anderen Modellierungstechniken ohne spezifische Analyseverfahren schränkt entscheidend ihren Einsatz ein. Ein wirklich gutes Modell eines Systems ist vertrauenswürdig (vgl. Abschn. 3) und zugleich analysierbar, insbesondere mit tiefliegenden Invarianten (vgl. Abschn. 4).

Viele Eigenschaften von Systemen lassen sich zurückführen auf Eigenschaften einzelner Zustände und Abläufe. Zur Beschreibung solcher Eigenschaften hat temporale Logik eine dominante Stellung erreicht [17]; mit Model Checking und abstrakter Interpretation als effizienten Analyseverfahren. Sehr nützlich ist auch die Unterscheidung von Liveness- und Safety-Eigenschaften von Alpern und Schneider.

Eine Theorie der Informatik sollte jedoch wesentlich tiefer liegende Eigenschaften formulieren und nachweisen können, auf der Basis nichttrivialer Invarianten (Abschn. 4) und einem spezifischen Begriff von Informationsfluss (Abschn. 5).

Ein Nutzer eines großen Systems benötigt einen modifizierten Begriff von „Korrektheit": Ihn interessiert nicht wirklich, ob das ganze System korrekt ist (viele große Systeme sind – für Grenzfälle – ohnehin nicht korrekt), sondern ihn interessiert nur, ob das System für seinen aktuellen Fall korrekte Resultate liefert. Idealerweise liefert ihm das System dafür eine plausible, ihm verständliche Begründung. Klassische Verifikation würde beide Anforderungen verfehlen: Auch ein fehlerhaftes System kann nützlich sein und der Hinweis auf gelungene formale Verifikation einer temporallogisch formulierten Eigenschaft stärkt nicht unbedingt das Vertrauen in die Korrektheit einer einzelnen Nutzung. Erste Vorschläge dazu sind *zertifizierende Algorithmen* [28] und die Verifikation zur Laufzeit [4]. Eine Theorie der Informatik muss flexible Konzepte zum Begriff der Korrektheit und ihres Nachweises suchen.

11. Zeit, Kausalität, Beobachtung, etc.

Für rechnergestützte Echtzeitsysteme, beispielsweise die Steuerung eines Airbags, sind klassische Echtzeitmodelle angemessen. Viele Modellierungstechniken argumentieren mit einem naiven Zeitbegriff, als sei die jeweils aktuelle Zeit in beliebiger Präzision und ohne zusätzlichen Aufwand verfügbar. Lamport [24] weicht diese Sicht ein wenig auf, ohne sich ganz davon zu lösen. Manche Modellierungstechniken, beispielsweise ESTEL, ESTEREL und Statecharts, verwenden die Hypothese „unendlich schneller" digitaler Systeme, weil solche Systeme tatsächlich um Größenordnungen schneller als ihre menschlichen Nutzer arbeiten.

Tatsächlich wird „Zeit" oft verwendet, wo es konzeptionell eher um Ursache-Wirkung-Zusammenhänge geht. „Zeit" und „Kausalität" in Systemen, sowie „Beobachtung" in Modellen sind Begriffe, deren Zusammenhang im Rahmen der Modellbildung der Informatik noch nicht wirklich verstanden ist. Ein herausforderndes Beispiel ist eine formale Beschreibung des *Applesort-Algorithmus* (vgl. [35]): Äpfel rollen hintereinander über ein schräg gelagertes Brett mit zunehmend größeren Löchern. Jeder Apfel fällt durch das erste Loch, dessen Durchmesser den Durchmesser des Apfels erreicht oder übertrifft. Bei n Löchern klassifiziert dieser Algorithmus die Äpfel in n Größenklassen.

12. Verfeinern und Komponieren

Große Systeme werden in aller Regel aus abstrakten Anforderungen verfeinert oder aus kleineren Systemen zusammengesetzt. Es gibt eine Vielzahl genereller Methoden, Prinzipien und Formalismen (z. B. [3]) zur systematischen Verfeinerung von Spezifikationen. Für logische Spezifikationen fundamental ist das Prinzip der *Verfeinerungsimplikation*: Die Spezifikation der Verfeinerung impliziert die Spezifikation des gegebenen Systems. Entsprechende Methoden, Prinzipien und Formalismen zur Komposition logischer Spezifikationen wurden für die Sprache *Z* [34], Lamports *TLA* [25] und Broys strombasiertes *Focus* [10] vorgeschlagen, mit dem weitreichenden Vorschlag, Komposition logisch als Konjunktion zu fassen. Auf operationeller Ebene schlägt [32] einen generellen Operator zur Komposition vor, der assoziativ ist und keine Spezifikation des Inneren von Komponenten verlangt. Alle diese Methoden, Prinzipien und Formalismen stellen richtige und wichtige Fragen, haben sich aber nicht wirklich durchgesetzt.

13. Berechenbarkeit

Die Theorie der berechenbaren Funktionen wurde lange als Grundlage einer Theorie der Informatik angesehen. Alle Versuche, die Church-Turing-These zu widerlegen, sind gescheitert. Allerdings wurde diese These oft unzulässig allgemeiner formuliert. Tatsächlich beschreibt sie lediglich einen Sachverhalt zur systematischen Manipulation von Zeichenketten [38]. Informatik und damit eine Theorie der Informatik geht weit über die Manipulation von Zeichenketten hinaus; vielmehr geht es um die *Interpretation* von Zeichen in der realen Welt. Mit der Manipulation von Zeichenketten werden dann Effekte in der realen Welt beschrieben und erreicht. Diese Zusammenhänge werden in [12] nüchtern dargestellt. In einer Theorie der Informatik hat die Theorie berechenbarer Funktionen einen wichtigen Platz – neben anderen Konzepten. Das gilt entsprechend auch für die formale Logik.

14. Informatik als Engineering-Disziplin

Eine Engineering-Disziplin, beispielsweise Electrical Engineering oder Chemical Engineering, basiert auf einer Wissenschaft (im Beispiel Physik und Chemie), deren Prinzipien, Einsichten, dann für die menschlichen Interessen nutzbar gemacht werden. Software ist aber keine solche Wissenschaft. Software ist das Resultat von Aktivitäten im Rahmen des Software Engineering [7, 9]. Auf welcher Wissenschaft basiert dann das Software Engineering? Man könnte es mit „Algorithmen“ und „Algorithm Engineering“ versuchen, wenn diese Bezeichnung nicht schon für einen engen „Algorithmen“-Begriff stehen würde, und wenn es denn einen hinreichend umfassenden Algorithmenbegriff gäbe. Hierher gehört „Informatik“ als Wissenschaft, als Grundlage verschiedener Engineering-Disziplinen. Eine davon ist dann das „Software Engineering“.

Schluss

Mit diesem Text möchte ich Interesse wecken am Ziel einer umfassenden Theorie der Informatik. Vorbild einer solchen Theorie ist die Physik. Grundlegende Idee sind Vorschläge für formale Konzepte, die weit über Rechentechnik und Programmierung hinausgehen, mit einem Blickwinkel auch aus den Anwendungen und ihren Grenzen. Der Text gibt einige wenige Anregungen zur Entwicklung einer solchen umfassenden Theorie. Weitere Anregungen finden sich in der einschlägigen Literatur. Durch eine solche Theorie der Informatik werden bisherige Prinzipien der Informatik nicht obsolet; sie können aber besser eingeordnet und aufeinander bezogen werden, zusammen mit zukünftigen Engineering-Konzepten der Informatik.

Funding Open Access funding provided by Projekt DEAL.

Literatur

Verwendete Literatur

1. https://www.omg.org/spec/BPMN/2.0/. Zugegriffen: 9.3.2020
2. Agha G (1986) Actors: a model of concurrent computation in distributed systems. MIT Press, Cambridge, MA
3. Back R, Wright J (1998) Refinement calculus. Springer, New York
4. Bartocci E, Falcone Y, Francalanza A, Reger G (2018) Introduction to runtime verification. In: Lectures on runtime verification. Introductory and advanced topics. Lecture notes in computer science, Bd. 10457. Springer, Cham, S 1–33
5. Berry G, Boudol G (1990) The chemical abstract machine. In: Proc. ACM-POPL

6. Booch G, Rumbaugh J, Jacobson I (2005) The unified modeling language user guide, 2. Aufl. Addison Wesley, Boston
7. Brennecke A, Keil-Slawik R (1996) (Hrsg) Position Papers for Dagstuhl Seminar 9635 on History of Software Engineering
8. Brock DC (Hrsg) (2006) Understanding Moore's law: four decades of innovation. Chemical Heritage Foundation, Philadelphia
9. Broy M (1998) A logical basis for modular software and systems engineering. In: SOFSEM 98
10. Broy M, Stolen K (2001) Specification and development of interactive systems. Springer, New York
11. Carriero N, Gelernter D, Mattson T, Sherman A (1994) The Linda alternative to message-passing systems. Parallel Comput 20(4):633–655
12. Cleland CE (1993) Is the Church-Turing thesis true? Minds Mach 3(3):283–312
13. Cockshott P, Michaelson G (2007) Are there new models of computation? Reply to Wegner and Eberbach. Computer J 50(2):232–247. https://doi.org/10.1093/comjnl/bxl062
14. Dershowitz N, Falkovich E (2014) Generic parallel algorithms. In: Proceedings of computability in Europe. LNCS, Bd. 8493
15. Dijkstra EW (1990) Reasoning about programs. In: The distinguished lecture series, academic leaders in computer science and electrical engineering, Bd. III. University Video Communications, Stanford
16. Furia CA, Meyer B, Velder S (2014) Loop Invariants: analysis, classification, and examples. Acm Comput Suveys. https://doi.org/10.1145/2506375
17. van Glabbeek RJ (2001) The linear time – branching time spectrum I: the semantics of concrete, sequential processes. In: Handbook of process algebra. Elsevier, Amsterdam (chapter 1)
18. Gries D (1981) The science of programming. Springer, New York
19. Gurevich Y (2000) Sequential abstract state machines capture sequential algorithms. ACM Trans Comput Log. https://doi.org/10.1145/343369.343384
20. Hoare CAR (1969) An axiomatic basis for computer programming. CACM 12:576–583
21. Hoare T (2010) What can we learn from Edsger W. Dijkstra? Austin Texas, 12. Okt. 2010. Edsger W. Dijkstra Memorial Lecture
22. Knuth DE (1973) The art of computer programming Bd. 1. Addison-Wesley, Boston
23. Küster-Filipe J (2000) Fundamentals of a module logic for distributed object systems. J Funct Log Program 200(3):52–62
24. Lamport L (1978) Time, clocks, and the ordering of events in a distributed system. Commun ACM 21(7). https://doi.org/10.1145/359545.359563
25. Lamport L (2002) Specifying systems. Addison-Wesley, Boston
26. Livio M (2009) Is God a Mathematician? Simon & Schuster, New York
27. McCarthy J (1963) Towards a mathematical science of computation. In: Proc. IFIP Congress 62. North-Holland, Amsterdam, S 21
28. McConnell RM, Mehlhorn K, Näher S, Schweitzer P (2011) Certifying algorithms. Comput Sci Rev. https://doi.org/10.1016/j.cosrev.2010.09.009
29. Milner R (2005) Software science: from virtual to reality. Bull EATCS 87:12–16
30. Naur P, Randell B (Hrsg) (1969) Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany, 7–11 Oct. 1968. NATO Scientific Affairs Division, Brüssel
31. Reisig W (2013) Understanding Petri nets. Springer, Berlin, Heidelberg
32. Reisig W (2019) Associative composition of components with double-sided interfaces. Acta Inform. https://doi.org/10.1007/s00236-018-0328-7
33. Shepherdson JC (1995) Mechanisms for computing over arbitrary structures. The universal Turing-machine
34. Spivey JM (1992) The Z notation: a reference manual, 2. Aufl. International series in computer science. Prentice Hall, Upper Saddle River, New Jersey
35. Stein LA (2005) Interaction, computation, and education. In: Goldin D et al (Hrsg) Interactive computation – the new paradigm. Springer, Berlin, Heidelberg
36. Tel G (2000) Introduction to distributed algorithms, 2. Aufl. Cambridge University Press, Cambridge, UK
37. Tucker JV, Zucker JI (2000) Computable functions and semicomputable sets on many-sorted algebras. Handbook of logic in computer science. Oxford University Press, Oxford
38. Turing A (1937) On computable numbers, with an application to the Entscheidungsproblem. Proc Lond Math Soc 42. https://doi.org/10.1112/plms/s2-42.1.230
39. Vitanyi PMB (2005) Time, space, and energy in reversible computing. In: 2nd ACM conference on computing frontiers
40. Wegner P (1997) Why interaction is more powerful than algorithms. CACM 40:80–91
41. Wegner P, Eberbach E (2004) New Models of Computation. Comput J 47(1):4–9

Weiterführende Literatur

42. van Leuven J, Wiedermann J (2012) Computation as an unbounded process. Theor Comput Sci 429:202–212
43. van Leuven J, Wiedermann J (2013) Rethinking computation. In: Brown M, Erden Y (Hrsg) What is computation Proc. 6th AISB Symp on Computing and Philosophy AISB Convention 2013.
44. van Leuven J, Wiedermann J (2014) Computation as knowledge generation, with application to the observer-relativity problem. In: Brown M, Erden Y (Hrsg) Proc. 6th AISB Symp on Computing and Philosophy AISB 50 Convention, London
45. Moschovakis YN (2001) What is an Algorithm? In: Enquist B, Schmidt W (Hrsg) Mathematics unlimited – 2001 and beyond. Springer, Berlin, Heidelberg
46. Wing JM (2008) Computational thinking and thinking about computing. Phil Trans R Soc A 366:1881

Wolfgang Reisig studierte in Karlsruhe und Bonn Physik und Informatik. 1979 promovierte er an der RWTH Aachen zur Analyse kooperierenden sequentieller Prozesse. Nach Stationen an der Universität Hamburg, bei der Gesellschaft für Mathematik und Datenverarbeitung (GMD), sowie an der Technischen Universität München war er von 1993 bis 2015 Professor für Softwaretechnik und Theorie der Programmierung an der Humboldt-Universität zu Berlin. In dieser Zeit war er u. a. Senior Researcher am International Computer Science Institute (ICSI) in Berkeley und erhielt die „Lady Davis Visiting Professorship“ am Technion in Haifa, den Beta Chair der Technischen Universität Eindhoven und zweimal einen IBM Faculty Award für seine Beiträge zur Analyse von Geschäftsprozessen und Servicemodellen. Von 2009 bis 2017 war er Sprecher eines DFG-Graduiertenkollegs zu serviceorientierten Architekturen. Reisig ist Mitglied der Europäischen Akademie der Wissenschaften, Academia Europaea. Er hat zahlreiche Artikel und Bücher zu Petrinetzen publiziert und herausgegeben. Seit 1982 ist er Mitglied des Petri Net Conference Steering Committees und Mitherausgeber der Zeitschrift „Software and Systems Modeling“.