Modulare hierarchische Modellierung als Grundlage der Software- und Systementwicklung

Manfred Broy · Bernhard Rumpe

Einleitung

Der Entwurf komplexer Systeme ist eine der großen Herausforderungen für Ingenieure und aufgrund des damit immer komplexer werdenden Anteils an Software insbesondere auch für Informatiker. Heutige softwareintensive Systeme sind in aller Regel multifunktional, vielfältig vernetzt und in technische oder organisatorische Prozesse eingebettet. Die Komplexität der Entwicklung technischer Anwendungen hat unter Anderem folgende Ursachen:

- *Multifunktionalität*: Typischerweise dienen heutige Systeme nicht mehr nur einer spezifischen Funktion, sondern bieten über spezielle Nutzschnittstellen eine umfangreiche Familie von Funktionen an.
- *Mensch-Maschine-Schnittstelle*: Die Systeme umfassen oft mehrere Mensch-Maschine-Schnittstellen mit multimodalen Ein- und Ausgabedialogen.
- *Anpassbarkeit und Personalisierung*: Heutige Systeme müssen hochgradig nutzer-orientiert sein. Sie müssen von Anwendern auf deren spezifische Bedürfnisse personalisiert werden können und sich auf Basis des Anwenderverhaltens adaptiv optimieren. Die erforderliche Flexibilität kann grundsätzlich durch Software realisiert werden.
- *Einbettung in das Gesamtsystem*: Neben der Mensch-Maschine-Schnittstelle und der Einbettung in das Gesamtsystem mit anderen Geräten, Services oder dem Web ist das Informationsverarbeitungssystem über Sensoren und Aktuatoren mit seiner Umgebung verbunden und steuert und überwacht dabei technische Vorgänge.
- *Softwarearchitektur der Applikation*: Die eigentliche Applikation wird in der Regel durch umfangreiche Software realisiert; die Softwarearchitektur strukturiert dabei in der Regel Millionen LOC (Lines-of-Code) selbstgeschriebener und von Fremdherstellern zugekaufter und integrierter Software-Komponenten.
- *Systemplattform*: Die Applikationssoftware läuft in der Regel abgestützt auf einer Softwaresystemplattform bestehend aus Betriebssystem, Middleware, diagnostischen und anderen Services sowie Systemdiensten.
- *Hardwarearchitektur*: Die Software wird auf einer Hardwarearchitektur ausgeführt, die normalerweise aus einem Netzwerk von Prozessoren besteht, die über Bussysteme untereinander und mit Sensoren, Aktuatoren, den Mensch-Maschine-Schnittstellen und zumeist auch mit weiteren Systemen verbunden sind.
- *Deploymentabbildung*: Die Abbildung der Applikationssoftware auf die Hardware wird durch das Deployment bestimmt, das festlegt, welche Softwarekomponenten auf welchen Hardware-Prozessoren zur Ausführung kommen.
- *Dynamisierung des Deployments:* Die Aktualisierung der Software auf einem bereits laufenden verteilten System, die dynamische Relokation von Daten, Prozessen oder Aufgaben zur Lastverteilung und die Absicherung der Dynamik vor bösartigen Angriffen auf Daten oder Codes erfordern architekturelle Sicherheitsmaßnahmen.

Diese Zusammenstellung zeigt, wie komplex die Struktur heutiger Systeme ist. Um diese Komple-

DOI 10.1007/s00287-006-0124-6

Manfred Broy
Institut für Informatik, Technische Universität München,
80290 München, Deutschland
E-Mail: broy@in.tum.de
http://wwwbroy.informatik.tu-muenchen.de

Bernhard Rumpe
Technische Universität Braunschweig,
38106 Braunschweig, Deutschland
E-Mail: b.rumpe@tu-bs.de
http://www.sse.cs.tu-bs.de

Zusammenfassung:
Die Entwicklung komplexer eingebetteter Softwaresysteme, wie sie heute beispielsweise in Telekommunikationssystemen, Fahr- oder Flugzeugen oder mit der Steuersoftware von Automatisierungssystemen im Einsatz sind, erfordert ein strukturiertes, modulares Vorgehen und angemessene Techniken zur präzisen Beschreibung von Anforderungen, der Architektur des Systems mit ihren Komponenten, der Schnittstellen zur Systemumgebung und zwischen den internen Komponenten, der Wechselwirkung zwischen gesteuertem und steuerndem Teilsystem und schließlich der Implementierung. Mit dem frühzeitigen und durchgängigen Einsatz geeigneter Modelle (Stichwort UML („Unified Modeling Language") und MDA („Model Driven Architecture")) werden große Hoffnungen verbunden, die Entwicklungsaufgaben beherrschbarer zu gestalten.

Dieser Artikel beschreibt die theoretischen Grundlagen für ein konsequent modellbasiertes Vorgehen in Form eines zusammengehörigen, homogenen und dennoch modularen Baukastens von Modellen, der hierfür zwingend erforderlich ist. Besondere Schwerpunkte liegen hierbei auf den Themen

- Schnittstellen,
- Hierarchische Zerlegung,
- Architekturen durch Komposition und Dekomposition,
- Abstraktion durch Schichtenbildung,
- Realisierung durch Zustandsmaschinen,
- Verfeinerung von Hierarchie, Schnittstellen und Verhalten,
- Wechsel der Abstraktionsebenen und
- Integrierte Sicht auf die gesteuerten und steuernden Teilsysteme.

Dieser Baukasten der Modellierung muss wie bei allen anderen Ingenieursdisziplinen einer durchdachten, in sich stimmigen logisch-mathematischen Theorie entsprechen. Die hier vorgestellte Theorie besteht aus einem Satz von Notationen und Theoremen, die eine Basis für wissenschaftlich fundierte, werkzeugunterstützbare Methoden liefern und eine den Anwendungsdomänen (Stichwort Domänenspezifische Sprachen) pragmatisch angepasste Vorgehensweise ermöglicht.

Für eine wissenschaftlich abgesicherte Methode steht weniger die syntaktische Form der Modellierungssprache als vielmehr die Modellierungstheorie im Zentrum. Die Repräsentation von Modellen durch textuelle oder grafische Beschreibungsmittel ist ohne Zweifel eine wichtige Voraussetzung für den praktischen Einsatz von Modellierungstechniken, muss aber als komfortabler und grundsätzlich austauschbarer „Syntactic Sugar" gesehen werden.

xität in den Griff zu bekommen, bieten sich folgende Ansätze an:

- geeignete Modelle für alle wesentlichen Teilaspekte (Sichten),
- geeignetes Vorgehen,
- geeignete Werkzeuge und
- einheitliche geeignete Plattformen.

Dadurch können komplexe Entwicklungsaufgaben gegliedert und letztlich besser beherrscht werden („Divide et Impera"). Im Folgenden konzentrieren wir uns auf die Fragen eines angemessenen Satzes von aufeinander abgestimmten und in ein zielgerichtetes Vorgehen integrierbaren Modellen.

Modelle und Modellierung

Ein Modell hat nach [36] grundsätzlich

1. einen Zweck,
2. einen Bezug zu einem Original, und
3. abstrahiert bestimmte Eigenschaften des Originals.

Aufgrund der Immaterialität von Software, die bei einem weit gefassten Modellbegriff auch ein Modell von sich selbst bzw. den durch sie beschriebenen Abläufen ist, bestehen zwischen dem Original und seinen Modellen vielfältige Beziehungen, die sich auch durch den jeweiligen Einsatzzweck bemerkbar machen [34, 35]. So werden Modelle nicht nur zur Erringung eines substantiellen Anforderungs- bzw. Systemverständnisses, sondern häufig auch zur

Summary:
The development of complex, embedded software systems as used in telecommunication, cars, airplanes and automation systems requires structured, modular procedures and adequate techniques for a precise description of requirements, architectures and itstheir components, their interfaces to the systems' contexts as well as for internal interfaces, the interplayaction between control and controlled device, and finally for the implementation. Early and integrated use of models (like e.g. UML and MDA advocate) generates immense expectations to successfully control the development process.

This article describes the theoretical foundations of a strictly model based development in terms of an integrated, homogeneous, but yet modular construction kit for models, which is definitely needed for such an approach. Emphasis needs to be taken on

- Interfaces,
- Hierarchical decomposition,
- Architectures built through composition and decomposition,
- Abstraction through layering,
- Implementation through state machines,
- Refinement of hierarchy, interfaces and behavior,
- Alteration between abstraction viewpoints, and
- Integrated view on controlling and controlled system components.

In analogy to any other engineering discipline, this modeling constriction kit must correspond to a sound logical, mathematical theory. The theory introduced here consists of a set of notations and theorems and builds a solid basis for scientifically funded, tool-based methods and pragmatic, domain specific processes.

For a scientifically grounded method, the concrete syntactic form of a modeling language is far less important than the central modeling theory. A concrete representation of models through textual or graphical notations is undoubtedly an important prerequisite for the practical use of modeling techniques. However, it is only comfortable and replaceable "syntactic sugar".

konstruktiven Generierung von Code oder zur qualitätssichernden Ableitung von Tests eingesetzt [33]. Auch zu diesen Zwecken bietet eine durchdachte Modelltheorie grundsätzliche Vorteile, da so Entwickler ein einheitliches Modellverständnis entwickeln und Werkzeugketten besser aufeinander abgestimmt werden können.

Außerordentlich relevant für das Verständnis zur Modellbildung in der Informatik ist, dass sich die Modelle für softwareintensive Systeme grundsätzlich von den Modellen der klassischen Ingenieurwissenschaften unterscheiden (müssen), weil Softwaresysteme grundsätzlich andere Charakteristika aufweisen. Während klassische Ingenieurswissenschaften sich vor allem auf das klassische Integrations- und Differentiationskalkül stetiger Funktionen der Mathematik stützen, sind in der Informatik diskrete, „digitale" Modelle für verteilte, interagierende Systemkomponenten vorherrschend. Softwaresysteme besitzen eben nicht die kontinuierlichen Ein-/Ausgaben der Systeme der Regelungstechnik, sondern verarbeiten digitalisierte, diskrete Nachrichten, Ereignisse und Aktionen und stützen sich auf ebenfalls diskrete Zustände. Das Verhalten verteilter Softwaresysteme ist abhängig von diesen Zuständen und weist in den Zustandsübergängen keinerlei „Stetigkeit" sondern „Sprunghaftigkeit" auf.

Als Konsequenz der im Grundsatz nicht auf die Bedürfnisse der Informatik zugeschnittenen Ingenieursmathematik musste und muss die Informatik zwangsläufig eine eigene mathematische Theorie zur Modellbildung diskreter Systeme entwickeln. Dabei ist die Theorie diskreter Modelle – auch wenn bereits erste Modellierungsstandards verfügbar sind (siehe [40]) – immer noch in der Entwicklung begriffen und keineswegs abgeschlossen. Unvermeidlich sind die heute verfügbaren Standards, allen voran die UML, zu einer syntaktisch umfangreichen Sprache ausgeufert, ohne die fundamental ausgearbeitete, verstandene und trotz vieler Einzelansätze [6] fehlende allgemein anerkannte Semantik [22]. Einzelne, im Kern gut ausgearbeitete Theorien wie Entity/Relationship-Diagramme [15], der Relationenkalkül [16], Grammatiken und Automatentheorie sowie ihre hierarchische Erweiterung,

die Statecharts [21] wurden in der UML syntaktisch integriert und um zahlreiche syntaktische Elemente erweitert, ohne dabei eine theoretische Fundierung und die erforderliche semantische Integration zu leisten.

UML ist stark von praktischen Erfahrungen geprägt. Es ist Aufgabe der Wissenschaft, dazu die Grundlagen zu schaffen. Eine gute Theorie macht aber auch ein Satz von Theoremen aus, der die Fähigkeiten und Einschränkungen der Modellierungstheorie expliziert und ihre Anwendungsmöglichkeiten regelt. Dies ist in der UML bisher nur ausschnittsweise vorhanden, weshalb die UML vor allem zu informellen Dokumentionszwecken genutzt wird. Die Stärken einzelner in der UML enthaltener Theorien sind nicht integriert und daher auch nicht integriert nutzbar. Allerdings bleibt es das primäre Verdienst der UML, die Nützlichkeit von Modellbildung in Softwareentwicklungsprojekten einer breiteren Menge von Entwicklern sichtbar gemacht zu haben. Die Situation ist vergleichbar mit der Entwicklung der Programmierkonzepte, wo auch zunächst experimentiert wurde, um dann praktisch bewährte Konzepte mit einer allgemeingültigen Theorie zu unterfüttern, sodass Sprachkonzepte sich nun in unterschiedliche, aber sowohl syntaktisch als auch semantisch vergleichbaren Ausprägungen in allen gängigen Programmiersprachen wieder finden. Erst die Entwicklung semantischer Grundlagen hat nach der Exploration von Programmiersprachenkonzepten eine Konsolidierungsphase ermöglicht, die dann auch vertrauenswürdige, optimierende Übersetzer und viele weitere Werkzeuge hervorbringen konnte. Dazu ist es notwendig, sich von den engeren, weitgehend syntaktisch repräsentierten Ausprägungen der UML zu lösen und einen theoretischen Kern der Modellierung diskreter Systeme zu erarbeiten.

Es ist weiterhin eine fundamentale Aufgabe der Grundlagenforschung in der Informatik, die Entwicklung einer der Informatik angepassten, diskreten und zustandsbehafteten Modellbildungstheorie voran zu treiben. Eine solche Modellbildungstheorie ist notwendig, um beispielsweise optimierende und zuverlässige Übersetzer zwischen Sprachen zu entwickeln, Testfall- und Testüberdeckungskriterien zu definieren, die Migration zwischen (Versionen von) Modellierungssprachen und semantisch fundierte Interoperabilität auf einer integrierten Werkzeugbasis zu ermöglichen. Dies ist umso wichtiger, als mit den domänenspezifischen Sprachen [13] sonst eine Vielfalt syntaktischer Sprachen entstehen wird, die keinen sauberen semantischen Kern besitzen, damit nicht vergleichbar ist und deren Probleme sich bei Themen wie Interoperabilität, Integration oder sprachlicher und technischer Migration potenzieren.

In diesem Artikel beschreiben wir unsere Version einer möglichen Grundlage für eine solche Modellierungstheorie. Kapitel 2 gibt eine Übersicht über die Kernelemente einer solchen Theorie. Kapitel 3 beschreibt die wesentlichen Systemsichten und Modelle. In Kapitel 4 werden die Komposition und die Übergänge zwischen diesen Systemsichten beschrieben und in Kapitel 5 Beziehungen zwischen Systemmodellen diskutiert. Kapitel 6 gibt schließlich einen Ausblick zum methodischen Einsatz dieser Modellierungstheorie.

Theorie der digitalen Modellierung

Grafische Darstellungen und Modellierungssprachen für Systemmodelle sind in der Softwareentwicklung weit verbreitet. Allerdings sind für den Einsatz der digitalen Modellierung in der Systementwicklung nicht nur das reine „Zeichnen“ und die intuitive grafische Repräsentation digitaler Modelle von Bedeutung. Für methodisches Vorgehen und eine tief greifende Werkzeugunterstützung mit hohem Automatisierungsgrad ist eine ausgereifte Theorie der digitalen, von komplexen internen Zuständen abhängigen Modellierung eine unabdingbare Grundlage. Diese erlaubt eine genaue Festlegung der Bedeutung der verwendeten Beschreibungen und wie diese zueinander in Beziehung gesetzt werden.

Typischerweise wird die digitale Modellierung zur Darstellung markanter Sichten eingesetzt. Um für ein komplexes System die Modellierung auf bestimmte Gesichtspunkte konzentrieren zu können, werden Modellperspektiven entwickelt wie:

- Datenmodell,
- Zustandsmodell,
- Schnittstellenmodell,
- Ablaufmodell und
- (physisches und logisches) Strukturmodell.

Für jede dieser Sichten sind intuitiv ansprechende Kombinationen aus Diagrammen und textuellen

Annotationen zu bilden, die das schnelle Verstehen von Systembeschreibungen garantieren und präzise Modellbildungen erlauben. Auf Basis einer übergreifenden, sauber fundierten und umfassend verstandenen Theorien können und müssen die so dargestellten Modelle zueinander in Beziehung gesetzt werden, sodass ein zusammenhängendes Verständnis in einem konsistenten Modell des Systems entsteht. Dazu müssen in der Theorie mathematisch-logische Repräsentationen der Modelle entwickelt und in entwicklungstechnisch nutzbare mathematische Zusammenhänge gebracht werden.

Die Theorie der digitalen Modelle und der digitalen Modellierung umfasst daher neben den mathematischen Theorien für die einzelnen Systemsichten insbesondere auch Abbildungen und Relationen, die das Zusammenspiel und den Übergang zwischen den Systemsichten erklären und die Entwicklungsschritte der Systeme formal erfassen.

In den nachfolgenden Kapiteln wird ein Ansatz für eine solche Theorie in Form eines Systemmodells skizziert, der aus allen Formen zur Sichtenbildung die markanten, homogen kombinierbaren Ansätze zunächst isoliert darstellt und schließlich zu einem Modellierungsbaukasten integriert. Dieses Systemmodell kann als Erweiterung von [25] und Kernergebnissen aus [11] verstanden werden.

Die entwickelte Theorie digitaler Modellierung ist über die letzten 15 Jahre entstanden und gereift. Alternative Ansätze bieten meist nur Bestandteile einer solchen Theorie an. Dazu gehören die bereits erwähnten strukturellen bzw. rein verhaltensorientierten Modelle, die syntaktisch in die UML integriert sind, aber auch algebraische Spezifikationssprachen ([41], CASL [42]) oder die damit verwandten die Logiksprachen (B [43], Z [44] oder HOL [45]). Diese Sprachen besitzen zwar große Allgemeinheit, um damit Modelle zu formulieren, eine Theorie digitaler, verteilter Systeme wäre darin erst zu definieren. Weitere Klassen von Theorien konzentrieren sich jeweils auf einen Aspekt, etwa auf die Interaktion (CCS [46] und CSP [47]) und gewisse dynamische Effekte (π-Kalkül [48]). Die ASM-Methode [49] ist demgegenüber eine relativ vollständige Beschreibungstechnik, bietet aber ebenfalls keine Möglichkeit zwischen Black- und Whitebox-Sichten zu wechseln. In der dargebotenen Form ist die nachfolgend dargestellte Theorie digitaler Modelle am umfassendsten und bietet die beste semantische Integration verschiedener Sichten.

Die klassischen Systemsichten und -modelle

In diesem Abschnitt beschreiben wir wesentliche Systemsichten und -modelle, die bei der Beschreibung von Systemen typischerweise auftreten. Wir verzichten dabei darauf, eine Syntax im Sinn einer Beschreibungssprache einzuführen und konzentrieren uns auf die mathematische Darstellung der Modellkonzepte.

Datenmodell

Datenmodelle beschreiben ein System von Datentypen und legen darauf operierende Funktionen fest. Wir nutzen für die Theorie zweckmäßigerweise das Konzept der *Algebraischen Spezifikation* (vgl. [4, 17, 38]), das in der Informatik inzwischen gut verstanden und in verschiedenen, konzeptuell abgespeckten Ausprägungen (insbesondere OCL [40] oder dem in Klassen integrierten Modulkonzept der Objektorientierung) im Einsatz ist. Bei datenintensiven Anwendungen ist die Nutzung von *Entity/Relationship-Diagrammen* [15] für den Datenstrukturanteil hilfreich, wobei E/R-Diagramme als notationelle Variante algebraischer Datentypen angesehen werden können.

Für unsere weiteren Betrachtungen benötigen wir lediglich die Vorstellung, dass ein Datenmodell eine Menge von Typen (oder „Sorten“) als Bezeichnungen für Datenmengen und eine Menge typisierte Funktions- und Operationssymbole als Bezeichnungen für Funktionen vorgibt. Die Wirkungsweise der Funktionen wird etwa durch eine eigenschaftsorientierte, oft algebraische Spezifikation festgelegt.

Die Theorie der Datenmodellierung ist mittlerweile gut verstanden und befindet sich in einem stabilen und nutzbaren Stand.

Zustandsmodell

Ein fundamentales Modell zur Beschreibung von Systemen sind Zustandsmaschinen. Sie können sowohl zur Beschreibung eines Systems, wie zur Beschreibung von Teilsystemen, den Komponenten eines Systems, genutzt werden. Wir wählen (exemplarisch) aus den vielen existierenden Varianten von Systemen *Zustandsmaschinen mit Ein- und Ausgabe*, sogenannte Mealy/Moore-Automaten, die jedoch auf unendliche Mengen von Zuständen und Ein-/Ausgaben erweitert werden [19, 20, 32]:

$$\Delta : (\mathrm{STATE} \times \mathrm{INPUT}) \rightarrow \mathcal{P}(\mathrm{STATE} \times \mathrm{OUTPUT}).$$

Hier ist STATE ein (interner, gekapselter) Zustandsraum der mathematisch eine Menge von Zuständen repräsentiert. Eine einfache Methode zur Definition eines Zustandsraums ist die Definition einer Familie von Zustandsattributen (Variablen) mit festgelegten Typen, deren jeweilige Belegungen einen Zustand charakterisieren. Dabei handelt es sich wieder um ein Datenmodell, das mit den Mitteln der Datenmodellierung beschrieben werden kann.

Genau genommen nehmen wir an, dass es sich bei den betrachteten Zustandsmaschinen um auf unendliche Zustandsräume verallgemeinerte Moore-Automaten handelt. Die Ausgabe hängt dabei nur vom jeweiligen Zustand, nicht aber von der aktuellen Eingabe, ab. Dies wird durch folgende Bedingung an die Zustandsübergangsrelation erfasst:

$$(\sigma', y) \in \Delta(\sigma, x) \Rightarrow \forall x' : \exists \sigma'' : (\sigma'', y) \in \Delta(\sigma, x').$$

Diese Eigenschaft eines Moore-Automaten modelliert den Umstand, dass jede Eingabe Zeit benötigt, um verarbeitet zu werden, und sich deshalb frühestens im drauffolgenden Schritt der Zustandsmaschine auf die Ausgabe auswirken kann.

Eine Zustandsmaschine beschreibt Zustandsübergänge. Für jede Eingabe $x \in$ INPUT und jeden Zustand $\sigma \in$ State erhalten wir durch $(\sigma', y) \in \Delta(\sigma, x)$ einen Nachfolgezustand $\sigma' \in$ State und eine Ausgabe $y \in$ OUTPUT. Der Automat ist im automaten-theoretischen Sinne „nichtdeterministisch", weil immer eine Menge an Paaren von Folgezuständen und Ausgaben zugeordnet werden kann. Für Zwecke der Spezifikation nutzen wir dies als Unterspezifikation, die der später festzulegenden Implementierung noch Alternativen offen lässt.

Im einfachsten Fall sind Eingabe und Ausgabe atomare Ereignisse. Wir sind jedoch an Zustandsmaschinen interessiert, die über eine typisierte Ein-/Ausgabeschnittstelle zum Austausch von Nachrichten verfügen. Wir betrachten deshalb Automaten, die über eine Menge von Eingabekanälen Eingaben erhalten und über eine Menge von Ausgabekanälen Ausgaben erzeugen. Wir lassen dabei zu, dass über jeden Kanal in jedem Zustandsübergang eine endliche Sequenz von Nachrichten fließt. Natürlich kann die Sequenz auch leer sein.

Durch diese Einbeziehung von Ein- und Ausgabekanälen erhalten wir einen differenzierten Begriff einer syntaktischen Schnittstelle für Zustandsmaschinen. Abbildung 1 zeigt ein einfaches Beispiel für ein System mit je drei typisierten Ein- und Ausgabekanälen.

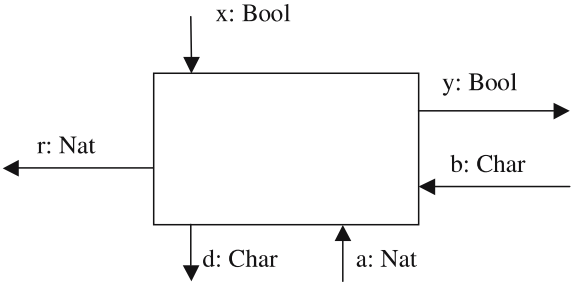

Abb. 1 Beispiel für ein System mit typisierten Ein- und Ausgabekanälen

Ein- und Ausgaben fließen also wie in verteilten Systemen üblich über gerichtete Kanäle. Dies gilt auch, wenn die dann als „logische" Kanäle dargestellten Kommunikationsverbindungen durch Middleware wie CORBA oder Busstrukturen realisiert werden, wie etwa VPN via Internet. Zur Modellierung verwenden wir Mengen von *Kanälen* C (oft auch Ports genannt, [31]), wobei für jeden Kanal ein Datentyp festgelegt wird. Der Datentyp legt fest, welche Datenwerte über den Kanal gesendet werden.

In jedem Schritt der Zustandsmaschine wird über jeden Kanal eine endliche (also gegebenenfalls auch leere) Sequenz von Werten übermittelt. Dabei besteht bei der Wahl des Schrittmodells, anschaulich des Zeittakts, der die Granularität, den betrachteten Zeitraum, der Übergänge festlegt, Flexibilität. Wählt man eine grobe Granularität und damit einen langen Zeittakt, so erhält man in jedem Zustandsübergang pro Zeittakt lange Ein- und Ausgabesequenzen auf den Kanälen. Wählt man eine feine Granularität und damit einen kurzen Zeittakt, so sind die Sequenzen meist leer oder enthalten nur eine oder sehr wenige Nachrichten.

Weil C eine Menge von Kanälen darstellt, ist jedes Element $x \in$ INPUT bzw. $x \in$ OUTPUT formal eine Abbildung, die jedem Kanal in $c \in C$ eine endliche Sequenz $x(c) \in$ UM* zuordnet, wobei wir mit UM das Universum aller betrachteter Nachrichten bezeichnen. Präziser gesagt lassen sich die Nachrichten sogar typisieren: Wir fordern $x(c) \in M_c^*$ entsprechend dem Typ $M_c \subseteq$ UM des Kanals c. Mit SEQ(C) bezeichnen wir für eine Menge von Kanälen C alle Belegungen der Kanäle mit Sequenzen von Nachrichten, wobei wir annehmen, dass

diese Nachrichten den Typisierungen der Kanäle entsprechen.

Mit $\mathrm{In}(\Delta)$ bezeichnen wir die Menge der typisierten Eingabekanäle von Δ und mit $\mathrm{Out}(\Delta)$ bezeichnen wir die Menge der typisierten Ausgabekanäle einer Zustandsmaschine Δ.

Zusätzlich zur Übergangsfunktion ist für eine Zustandsmaschine ein Anfangszustand $\sigma_0 \in \mathrm{STATE}$ festzulegen. Damit beschreibt das Paar (Δ, σ_0) ein (nicht notwendig deterministisches) Systemverhalten. Mit $\mathbb{M}$ bezeichnen wir die Menge aller Zustandsmaschinen (Δ, σ_0) mit gegebenen Anfangszustand σ_0, die wie oben beschrieben verallgemeinerte Moore-Automaten sind.

Eine Zustandsmaschine Δ heißt *total*, wenn

$$\Delta(\sigma, x) \neq \emptyset$$

für alle Zustände σ und alle Eingaben x. Andernfalls heißt Δ *partiell*.

Schnittstellen

Das Konzept der *Schnittstelle* ist essentiell für die gezielte Abstraktion von Systemen sowie Teilsystemen und damit die Beherrschung umfangreicher Systeme der Informatik im Rahmen von Architekturen. Die Idee der Schnittstelle erlaubt eine entscheidende Abstraktion von der Implementierung zur reinen Darstellung der Wirkung eines Systemteils (anschaulich spricht man von „Black Box Sicht“ oder dem dahinter stehenden Prinzip des „Information Hiding“). Wir führen ein sehr elementares Verhaltensmodell für Kommunikationsschnittstellen ein. Dafür verwenden wir das Konzept der Datenströme, die kommunizierte Datenwerte und ihre kausalen und zeitlichen Abhängigkeiten beschreiben können. Später definieren wir dafür explizit Abstraktionen.

Ein *Datenstrom* (synonym Nachrichtenstrom) eines gegebenen Datentyps mit Datenelementen aus der Menge M ist eine unendliche Folge von Datensequenzen [9]:

$$s : \mathbb{N}_+ \to M^*.$$

Hier bezeichnet $\mathbb{N}_+$ die echt positiven ganzen Zahlen, also die natürliche Zahlen ohne die Null. Datenströme dienen in Schnittstellen der Darstellung der übertragenen Nachrichten. Für jede Zahl $t \in \mathbb{N}_+$ gezeichnet $s(t)$ die Sequenz von Nachrichten, die im Zeitintervall t übertragen werden. Wir verwenden hier ein einfaches aber hinreichend ausdruckstarkes *Zeitmodell*, das innerhalb von Zeiteinheiten nur Reihenfolgen der auftretenden Nachrichten sichert. Die Menge der echt positiven natürlichen Zahlen $\mathbb{N}_+$ bezeichnet die Zeitintervalle. Die Zeit ist somit in eine unendliche Folge gleichlanger Zeitintervalle eingeteilt. Wir arbeiten im Modell mit einer diskreten, globalen Zeit.

Als Operatoren steht uns unter anderem das „vorne anfügen“ zur Verfügung: $a\&s$ fügt am Anfang des Stroms s eine Sequenz $a \in M^*$ an. Es gilt:

$$(a\&s)(1) = a;$$
$$(a\&s)(n) = s(n-1) \text{ für } n > 1.$$

Sei C eine der oben beschriebenen Menge typisierter Kanäle, dann ist eine *Kanalbelegung* x von der Form

$$x : C \to (\mathbb{N}_+ \to \mathrm{UM}^*)$$

eine Zuordnung („Belegung“) von Strömen zu Kanälen. Dann ist $x(c)$ ein Datenstrom für jeden Kanal in $c \in C$ mit dem entsprechenden Typ der Daten aus der Menge M_c. Wir bezeichnen die Menge aller Belegungen für die Menge C von typisierten Kanälen mit $\mathbb{H}(C)$ (für „History“) oder auch in Kurzform mit $\vec{C}$.

Die Operatoren für Nachrichtenströme können auf Kanalbelegungen erweitert („geliftet“) werden. Beispielsweise fügt $a\&x$ vorne eine Belegung von Kanälen durch Sequenzen $a : C \to \mathrm{UM}^*$ an die Kanalbelegung x an. Es gilt $(a\&x)(c) = a(c)\&x(c)$ für jeden Kanal $c \in C$.

Nun wenden wir uns der Modellierung des *Schnittstellenverhaltens* von Komponenten zu. Seien I und O jeweils Mengen von typisierten Kanälen. Mit $(I \blacktriangleright O)$ bezeichnen wir die syntaktische Schnittstelle einer Komponenten mit Eingabekanälen aus I und Ausgabekanälen aus O.

Das Schnittstellenverhalten einer Komponente mit syntaktischer Schnittstelle $(I \blacktriangleright O)$ modellieren wir durch Funktionen der Form

$$F : \vec{I} \to \wp(\vec{O}).$$

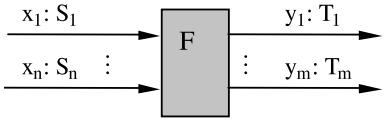

Abb. 2 Grafische Darstellung der Funktion F als Datenstromkomponente

Wie bereits bei Automaten wird die Mengenwertigkeit zur Darstellung von Alternativen (Unterspezifikation) eingesetzt. Bei diesen Funktionen F setzen wir voraus, dass die folgende Eigenschaft erfüllt ist, die ein angemessenes Verhalten in Hinblick auf den Zeitfluss sicherstellt (für alle Belegungen $x, z \in \vec{I}$ der Eingabekanäle, wobei $x{\downarrow}t$ die Einschränkung des Ströme in der Belegung x auf die ersten t Intervalle für $t \in \mathbb{N}$ bezeichnet):

$$x{\downarrow}t = z{\downarrow}t \Rightarrow \{y{\downarrow}t+1 : y \in F(x)\} = \{y{\downarrow}t+1 : y \in F(z)\}.$$

Diese Eigenschaft heißt *starke Kausalität*. Sie stellt sicher, dass im Modell ein, der Realität entsprechender konsistenter Zeitfluss gegeben ist. Eine Komponente kann demnach auf eine Eingabe erst reagieren, nachdem die Eingabe eingetroffen ist (eine ausführliche Diskussion findet sich in [9]). Man beachte, dass in Analogie zur Eigenschaft der Moore-Automaten, auch hier die aktuelle Ausgabe zum Zeitpunkt t nicht von der aktuellen Eingabe zum Zeitpunkt t (sondern nur von den Eingaben zu den Zeitpunkten $< t$) abhängt.

Die Funktion F modelliert ein Schnittstellenverhalten in höchst abstrakter Form. Für jede Belegung x der Eingabekanäle werden durch $F(x)$ die möglichen Belegungen der Ausgabekanäle festgelegt. Ist $F(x)$ stets einelementig, so heißt F *deterministisch*. Ist $F(x)$ für wenigstens eine Eingabehistorie $x \in \vec{I}$ die leere Menge, dann können wir aus der starken Kausalität folgern, dass $F(x)$ für alle Eingabeströme leer ist. Dann heißt F *paradox*.

Mit $\mathrm{In}(F)$ bezeichnen wir die Menge I der Eingabekanäle von F und mit $\mathrm{Out}(F)$ bezeichnen wir die Menge O der Ausgabekanäle von F. Mit $\mathbb{F}$ bezeichnen wir die Menge aller solchen Funktionen, die das Gesetz der starken Kausalität erfüllen.

Logische Architekturen aus Komponenten

Architekturen beschreiben Sichten auf die interne Strukturierung eines Systems [23]. Eine Komponentenarchitektur besteht aus einer Familie von Komponenten („Teilsystemen“), die über Kanäle untereinander und mit ihrer Umgebung verbunden sind. Die Komponenten selbst sind wieder Systeme und können durch ihr Schnittstellenverhalten oder durch Zustandsmaschinen beschrieben sein.

Sei $\mathbb{K}$ die Menge aller Identifikatoren für Komponenten. Formal ist eine Komponentenarchitektur Ψ durch eine endliche Menge $K \subseteq \mathbb{K}$ von Identifikatoren für Komponenten gegeben und eine Zuordnung von Verhalten zu den Identifikatoren der Komponenten [29, 30]. Das Verhalten kann durch eine Zustandsmaschine oder durch Schnittstellenverhalten angegeben werden.

$$\Psi : K \to \mathbb{F} \cup \mathbb{M}.$$

Jede Komponente mit Identifikator $k \in K$ hat eine festgelegte Menge $\mathrm{In}(\Psi(k))$ von Eingabe- und Ausgabekanälen $\mathrm{Out}(\Psi(k))$. Wir nehmen der Einfachheit halber an, dass für alle Komponenten einer Architektur die Mengen der Ausgabekanäle paarweise disjunkt sind. Formal heißt das, dass für alle $k, j \in K$ mit $k \neq j$:

$$\mathrm{Out}(\Psi(k)) \cap \mathrm{Out}(\Psi(j)) = \emptyset.$$

Dadurch wird festgelegt, dass jeder Kanal genau eine Komponente hat, von der er ausgeht (seine Quelle). Mit anderen Worten, es existieren keine Bezeichnungskonflikte bei den Ausgabekanälen. Die Vereinigung

$$\mathrm{Chan}(\Psi) = \bigcup_{k \in K} \mathrm{In}(\Psi(k)) \cup \bigcup_{k \in K} \mathrm{Out}(\Psi(k))$$

über die Ein- und Ausgabekanäle aller Komponenten bildet die Menge aller Kanäle der Architektur.

Es gibt drei Arten von Kanälen in einer Architektur: (1) Ein Eingabekanal der Architektur wird von wenigstens einer Komponente gelesen, aber von keiner Komponente befüllt:

$$\mathrm{In}(\Psi) = \mathrm{Chan}(\Psi) \setminus \bigcup_{k \in K} \mathrm{Out}(\Psi(k)).$$

(2) Tritt ein Kanal in einer Komponente als Ausgabekanal auf, so kann er sowohl in einer anderen Komponente als Eingabe, als auch als Ausgabekanal der Architektur definiert werden. Die Architektur legt selbst fest, welche Kanäle sie als Schnittstelle $\mathrm{Out}(\Psi)$ nach außen frei gibt. Die Rahmenbedingung ist jedoch:

$$\mathrm{Out}(\Psi) \subseteq \bigcup_{k \in K} \mathrm{Out}(\Psi(k)).$$

(3) Die Menge der internen Kanäle der Komposition ist damit gegeben durch:

$$\mathrm{Internal}(\Psi) = \mathrm{Chan}(\Psi) \setminus \mathrm{In}(\Psi) \setminus \mathrm{Out}(\Psi)$$

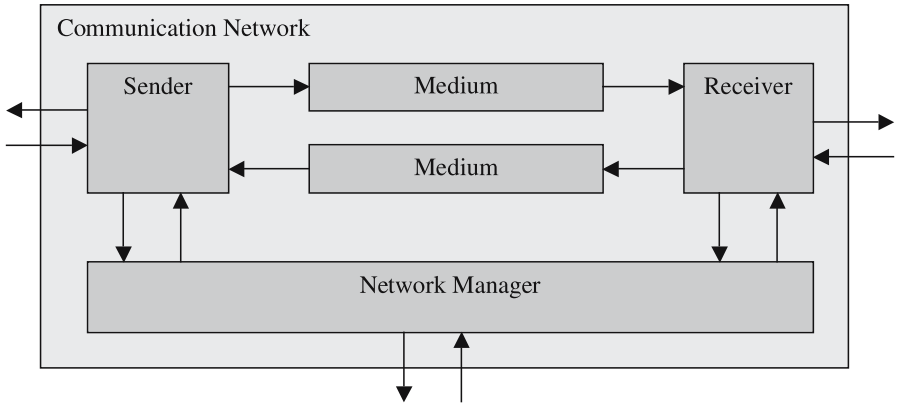

Abb. 3 Beispiel für eine Architektur

Tritt ein Kanal in einer Komponente als Ausgabekanal und in einer anderen als Eingabekanal auf, so verbindet der Kanal die Komponenten durch eine Kommunikationsverbindung. Kanäle können von mehreren Komponenten gelesen oder gar nicht genutzt werden. Nachfolgend gehen wir vereinfachend davon aus, dass genau die intern genutzten Kanäle nicht nach außen geführt werden, also

$$\mathrm{Out}(\Psi) = \mathrm{Chan}(\Psi) \setminus \bigcup_{k \in K} \mathrm{In}(\Psi(k)).$$

Abbildung 3 zeigt ein einfaches Beispiel für eine Architektur (ohne Angabe der Kanalnamen oder deren Typen).

Ist für die Komponenten einer Architektur ein Verhalten in Form einer Schnittstelle oder einer Zustandsmaschine spezifiziert, so sprechen wir von einer ausgearbeiteten Architektur. Andernfalls ist diese während der Entwicklung noch weiter zu präzisieren.

Neben Zustandsmaschinen und Schnittstellenverhalten bieten die Architekturen eine dritte Möglichkeit Systeme zu modellieren. Im Gegensatz zu Zustandsmaschinen und unserem Schnittstellenmodell, das abstrakte Verhalten im Sinne einer „Black Box"-Sicht beschreibt, zielen Architekturen darauf ab, die Struktur also den inneren Aufbau eines Systems darzustellen.

Hierarchisches Systemmodell

Mit den drei Arten von Modellen für Systeme und ihrer einheitlichen Festlegung der syntaktischen Schnittstellen lassen sich nun systematisch hierarchische Systeme bilden. Auf der untersten Ebene sind Systemmodelle durch Schnittstellenmodelle oder Zustandsmaschinen gegeben.

$\mathbb{A}_0 = \mathbb{F} \cup \mathbb{M}$.

Dadurch ist die Menge aller nicht weiter zerlegten Zustandsmaschinen und Schnittstellenmodelle gegeben. Wir definieren induktiv Systemarchitekturen der Hierarchie durch:

$$\mathbb{A}_{i+1} = \mathbb{A}_i \cup \{\Psi : K \to \mathbb{A}_i : K \subseteq \mathbb{K}\},$$

wobei bzgl. der Nutzung von Komponentennamen und Kanälen einige offensichtlich bereits beschriebene (und deshalb hier nicht weiter dargestellte) Randbedingungen gelten.

Sei $\mathbb{A}$ die Menge aller *hierarchischen Architekturen* von Systemen beliebiger Höhe. Die Menge $\mathbb{A}$ umfasst alle eingeführten Spielarten von Systemmodellen und erlaubt somit eine flexible Darstellung komplexer hierarchischer Systeme.

Komposition von und Übergänge zwischen Systemsichten

Soweit haben wir nur verschiedene Arten von Systemsichten eingeführt. In diesem Abschnitt definieren wir die Kompositionsoperatoren und die Abstraktion zur Beschreibung der Übergänge in der Form von Abbildungen zwischen diesen Systemsichten, die den methodischen Umgang zwischen den Sichten bei der Entwicklung, beim Re-Engineering, bei der Analyse der Konsistenz von Sichten oder der Verifikation und Validierung erlauben.

Komposition von Systemen

Ein essentielles Konzept in der methodischen Entwicklung komplexer Systeme ist die Komposition, also die Zusammensetzung, von kleineren (Sub-)Systemen und Komponenten zu größeren

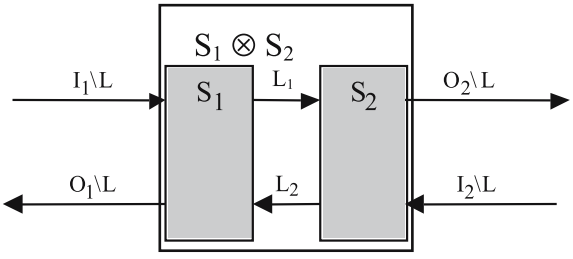

Abb. 4 Komposition $S_1 \otimes S_2$

Systemen. Wir zeigen die Komposition für alle Systemformen, die wir eingeführt haben. Dabei setzen wir in diesem Abschnitt stets zwei Systeme S_1 und S_2 mit den Signaturen $(I_1 \blacktriangleright O_1)$ bzw. $(I_2 \blacktriangleright O_2)$ voraus. Dabei seien die Bezeichnungen der Kanäle und Systeme konfliktfrei, also die Mengen O_1 und O_2 der Ausgabekanäle disjunkt. Nach Vorgabe haben übereinstimmende Kanalidentifikatoren identische Typisierungen.

Wenn wir die Systeme S_1 und S_2 mit den beschriebenen syntaktischen Schnittstellen zu dem System $S_1 \otimes S_2$ mit der entstehenden Schnittstelle $(I \blacktriangleright O)$ zusammensetzen, verbinden wir die Ausgabekanäle mit Eingabekanälen mit gleichen Kanalidentifikatoren in internen Kanalverbindungen. Die Menge der internen Verbindungen L ergibt sich dabei wie folgt:

$$L = L_1 \cup L_2 \quad \text{wobei} \quad L_1 = (I_2 \cap O_1),\ \ L_2 = (I_1 \cap O_2).$$

Die Menge der Ein- und Ausgabekanäle für die durch Komposition entstehende Komponente erhalten wir wie folgt:

$$\text{In}(S_1 \otimes S_2) = I = (I_1 \cup I_2)\backslash L,$$
$$\text{Out}(S_1 \otimes S_2) = O = (O_1 \cup O_2)\backslash L.$$

Dies definiert die syntaktische Schnittstelle gegeben durch die Menge der Ein-/Ausgabekanäle der durch Komposition entstehenden Komponente. Man beachte, dass in der Komposition Kommunikationsverbindungen zwischen den komponierten Komponenten entstehen, wobei noch Rückkloppungsschleifen entstehen können.

Komposition von Zustandsmaschinen

In diesem Abschnitt zeigen wir, wie Zustandsmaschinen zusammengesetzt, „komponiert" werden. Seien zwei Zustandsmaschinen $(j = 1, 2)$:

$$\Delta_j : (\text{STATE}_j \times \text{SEQ}(I_j)) \to \wp(\text{STATE}_j \times \text{SEQ}(O_j))$$

mit Anfangszuständen $\sigma 0_j \in \text{STATE}_j$ gegeben. Durch die Komposition der Übergangsfunktionen Δ_1 und Δ_2 erhalten wir die Übergangsfunktion:

$$\Delta : (\text{STATE} \times \text{SEQ}(I)) \to \wp(\text{STATE} \times \text{SEQ}(O))$$

definiert durch $\Delta = \Delta_1 \otimes \Delta_2$. Als Zustandsmenge der durch Komposition entstehenden Komponente erhalten wir das Kreuzprodukt der Zustände

$$\text{STATE} = \text{STATE}_1 \times \text{STATE}_2$$

mit Anfangszustand $(\sigma 0_1, \sigma 0_2)$, die Signatur der Maschine Δ ist wie oben beschrieben und die wird definiert durch die Gleichung

$$\Delta((\sigma_1, \sigma_2), x) = \{((\sigma_1', \sigma_2'), z|O) : \exists z \in \text{SEQ}(I \cup O \cup L) :$$
$$z|I = x \wedge (\sigma_j', z|O_j) \in \Delta_j(\sigma_j, z|I_j) \text{ für } j = 1, 2\}.$$

Wir definieren für die Maschinen $M_i = (\Delta_i, \sigma 0_i)$ für $i = 1, 2$ die Komposition $M_1 \otimes M_2$ durch die Gleichung

$$M_1 \otimes M_2 = (\Delta_1 \otimes \Delta_2, (\sigma 0_1, \sigma 0_2)).$$

Da wir auf unendliche Zustandsräume verallgemeinerte Moore-Maschinen betrachten und deshalb die Ausgabe nur von dem Zustand abhängt, ist die Definition auch bei Rückkopplung konsistent und insbesondere die Komposition totaler Zustandsmaschinen wieder total [20]. Hier zahlt sich die Festlegung auf Moore-Maschinen aus.

Komposition von Schnittstellenverhalten

Analog zur Komposition von Zustandsmaschinen können wir Schnittstellenverhalten komponieren. Seien folgende Schnittstellenverhalten gegeben:

$$F_1 : \vec{I}_1 \to \wp(\vec{O}_1), \quad F_2 : \vec{I}_2 \to \wp(\vec{O}_2).$$

Wir definieren die parallele Komposition mit Rückkopplung mit der oben beschriebenen Signatur des komponierten Systems wie folgt:

$$F_1 \otimes F_2 : \vec{I} \to \wp(\vec{O})$$

spezifiziert durch:

$$(F_1 \otimes F_2).x = \{z|O : \exists z \in \mathbb{H}(I \cup O \cup L) : z|I$$
$$= x \wedge z|O_j \in F_j(z|I_j)$$
$$\text{für } j = 1, 2\}.$$

Durch die starke Kausalität reicht diese einfache Gleichung aus, um die Komposition präzise zu definieren. Insbesondere ist $F_1 \otimes F_2$ nur genau dann paradox, wenn es F_1 oder F_2 sind.

Komposition von Architekturen

Die Komposition zweier Architekturen der Form $(j = 1, 2)$

$$\Psi_j : K_j \to \mathbb{A}$$

ist einfach. Als Kontextbedingung fordern wir, dass die Mengen K_1 und K_2 der Komponentenidentifikatoren disjunkt sind und auch für die Kanalnamen keine Bezeichnungskonflikte existieren. Wir definieren die Architektur

$$\Psi_1 \otimes \Psi_2 : (K_1 \cup K_2) \to \mathbb{A}$$

durch $(j = 1, 2)$ die Gleichung

$$(\Psi_1 \otimes \Psi_2)(k) = \Psi_j(k) \text{ falls } k \in K_j.$$

Die Architektur des zusammengesetzten Systems besteht aus der Menge aller Komponenten beider Systeme.

Eigenschaften der Komposition

Die dargestellten Kompositionsoperatoren sind jeweils zweistellig. Durch die paarweise Komposition entstehen jeweils wieder Systeme, die weiter komponierbar sind. Darüber hinaus sind die Kompositionsoperatoren alle so definiert, dass die Reihenfolge der Komposition keine Rolle spielt, denn die Kompositionsoperatoren sind kommutativ und assoziativ (solange keine Namenskonflikte bei den Ausgabekanälen auftreten). Dadurch ist die Reihenfolge der Kompositionen irrelevant und das Ergebnis immer eindeutig festgelegt.

Die Komposition kann damit induktiv auf nichtleere Mengen von Komponenten in Architekturen, Zustandsmaschinen bzw. Schnittstellenverhalten Z ausgedehnt werden:

$$\bigotimes\{z\} = z$$
$$\bigotimes Z = z \otimes \left(\bigotimes(Z \backslash \{z\})\right) \quad \text{für beliebiges } z \in Z.$$

Wir können somit beliebige endliche (unter gewissen Voraussetzungen sogar unendliche) Mengen von Komponenten ohne Namenskonflikte komponieren, solange die Komponenten in derselben Systemsicht definiert sind. Auch deshalb sind Übergänge zwischen Systemsichten notwendig.

Übergang zwischen Systemsichten

In diesem Abschnitt betrachten wir Übergänge zwischen den durch $\mathbb{A}$ eingeführten Systemsichten, um damit auch heterogen definierte Komponenten zur komponieren. Folgende Abbildungen werden dazu eingeführt:

$Z2S : \mathbb{M} \to \mathbb{F}$	überführt einen Zustandsmaschine in ein Schnittstellenverhalten,
$A2S : \mathbb{A} \to \mathbb{F}$	überführt eine Architektur in ein Schnittstellenverhalten,
$S2Z : \mathbb{F} \to \mathbb{M}$	überführt ein Schnittstellenverhalten in eine Zustandsmaschine,
$A2Z : \mathbb{A} \to \mathbb{M}$	überführt eine Architektur in eine Zustandsmaschine.

Diese Übergänge verbinden die heterogen definierten Systemsichten zu konsistenten Systembeschreibungen. Die Übergänge von der Zustandssicht oder der Architektursicht in die Schnittstellensicht ($Z2S$ und $A2S$) beinhalten eine Abstraktion. Wir sprechen von *Schnittstellenabstraktion*. Die Umkehrung dieser Abstraktion ist in der Regel keine eindeutige Abbildung – zu jeder Schnittstellensicht gibt es viele Zustands- oder Architektursichten. Bei der Anwendung von $S2Z$ und $A2Z$ handelt sich daher um einen Entwurfsschritt auf dem Weg zur Implementierung. Wir bieten diese kanonischen Übersetzungen $S2Z$ und $A2Z$ an, weil sie sich zum Beispiel zur Verifikation, zum Model Checking oder zur Nutzung als nicht weiter optimierte Implementierung eignen.

Schnittstellenabstraktion für Zustandsmaschinen

Sei ein System durch eine Zustandsmaschine gegeben:

$$\Delta : (\text{STATE} \times \text{SEQ}(I)) \to \wp(\text{STATE} \times \text{SEQ}(O)).$$

Die Zustandsmaschine Δ induziert ein im Sinne der Ein/Ausgabe, also der Black-Box-Sicht, äquivalentes zunächst mit dem jeweiligen Startzustand parametrisiertes Schnittstellenverhalten

$$V : \text{STATE} \to (\vec{I} \to \wp(\vec{O})),$$

das eine Abstraktion der Zustandsmaschine Δ darstellt. Dabei werden Zustandsdetails abstrahiert, während das Schnittstellenverhalten erhalten bleibt. Für jeden Zustand $\sigma \in \text{STATE}$, jede Eingabe $a \in \text{SEQ}(I)$ und jede Kanalbelegung $x \in \vec{I}$ wird das Schnittstellenverhalten V wie folgt definiert:

$$V(\sigma)(a\&x) = \{b\&y : \exists \sigma' \in \text{STATE} : (\sigma', b) \in \Delta(\sigma, a) \wedge y \in V(\sigma')(x)\}.$$

Da die rechte Seite der Gleichung in $V(\sigma')(x)$ inklusionsmonoton ist, existiert eine eindeutige, im mengentheoretischen Sinn größte Lösung [20, 32]. Da Δ ein Moore Automat ist, ist $V(\sigma)$ stets stark kausal. Für Zustandsmaschinen $M = (\Delta, \sigma_0)$ legen wir das Schnittstellenverhalten fest mit $Z2S(M) = V(\sigma_0)$.

Schnittstellenabstraktion für Architekturen

Auch einer Architektur Ψ ordnen wir eine Schnittstellenabstraktion $A2S[\Psi]$ zu. Sei K die Menge der Komponenten der Architektur Ψ. Dabei machen wir uns die induktive Definition hierarchischer Architekturen zunutze. Denn wurde für jede Komponente $k \in K$ von Ψ ein Schnittstellenverhalten festgelegt, so wird mit der Komposition $\otimes$ das Schnittstellenverhalten für Ψ festgelegt.

Auf unterster Hierarchieebene sind alle Komponenten als Zustandsmaschinen oder bereits explizit als Schnittstellen gegeben. Den Zustandsmaschinen weisen wir ihr Schnittstellenverhalten wie eben festgelegt zu. Die folgende Gleichung ist aufgrund der induktiven Definition von $\mathbb{A}$ eindeutig. Für $\Psi \in \mathbb{A}_{i+1}$ ist:

$$A2S(\Psi) = \bigotimes\{\Psi(k) : k \in K \wedge \Psi(k) \in \mathbb{F}\} \otimes \bigotimes\{Z2S(\Psi(k)) : k \in K \wedge \Psi(k) \in \mathbb{M}\} \otimes \bigotimes\{A2S(\Psi(k)) : k \in K \wedge \Psi(k) \in \mathbb{A}_i\}.$$

Damit wird jeder Architektur in der Hierarchie ein Schnittstellenverhalten zugeordnet.

Schnittstellen als Zustandsmaschinen

Jedes Schnittstellenverhalten kann kanonisch auf eine Zustandsmaschine abgebildet werden. Wir beschränken uns aus Gründen der Einfachheit auf den deterministischen Fall. Die Ergebnisse sind aber auf den nichtdeterministischen Fall verallgemeinerbar. Sei das Schnittstellenverhalten

$F : \vec{I} \to \vec{O}$

gegeben. Wir definieren daraus eine Zustandsmaschine

$\Delta_F : (ST \times \text{SEQ}(I)) \to (ST \times \text{SEQ}(O))$

wie folgt: Wir wählen als Zustandsraum den Funktionsraum

$ST = (\vec{I} \to \vec{O})$,

der alle Schnittstellenverhalten umfasst. Die Zustandsübergange spezifizieren wir, indem wir das Residuum des Verhaltens nach Eingabe von x und Ausgabe von y als neuen Zustand nutzen:

$\Delta_F(G, x) = \{(H, y) : \forall s \in \vec{I} : G(x\&s) = y\&H(s)\}$.

Zusätzlich ist F der Anfangszustand der Maschine.

$S2Z[F] = (\Delta_F, F)$.

Im Allgemeinen ist jedoch der tatsächlich erreichbare Zustandsraum einer so konstruierten Zustandsmaschine deutlich kleiner als der hier festgelegte Raum aller Funktionen, die Schnittstellenverhalten beschreiben.

Ein kanonischer Operator zur Abbildung von Schnittstellenverhalten oder Zustandsmaschinen in einer Architektur kann nicht existieren, da die Architektur signifikant zusätzliche Realisierungsdetails umfasst, die durch einen Entwickler hinzuzufügen sind. Jedoch kann der Operator $A2S$ genutzt werden, um die Übereinstimmung eines manuell entwickelten Architekturentwurfs mit einer vorgegebenen Schnittstellenspezifikation oder einer Zustandsmaschine zu prüfen.

Architekturen als Zustandsmaschinen

Mit $S2Z$ können wir jedes Schnittstellenverhalten in eine Zustandsmaschine verwandeln. Auf Zustandsmaschinen existiert ein Kompositionsoperator, der völlig kompatibel zur Komposition von Schnittstellenverhalten ist und der jeder Architektur ein Schnittstellenverhalten zuordnet. Damit kann jede Architektur auch auf einer Zustandsmaschine abgebildet werden. Dabei bleibt der Zustandsraum endlich, wenn der Zustandsraum jeder einzelnen Zustandsmaschine endlich war.

Es entsteht allerdings typischerweise ein komponierter Zustandsraum in Form eines sehr großen Kreuzprodukts. Dieser Zustandsraum repräsentiert die Menge aller Zustände jeder der Komponenten

in der Architektur. Der direkte Weg einer Hintereinanderschaltung von *A2S* und *S2Z* führt sogar zu einem isomorphen Zustandsautomaten mit einem unendlichen Zustandsraum (bestehend aus Funktionsmengen).

Morphismen

In diesem Kapitel haben wir bereits gezeigt, wie sich Zustandsmaschinen und Architekturen auf Schnittstellenverhalten und Schnittstellenverhalten und Architekturen auf Zustandsmaschinen abbilden lassen. Unverzichtbar ist, dass diese Abbildungen mit den verfügbaren Kompositionen verträglich sind, mathematisch ausgedrückt, dass sie *Morphismen* bilden. Wichtig sind die Eigenschaften der Form:

$$Z2S[M_1 \otimes M_2] = Z2S[M_1] \otimes Z2S[M_2],$$
$$A2S[\Psi_1 \otimes \Psi_2] = A2S[\Psi_1] \otimes A2S[\Psi_2].$$

Diese Gleichungen können durch Induktion über die Zeitintervalle bewiesen werden und gelten analog für Mengen komponierter Zustandsmaschinen bzw. Architekturen. Komposition und Semantik im Sinne der Abbildung auf das Schnittstellenverhalten sind also vertauschbar.

Die Übersetzung *S2Z* von Schnittstellenverhalten in Zustandsmaschinen bildet außerdem mit seiner Inversen einen Isomorphismus. Für deterministische Funktionen F gilt:

$$Z2S[S2Z[F]] = F.$$

In ähnlicher Form stellt die Übersetzung von deterministischen Zustandsmaschinen M in Schnittstellenverhalten und zurück auf der Menge der deterministischen Zustandsmaschinen:

$$S2Z[Z2S[M]]$$

eine Isomorphie in Bezug auf das Verhalten dar. Es wird aus M eine verhaltensäquivalente Zustandsmaschine $S2Z[Z2S[M]]$ mit kanonisch minimalem Zustandsraum erzeugt.

Relationen zwischen Systemmodellen

In diesem Kapitel betrachten wir Relationen zwischen den Systemsichten. Sie dienen der Formalisierung der in einem Entwicklungsprozess notwendigen und durchführbaren Enwicklungsschritte.

Eigenschaftsverfeinerung

Gegeben seien zwei Schnittstellenverhalten $F_1, F_2 : \vec{I} \to \wp(\vec{O})$. F_2 heißt *Eigenschaftsverfeinerung* von F_1, wenn gilt:

$$\forall x \in \vec{I} : F_2(x) \subseteq F_1(x).$$

Wir notieren diese Relation mit $F_2 \subseteq F_1$. Damit ist F_2 eine deterministischere Beschreibung als F_1 und damit spezifischer (das Verhalten ist genau festgelegt) und letztendlich implementierungsnäher. Zu beachten ist hier, dass F_1 wie F_2 zu jeder möglichen Eingabe eine Menge von potentiellen Ausgaben beschreibt. F_2 hat allerdings weniger mögliche Ausgaben und ist damit deterministischer und erlaubt ein detaillierteres Verständnis und genauere Aussagen über die beschriebene Komponente. Ein Schnittstellenverhalten, ist völlig festgelegt, wenn es deterministisch ist, also $|F(x)| = 1$ für alle $x \in \vec{I}$ gilt. In diesem Fall hat F genau genommen die Signatur $F : \vec{I} \to \vec{O}$.

Ist ein Schnittstellenverhalten vollkommen deterministisch ist keine weitere (nicht auf paradoxes Verhalten führende) Eigenschaftsverfeinerung möglich. Deshalb ist dieses Konzept der Verfeinerung vor allem in der Anforderungsspezifikation nützlich, wenn Schritt für Schritt weitere Eigenschaften zu einer Schnittstellenbeschreibung hinzugenommen werden und damit weniger Realisierungen möglich sind.

Das Konzept der Verfeinerung lässt sich vom Schnittstellenverhalten problemlos auf Zustandsmaschinen übertragen. Wir betrachten zwei Zustandsmaschinen ($i = 1, 2$) gleicher Signatur

$$\Delta_i : \text{STATE} \times \text{INPUT} \to \wp(\text{STATE} \times \text{OUTPUT}).$$

Gilt für alle Eingaben x und alle Zustände σ

$$\Delta_2(\sigma, x) \subseteq \Delta_1(\sigma, x)$$

und ist Δ_2 total, so heißt Δ_2 Verfeinerung von Δ_1. Eine Zustandsmaschine heißt traditionsgemäß deterministisch, wenn sie immer genau einen Übergang anbietet, also $|\Delta(\sigma, x)| = 1$ für alle $x \in \vec{I}$. In diesem Fall hat Δ auch die Signatur $\Delta : \text{STATE} \times \text{INPUT} \to \text{STATE} \times \text{OUTPUT}$. Eine deterministische Zustandsmaschine hat ein deterministisches Schnittstellenverhalten. Die Komposition deterministischer Komponenten bleibt

deterministisch. Deshalb beschreibt eine Architektur aus deterministischen Komponenten ebenfalls ein deterministisches System [20].

Verallgemeinernd sind wir bei Zustandsmaschinen auch an Verfeinerungen interessiert, bei denen die Zustandsräume der Maschinen modifiziert werden. Gegeben sei ($i = 1, 2$)

$$\Delta_i : \text{STATE}_i \times \text{INPUT} \to \wp(\text{STATE}_i \times \text{OUTPUT}).$$

Δ_2 heißt Verfeinerung von Δ_1, falls eine simulierende Relation

$$\alpha \subseteq \text{STATE}_i \times \text{STATE}_j$$

existiert, sodass Δ_2 total ist und für die Anfangszustände gilt $\alpha(\sigma 0_1, \sigma 0_2)$. Außerdem gilt für alle in Δ_1 erreichbaren Zustände $\sigma_1 \in \text{STATE}_1$ und alle $\sigma_2 \in \text{STATE}_2$ mit $\alpha(\sigma_1, \sigma_2)$ und alle Eingaben x:

$$\Delta_2(\sigma_2, x) \subseteq \{(\sigma_2', y) | \exists \sigma_1' : (\sigma_1', y) \in \Delta_1(\sigma_1, x) \wedge \alpha(\sigma_1', \sigma_2')\}.$$

Damit können wir in einer Verfeinerung auch den Zustandsraum ändern.

Essentiell ist die Kompatibilität der Kompositionsoperatoren mit jeglicher Form der Verfeinerung: Wird eine Komponente einer Architektur verfeinert, so bedeutet dies auch eine Verfeinerung der Gesamtarchitektur. Es gilt (ohne Beschränkung unter der Annahme Ψ_1 und Ψ_2 haben dieselben Kanalidentifikatoren K):

$$\forall k \in K : A2S[\Psi_2(k)] \subseteq A2S[\Psi_1(k)]$$

impliziert

$$A2S[\Psi_2] \subseteq A2S[\Psi_1].$$

Der Beweis hierfür folgt induktiv über die Hierarchie aus der Definition der Schnittstellenkomposition. Er kann genau so auf Zustandsmaschinen und Architekturen übertragen werden. Zusätzlich gibt es für Zustandsmaschinen und Architekturen darauf zugeschnittene Verfeinerungsoperatoren, die häufige Verfeinerungsmuster operationalisieren. Dazu gehören zum Beispiel Teilungen des Zustandsraums [26, 32] oder durch Modifikation der inneren Architektur durch Faltung und Entfaltung, in einer Form dass eine Eigenschaftsverfeinerung auf dem Schnittstellenverhalten der übergeordneten Architektur gesichert bleibt [29, 30, 32].

Wechsel der Abstraktionsebene

Eine weitere Verfeinerungsrelation für Schnittstellenverhalten, die die oben eingeführte Eigenschaftsverfeinerung sogar subsumiert, erlaubt den Wechsel der Abstraktionsebene bei Nachrichtenströmen und Komponenten. Seien zwei Schnittstellenverhalten

$$F_1 : \vec{I}_1 \to \wp(\vec{O}_1), \qquad F_2 : \vec{I}_2 \to \wp(\vec{O}_2)$$

gegeben, die dieselbe Komponente auf unterschiedlichen Abstraktionsebenen modellieren.

Zur Verbindung beider Verhalten seien folgende nicht notwendigerweise kausale, jedoch auch nicht paradoxe Abbildungen gegeben:

$$AI : \vec{I}_2 \to \wp(\vec{I}_1) \qquad RI : \vec{I}_1 \to \wp(\vec{I}_2)$$
$$AO : \vec{O}_2 \to \wp(\vec{O}_1) \qquad RO : \vec{O}_1 \to \wp(\vec{O}_2).$$

(AI, RI) und (AO, RO) heißen *Verfeinerungspaare*, falls $RI \circ AI$ und $RO \circ AO$ die Identität ergeben (vgl. [7]). (Die Identität ist für eine Menge C typisierter Kanäle das Schnittstellverhalten $Id : \vec{C} \to \wp(\vec{C})$ mit $Id(x) = \{x\}$ und die Funktionskomposition $\circ$ ist für Funktionen $F : \vec{C} \to \wp(\vec{C'})$, $F' : \vec{C'} \to \wp(\vec{C''})$ durch $(F \circ F')(x) = \{z \in F'(y) : y \in F(x)\}$ festgelegt). Falls mit den Verfeinerungspaare (AI, RI) und (AO, RO) folgende Gleichung gilt

$$F_1 = RI \circ F_2 \circ AO$$

heißt F_1 *Abstraktion* von F_2 und F_2 *Granularitätsverfeinerung* von F_1 (siehe Abb. 5). Eine solche Übersetzung von Kanälen kann zu einer Reihe von Schnittstellenanpassungen genutzt werden: Kanäle können gesplittet oder rekombiniert werden, neue Kanäle hinzugefügt oder unbenutzte entfernt werden, Datenstrukturwechsel in Schnittstellen werden ermöglicht.

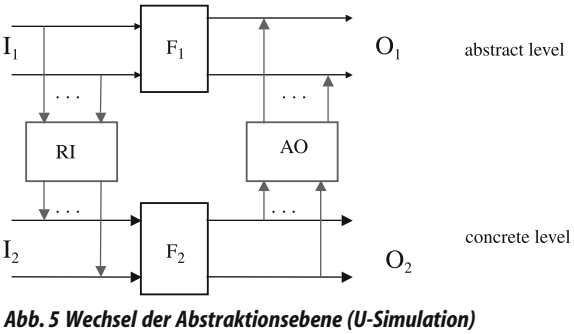

Abb. 5 Wechsel der Abstraktionsebene (U-Simulation)

Wieder ist essentiell, dass Abstraktion und Granularitätsverfeinerung kompatibel zu den anderen gegebenen Operatoren sind. Insbesondere kann die Anpassung einzelner Schnittstellen in eine Komposition eingebettet, die Granularitätsverfeinerung auf Zustandsmaschinen angewendet werden und Verhaltensverfeinerung von F_1 bzw. F_2 über die Verfeinerungspaare propagiert werden.

Systemmodelle für die schrittweise Entwicklung

In diesem Beitrag haben wir ein abstraktes Systemmodell mit einer Verhaltenssicht, einer Zustandssicht und einer hierarchisch gegliederten Struktur-/Architektursicht zusammen mit Kompositions- und Verfeinerungsoperatoren eingeführt und die Übergänge zwischen diesen Systemsichten festgelegt. Wir haben gezeigt, dass Komposition, Verfeinerungsformen und Wechsel zwischen den Systemsichten kompatibel zueinander sind und daher im Entwicklungsprozess in jeweils notwendigen Formen angewendet werden können. Wir haben aber wenig über die Anwendung der verfügbaren Operatoren gesagt. Völlig offen haben wir auch die Frage einer Beschreibungssprache gelassen, also der syntaktischen Form zur Darstellung der Modelle. Damit gehen wir genau den umgekehrten Weg der OMG, die mit der UML eine Sprache vorgibt, aber keine Modellierungstheorie hat.

Das Systemmodell ist unter anderem zur Modellierung allgemeiner Strukturen der Informationsverarbeitung wie Hardware und Software geeignet. Darüber hinaus können in dem Systemmodell auch Strukturen anderer Ingenieursdisziplinen, allen voran des Maschinenbaus beschrieben werden, allerdings in einer Abstraktion die einer diskreten Modellierung entspricht. Damit können auch Verhalten und Struktur von Anlagen des Maschinenbaus als abstrakte Zustandsmaschinen beschrieben werden. Diese Sicht entspricht auch der Perspektive der Automatisierungstechnik, wenn die Steuerung, beschrieben als Moore-Automat, mit einem mechanischen System, versehen mit Aktuatoren und Sensoren, ebenfalls modelliert als Moore-Automat kommuniziert. Für die integrierte Betrachtung diskreten und kontinuierlichen Verhaltens existiert eine Ergänzung der vorgestellten Modellierungstheorie in [37].

Und natürlich kann das Systemmodell auch zur Modellierung wirtschaftlicher und organisatorischer Abläufe im und zwischen Unternehmen eingesetzt werden (vgl. [39]). So entsteht eine nahtlose Verbindung zwischen makro-ökonomischen Abläufen, Geschäftsabläufen im Unternehmen, Produktionsprozessen und Steuerungen einzelner Maschinen, die eine integrierte Gesamtbetrachtung und evolutionäre Planung ermöglicht.

Abschließende Bemerkungen

Die beschriebene Theorie für die Modellierung ist auf den ersten Blick eine mathematische Theorie mit einer Reihe von Theoremen wie sie für bestimmte Ausprägungen der Algebra typisch sind. Über die Frage der Gültigkeit gewisser mathematischer Theoreme hinaus stellt sich für die Informatik selbstverständlich die Frage, ob und wie gut die betrachteten Modelle für die Aufgaben des Software und Systems Engineering geeignet sind. Diese Frage kann nicht mit den Mitteln der Mathematik beantwortet werden, sondern nur durch den Nachweis, dass diese Modelle methodisch sinnvoll und zielgerichtet in der Entwicklung softwareintensiver Systeme eingesetzt werden können. Dies ist eine Frage der Empirie und der Erfahrungen sowie erfolgreicher Experimente. Daneben ist die Frage nach der Werkzeugunterstützung für praktische Zwecke von entscheidender Bedeutung.

Für den vorgestellten Ansatz sind eine ganze Reihe von Experimenten durchgeführt worden und auch prototypische Werkzeugunterstützung (vgl. [1–3, 10]) wurde realisiert. Dadurch wurde die Eignung des Ansatzes für ein ingenieurmäßiges Vorgehen erprobt.

Ein weiterer wesentlicher Aspekt der modellbasierten Entwicklung betrifft die allgemeine wissenschaftliche Fundierung von Prinzipien des Software Engineerings. Im Laufe der letzten drei bis vier Jahrzehnte haben sich im Software Engineering eine Reihe fundamentaler Prinzipien herauskristallisiert, wie etwa „Wechsel der Abstraktionsebene“, „Divide and Coquer“, „Information Hiding“, „Modulare Entwicklung“, „Schrittweise Verfeinerung“ oder „Schichtenarchitekturen“. Die hier vorgestellten Techniken der Modellierung erlauben es, all diesen Begriffen eine sehr präzise Definition zu geben.

Auch wenn bei den Modellierungstechniken die Informatik in den vergangenen Jahren eindrucksvolle Fortschritte erzielt hat, gibt es noch viele offene Forschungsthemen. Ein Beispiel dafür ist die Verbindung der Modellierungstheorie mir

der Theorie der Programmiersprachen. Andere Beispiele sind Anwendungen der Modellierungstheorie auf das Gebiet der Entwurfs- und Architekturmuster oder domänenspezifische Ausprägungen der Modellierungstheorie.

Literatur

1. AutoFocus Webseite, http://autofocus.in.tum.de (2005)
2. AutoFocus 2 – Webseite, http://www4.in.tum.de/~af2/ (2006)
3. AutoRAID – Webseite, http://www4.in.tum.de/~autoraid/ (2005)
4. Bauer, F.L., Berghammer, R., Broy, M., Dosch, W., Geiselbrechtinger, F., Gnatz, R., Hangel, E., Hesse, W., Krieg-Brückner, B., Laut, A., Matzner, T., Möller, B., Nickl, F., Partsch, H., Pepper, P., Samelson, K., Wirsing, M., Wössner, H.: The Munich Project CIP, Vol 1: The Wide Spectrum Language CIP-L. LNCS 183, Springer (1985)
5. Bauer, A., Broy, M., Romberg, J., Schätz, B., Braun, P., Freund, U., Mata, N., Sandner, R., Ziegenbein, D.: Auto-MoDe-Notations, Methods, and Tools for Model-Based Development of Automotive Software. In: Proceedings of the SAE 2005 World Congress, Detroit, MI, Society of Automotive Engineers (2005)
6. Breu, R., Grosu, R., Huber, F., Rumpe, B., Schwerin, W.: Systems, Views and Models of UML. In: Schader, M., Korthaus, A. (eds.) The Unified Modeling Language, Technical Aspects and Applications. Heidelberg: Physica (1998)
7. Broy, M.: Interaction Refinement – The Easy Way. In: Broy, M. (ed.) Program Design Calculi. Springer NATO ASI Series, Series F: Computer and System Sciences, Vol. 118 (1993)
8. Broy, M., Hofmann, C., Krüger, I., Schmidt, M.: A Graphical Description Technique for Communication in Software Architectures. In: Joint 1997 Asia Pacific Software Engineering Conference and International Computer Science Conference (APSEC '97/ICSC'97) (1997)
9. Broy, M., Stølen, K.: Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement. Springer (2001)
10. Broy, M., Huber, F., Schätz, B.: AutoFocus – Ein Werkzeugprototyp zur Entwicklung eingebetteter Systeme. IFE-AF (1999)
11. Broy, M., Breu, R., Huber, F., Krüger, I., Rumpe, B., Schwerin, W.: Methodik, Sprachen und Grundlagen des Software Engineering – Abschlußbericht des Forschungslabors SysLab. Informatik Forsch. Entw. 16(1), 53–59 (2001)
12. Broy, M.: Modeling Services and Layered Architectures. In: König, H., Heiner, M., Wolisz, A. (eds.) Formal Techniques for Networked and Distributed Systems. LNCS 2767, Berlin: Springer (2003)
13. Czarnecki, K., Eisenecker, U.: Generative Programming: Methods, Techniques, and Applications. Addison-Wesley (2000)
14. Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R., Stafford, J.: Documenting Software Architectures: Views and Beyond. Series: The SEI Series in Software Engineering. Addison-Wesley Professional (2002)
15. Chen, P.: The Entity-Relationship Model – Toward a Unified View on Data. ACM Trans. Database Syst. 1(1) (1976)
16. Codd, E.F.: A Relational Model of Data for Large Shared Data Banks. Commun. ACM 13, 377–387 (1970)
17. Ehrig, H., Mahr, B.: Fundamentals of Algebraic Specifications I. Berlin: Springer (1985)
18. Filman, R., Elrad, T., Clarke, S., Aksit, M.: Aspect-Oriented Software Development. Addison-Wesley Professional (2004)
19. Grosu, R., Klein, C., Rumpe, B., Broy, M.: State Transition Diagrams. Technical report TUM-I9630, Technische Universität München (1996)
20. Grosu, R., Rumpe, B.: Concurrent Timed Port Automata. Technical report TUM-I9533, Technische Universität München (1995)
21. Harel, D.: Statecharts: A Visual Formalism for Complex Systems. Sci. Comput. Program. 8, 231–274 (1987)
22. Harel, D., Rumpe, B.: Meaningful Modeling: What's the Semantics of „Semantics"? Computer 37(10), 64–72 (2004)
23. Hasselbring, W., Reussner, R.: Handbuch der Software-Architektur. dPunkt (2006)
24. Herzberg, D., Broy, M.: Modeling Layered Distributed Communication Systems. Formal Aspects of Computing, No. 17, Springer (2005)
25. Klein, C., Rumpe, B., Broy, M.: A stream-based mathematical model for distributed information processing systems – SysLab system model –. In: Proceedings of the first International Workshop on Formal Methods for Open Object-based Distributed Systems. Chapman-Hall (1996)
26. Klein, C., Prehofer, C., Rumpe, B.: Feature Specification and Refinement with State Transition Diagrams. In: Fourth IEEE Workshop on Feature Interactions in Telecommunications Networks and Distributed Systems. Dini, P. (Ed.) IOS-Press (1997)
27. Krüger, I., Grosu, R., Scholz, P., Broy, M.: From MSCs to statecharts. In: Proceedings of DIPES'98, Kluwer (1999)
28. Parnas, D.: On the criteria to be used to decompose systems into modules. Commun. ACM 15, 1053–1058 (1972)
29. Philipps, J., Rumpe, B.: Refinement of Information Flow Architectures, ICFEM'97 (1997)
30. Philipps, J., Rumpe, B.: Refinement of Pipe And Filter Architectures. Formal Methods'99, LNCS 1708. Springer (1999)
31. Selic, B., Gullekson, G., Ward, P.T.: Real-time Objectoriented Modeling. New York: Wiley (1994)
32. Rumpe, B.: Formale Methodik des Entwurfs verteilter objektorientierter Systeme. Herbert Utz Wissenschaft (1996)
33. Rumpe, B.: Model-Based Testing of Object-Oriented Systems. In: de Boer, F., Bonsangue, M., Graf, S., de Roever, W.-P. (Eds.) Formal Methods for Components and Objects. International Symposium, FMCO 2002. Leiden, November 2002. Revised Lectures. LNCS 2852, Springer (2003)
34. Rumpe, B.: Agile Modellierung mit UML – Codegenerierung, Testfälle, Refactoring. Springer (2004)
35. Rumpe, B.: Modellierung mit UML – Sprache, Konzepte und Methodik. Springer (2004)
36. Stachowiak, H.: Allgemeine Modelltheorie. Wien: Springer (1973)
37. Stauner, T., Rumpe, B., Scholz, P.: Hybrid System Model. Technical Report TUM-I9903, Technische Universität München (1999)
38. Broy, M., Facchi, C., Grosu, R., Hettler, R., Hussmann, H., Nazareth, D., Regensburger, R., Stolen, K.: The Requirement and Design Specification Language SPECTRUM. Technische Universität München, Institut für Informatik, TUM-I9140, Oktober (1991)
39. Thurner, V.: Formal fundierte Modellierung von Geschäftsprozessen. Dissertation, Technischen Universität München, Fakultät für Informatik (2004)
40. OMG. The UML Standard Version 2.1 (2006)
41. Ehrig, H., Mahr, B.: Fundamentals of Algebraic Specifications I. Berlin: Springer (2005)
42. Astesiano, E., Bidoit, M., Krieg-Brückner, B., Mosses, P.D., Sannella, D., Tarlecki, A.: CASL: The {Common Algebraic Specification Language. J. Theor. Comput. Sci. 286(2), 153–196 (2002)
43. Abrial, J.-R.: The B-book. Assigning Programs to Meanings. Cambridge University Press (1996)
44. Spivey, J.: Understanding Z. Cambridge University Press (1988)
45. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL. Springer (2002)
46. Milner, R.: Communication and Concurrency. Prentice Hall (1989)
47. Hoare, A.: Communicating Sequential Processes. Prentice Hall (1985)
48. Milner, R.: A calculus of communicating systems. Springer (1980)
49. Börger, E., Stärk, R.: Abstract State Machines. A Method for High-Level System Design and Analysis. Springer (2003)