

## Modellieren mit HERAKLIT

### Prinzipien und Fallstudie

Peter Fettke<sup>1</sup>, Wolfgang Reisig<sup>2</sup>

**Abstract:** HERAKLIT ist ein laufendes Forschungsprogramm und Entwicklungsprojekt mit dem Ziel der Schaffung einer Infrastruktur zur Modellierung großer, rechnerintegrierter Systeme. Wir diskutieren die zentralen Anforderungen an solche Modelle (Hierarchien, Nutzersicht, Überführung informaler in formale Ideen, schematische Modelle, gleichrangiger Umgang mit digitalisierten und personengebundenen Prozessen) und erläutern, wie HERAKLIT diese Anforderungen unterstützt. Eine Fallstudie zeigt die Nutzbarkeit von HERAKLIT in der Praxis.

**Keywords:** systems composition; data modelling; behaviour modelling; composition calculus; algebraic specification; systems mining

### 1 Einleitung

Informatik hat zwei Gesichter: einerseits die technischen Grundlagen mit den formalen Konzepten zu ihrer Nutzung, andererseits die Anwendung. *Dijkstra* hat vielfach vorgeschlagen, die formale und informelle Sicht zu trennen und zwischen ihnen eine Mauer (engl. „firewall“) zu errichten [Di89]. Begründung: Das „correctness problem“ des Informatikers verlangt ganz andere Herangehensweisen als das „pleasantness problem“ des Anwenders. Modellierung findet in diesem Bild auf der einen oder anderen Seite dieser Mauer statt.

Demgegenüber schlagen wir hier vor, Modellierung als eine Tätigkeit zu verstehen, die einen möglichst bruchlosen Übergang zwischen formal formulierten und lebensweltlichen Sachverhalten erlaubt. Modelle verknüpfen Anwendung und Technik und beruhen auf denselben Grundlagen.

In diesem Beitrag beschreiben und motivieren wir zunächst im zweiten Abschnitt Anforderungen, die heutzutage an die Modellierung rechnerintegrierter Systeme gestellt werden. Kapitel 3 stellt dann die Konzepte vor, mit denen HERAKLIT [FR21a, FR21b] diese Anforderungen bewältigt. Eine umfangreiche Fallstudie zeigt im vierten Kapitel, dass die Erfüllung aller Anforderungen in einem integrierten Framework tatsächlich gelingt. Schließlich wird

---

<sup>1</sup> Saarland University, Saarbrücken, Germany, and German Research Center for Artificial Intelligence (DFKI), Saarbrücken, Germany, peter.fettke@dfki.de

<sup>2</sup> Humboldt-Universität zu Berlin, Berlin, Germany, reisig@informatik.hu-berlin.de

im fünften Kapitel diskutiert, in welchem Umfang andere Frameworks die Anforderungen aus Kapitel 2 entsprechen.

## 2 Anforderungen an die Modellierung rechnerintegrierter Systeme

Der Herstellung eines komplexen rechnerintegrierten Systems ist immer ein Planungsprozess vorgeschaltet, in dem die Architektur, Funktionsweise, beabsichtigten Effekte etc. des Produktes formuliert werden. Zentrales Werkzeug und Hilfsmittel dafür sind Modelle. Auch ein gegebenes komplexes System kann man besser verstehen und analysieren, indem man es modelliert. Ein Modell betont einige Aspekte des Systems, oft mit graphischen, standardisierten Darstellungen. Ein gutes Modell ist nicht nur anschaulicher und besser verständlich als eine umgangssprachliche Darstellung; es lässt sich auch vielfältig nutzen: man kann Funktionalität und Performanz-Engpässe erkennen, Kosten abschätzen, Korrektheit gegenüber einer Spezifikation nachweisen, Parameter optimieren und vieles mehr.

Moderne und zukünftige digitale Systeme, cyber-physikalische Systeme, digitale informationsbasierte Infrastrukturen, das *Internet of people, things and services* etc., werden zunehmend komplexer und wachsen immer mehr zusammen. Folglich sind bei der Gestaltung rechnerintegrierter Systeme vielfältige Herausforderungen zu überwinden. Nötig sind dafür Frameworks, die insbesondere für große Systeme wirklich Nutzen bringen. Dafür ist eine Reihe von Aspekten besonders wichtig:

*Komposition und hierarchische Verfeinerung lokaler Komponenten:* Was bedeutet es konzeptuell, dass rechnerintegrierte Systeme „groß“ sind? Es bedeutet zunächst einmal, dass einige Konzepte, die für kleine Systeme durchaus verwendet werden können, nun nicht mehr praktikabel sind. Am augenfälligsten gilt das für globale Zustände und globale Schritte. Und es bedeutet, dass ein großes System im Allgemeinen nicht amorph, gleichförmig gestaltet ist, sondern in irgendeiner Form komponiert ist aus kleineren, mehr oder weniger selbständigen agierenden Systemen. Komposition und Verfeinerung sind also fundamentale Konzepte zur Modellierung großer Systeme.

*Modelle aus Sicht der Anwender:* Es gibt mehrere Weisen, Informatik im Allgemeinen und Modellierung in der Informatik im Besonderen zu fassen. Häufig wird der Zugang über die theoretische Informatik gewählt, mit der Betrachtung von Zeichenketten und ihrer Transformation. Beliebt ist auch der Zugang von der technischen Seite mit digitalen Schaltungen und ihrer Abstraktion, hin zu Software, Datenbanken, etc. Zum Verständnis großer Systeme besonders nützlich erscheint aber der Zugang aus der Sicht der Anwender, Nutzer, Betreiber und der Zwecke, für die große Systeme betrieben werden.

*Systematische Überführung informeller, intuitiver Ideen zu einem gegebenen oder intendierten System in ein formales Modell:* Dafür bietet die Informatik strukturierte Konzepte. Allerdings münden sie zumeist in Programme, nicht in formale Modelle.

*Parametrisierte, schematische Modellierung:* Oft soll nicht nur ein einziges System modelliert werden, sondern eine Menge gleichartig strukturierter und verhaltensähnlicher Modelle.

*Integrierter, einheitlicher, gleichrangiger Umgang mit digitalisierten und lebensweltlichen Prozessen:* Ein Beispiel ist in einem Betrieb die automatisierte digitale Personalverwaltung, zusammen mit der Belehrung eines Arbeitnehmers über gefährliche Güter und dessen Bestätigung durch seine Unterschrift. Neben symbolischen, implementierten Konzepten muss ein Modell also auch pragmatische, nicht zur Implementierung vorgesehene Schritte darstellen können.

### 3 Von HERAKLIT verwendete Konzepte

Informatik wird üblicherweise von der Rechentechnik her oder vom Symbolischen her aufgefasst, also aus der Sicht der technischen Informatik, oder der klassischen theoretischen Informatik. Demgegenüber betrachtet HERAKLIT die Informatik aus einer dritten Sicht, der Sicht der Anwender. Für HERAKLIT zentral ist die holistische Auffassung von Systemen und ihren Modellen: Es reicht nicht, einzelne Anforderungen an das Systemverhalten, Verfahrensanleitungen für Anwender, Datensammlungen, Algorithmen, Softwarekomponenten, Hardwarekonzepte etc. isoliert zu betrachten, zu verstehen und zu verwenden. Vielmehr muss ihr Zusammenspiel verstanden werden. Dafür braucht man ein Modell, das alle diese Aspekte integriert. HERAKLIT ist auf diese Anforderungen an Modelle zugeschnitten.

#### 3.1 Konzeptionelle Sicht

HERAKLIT unterstützt die Modellierung von Systemen, die aus Teilsystemen zusammengesetzt sind, und die Hierarchien bilden, in denen konkrete Produkte, Maschinen, Menschen, Dokumente, Datenspeicher etc. kombiniert, separiert, transformiert, berechnet, bearbeitet, transportiert, erzeugt, vernichtet werden. HERAKLIT-Modelle können abstrakt, detailliert, schematisch sein; Daten, Gegenstände, Schritte zur Beschreibung dynamischen Verhaltens sind kleinteilig, aber auch umfassend formulierbar. Konkrete und abstrakte Datenstrukturen, sowie die Beschreibung von Verhalten werden integriert und aufeinander bezogen modelliert. HERAKLIT bietet dafür aufeinander abgestimmte Ausdrucksmittel, die den Modellierer bei der Formulierung seiner Ideen unterstützen. Der Modellierer wird nicht eingeschränkt; bei der Wahl des Abstraktionsgrades und der Art der Komposition von Modulen hat er inhaltlich alle Freiheiten; HERAKLIT schlägt aber konzeptionell vereinheitlichende Darstellungen vor. HERAKLIT integriert neue Konzepte mit bewährten, tiefliegend motivierten Konzepten der Modellierung von Software, Geschäftsprozessen und anderen Systemen. Zudem bietet HERAKLIT ein Konzept zur Abstraktion konkreter Daten, und damit die symbolische Modellierung großer Klassen von Systemen, die ähnlich aufgebaut sind und sich ähnlich verhalten.

Mit seinen wenigen, aber ausdrucksstarken Konzepten sind HERAKLIT-Modelle:

- für den Entwickler besser beherrschbar,
- für den Nutzer leicht verständlich,
- weniger fehleranfällig und leichter verifizierbar,
- einfacher änderbar, sowie
- schneller herstellbar und kostengünstiger als andere Methoden, insbesondere auch für wirklich große Systeme.

### 3.2 Technische Sicht

Technisch und formal neu ist die Integration von Architektur, statischen und dynamischen Konzepten. Dafür kombiniert HERAKLIT bewährte mathematisch basierte und intuitiv leicht verständliche Konzepte wie Prädikatenlogik, abstrakte Datentypen und Petrinetze, die auch bisher schon zur Spezifikation von Systemen verwendet werden; HERAKLIT kombiniert sie neu und ergänzt sie um den Kompositionskalkül. Das HERAKLIT-Framework

- unterstützt die hierarchische Partitionierung eines großen Systems in Module;
- kann technisch einfach, aber inhaltlich flexibel und ausdrucksstark Module zu großen Systemen komponieren;
- beschreibt diskrete Schritte in großen Systemen lokal konzentriert in einzelnen Modulen in frei gewähltem Detaillierungsgrad;
- stellt alle Arten von Modulen integriert mit den gleichen Konzepten dar, egal ob sie zur Implementierung vorgesehen sind oder nicht;
- repräsentiert Daten auf frei gewählter Abstraktionsstufe;
- berücksichtigt Datenabhängigkeiten im Kontrollfluss;
- kann von konkreten Daten abstrahieren, um verhaltensgleiche Instanziierungen in einem Schema zu fassen;
- kann Modelle generieren, die skalierbar, systematisch, änderbar und erweiterbar sind;
- unterstützt den Nachweis, dass ein Modell gewünschte Eigenschaften hat.

Mit den beschriebenen technischen Konzepten können Aspekte diskreter Systeme modelliert werden, wie es keine andere Modellierungstechnik ermöglicht:

*Module unterschiedlicher Hierarchiestufen können komponiert werden.* Beispielsweise kann in einem HERAKLIT-Modell ein Modul eine Systemkomponente auf feinster, operationeller

Stufe modellieren; zugleich werden seine benachbarten Module nur mit ihrer Schnittstelle, ihrem Namen und ggf. Teilen ihrer inneren Struktur repräsentiert.

*Lebensweltliche und abstrakte Objekte sind gleichrangige Elemente formaler Argumentation.* HERAKLIT folgt der Prädikatenlogik, und fasst lebensweltliche und formale Objekte gleichrangig als Interpretationen von Termen einer Signatur (eines Alphabets mit getypten Symbolen) auf.

*Verhaltensmodelle können parametrisiert werden.* Jede Interpretation der symbolischen Beschriftungen eines schematischen HERAKLIT-Moduls generiert ein eigenes Systemmodell. Beispiel: Ein schematisches Modul beschreibt die Prinzipien der Geschäftsprozesse aller Filialen eines Bekleidungsunternehmens; jede Interpretation beschreibt eine konkrete Filiale.

*Der Umfang von Modellen wächst linear in der Größe des modellierten Systems.* Das gilt insbesondere auch bei der Modellierung dynamischen Verhaltens. Erreicht wird dies mit dem konsequenteren Verzicht auf globale Konstruktionen (beispielsweise globale Zustände) bei der Bildung einzelner Abläufe, mit einem lokal beschränkten Kompositionsooperator, etc.

*Bewährte Verifikationstechniken werden übernommen.* Petrinette bieten eine Vielzahl effizienter rechnergestützter Verifikationstechniken. Die wichtigsten, insbesondere Platz- und Transitions-Invarianten von Petrinetzen, funktionieren auch auf symbolischer (Signatur-) Ebene und – entsprechend ausdrucksstärker – für die einzelnen Interpretationen einer Signatur. Eine Verifikation auf der Basis des *model checking* ist für einzelne Interpretationen einer Signatur möglich; dafür gibt es effiziente Verfahren. Ein großes, offenes Feld ist die kompositionale Verifikation: Eigenschaften eines komponierten Systems werden aus Eigenschaften der komponierten Module ableitbar.

*Ein einzelner Ablauf kann verteilt sein.* In einem Ablauf eines großen Systems treten Ereignisse oft unabhängig voneinander ein. Beispiel: In einer Verkaufsstelle interagieren die Mitarbeiter unabhängig voneinander mit einzelnen Kunden; sie werden nur gelegentlich synchronisiert beim Zugang zu knappen Ressourcen, beispielsweise beim Bezahlen an der Registrierkasse. Viele Frameworks verstehen Unabhängigkeit als „beliebige Reihenfolge“; das führt zu exponentiell vielen vermeintlich verschiedenen „sequentiellen“ Abläufen. HERAKLIT modelliert Unabhängigkeit von Ereignissen in Abläufen explizit. Das führt zu einem klaren Begriff von Nichtdeterminismus und zu einem Theorem, nachdem die Abläufe eines komponierten Systems aus den Abläufen der Komponenten ableitbar sind.

*HERAKLIT ist intuitiv einfach.* HERAKLIT verwendet intuitiv einfache Basis-Konzepte: Prädikatenlogik ist vielen Anwendern aus anderen Kontexten ohnehin geläufig. Auch Petrinette sind weit verbreitet. Der Kompositionskalkül ist an kleinen Beispielen unmittelbar einsichtig. Die graphischen Ausdrucksmittel von Petrinetzen und des *composition calculus* ergänzen sich harmonisch.

### 3.3 Die drei Dimensionen von HERAKLIT

HERAKLIT unterscheidet drei Dimensionen, die drei unterschiedliche, aber integrierte Aspekte der Modellierung kennzeichnen:

*Architektur:* Ein HERAKLIT-Modell besteht aus *Modulen*. Jedes Modul hat ein beliebig gestaltetes Inneres; insbesondere ist die Abstraktionsebene der Darstellung frei wählbar. Die Schnittstelle eines Moduls enthält beliebige gelabelte Elemente. Komposition von Modulen ist technisch einfach und immer assoziativ. Formale Grundlage dieses Architekturprinzips ist der Kompositionskalkül aus [Re19].

*Statische Aspekte:* Für den Umgang mit Daten und Datenstrukturen greift HERAKLIT auf abstrakte Datentypen und algebraische Spezifikationen zurück, wie sie sich in der Informatik von Anfang an bewährt haben und in Spezifikationsprachen wie VDM [Jo91] und Z [Bo01] seit langem verwendet werden [ST12]. Symbolische Darstellungen (Terme einer Signatur) werden wie in der Prädikatenlogik verwendet. Zu jedem Modul gehört eine Signatur, also eine Menge von Symbolen für Mengen, Konstanten und Funktionen, aus denen zusammen mit Variablen dann Terme gebildet werden. Eine Signatur, und damit ihre Terme, können in ganz unterschiedlichen Lebenswelten instanziert werden.

*Dynamische Aspekte:* Ein Modul zur Beschreibung von Verhalten enthält in seinem Inneren ein Petrinetz [Pe77]. Dabei ist jeder Platz des Petrinetzes ein prädikatenlogisches Prädikat, jeder Pfeil ist mit einem Term der Signatur des Moduls beschriftet, und jede Transition mit einer Bedingung. Für eine gegebene Instanzierung der Signatur kann die Menge der Objekte, auf die das Prädikat zutrifft, durch den Eintritt eines Ereignisses wachsen oder schrumpfen. Im Einzelnen wird das durch Terme an den Pfeilen des Petrinetzes beschrieben. Damit kann Verhalten abstrakt auf schematischer Ebene, aber auch konkret für eine „gemeinte“ Instanzierung dargestellt werden.

## 4 Fallstudie

Gegeben sei ein Restaurant-Betrieb mit mehreren Filialen. Alle Filialen sind nach demselben Schema aufgebaut und verhalten sich nach demselben Muster; sie unterscheiden sich aber in einigen Einzelheiten. Modelliert wird das Schema aller Filialen, ein Beispiel einer Filiale, und ein konkreter Ablauf in dieser Filiale.

### 4.1 Die Architektur der Filialen

Grundlegendes Konzept der Architektur der Filialen sind Module, wie sie in Abschnitt 3.2 generell für HERAKLIT-Modelle skizziert wurden. Im Inneren der hier verwendeten Module liegen Petrinetze in verschiedenen Ausprägungen. Generell hat ein Modul  $M$  zwei

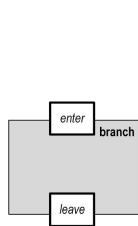


Abb. 1: abstrakte Sicht auf eine Filiale

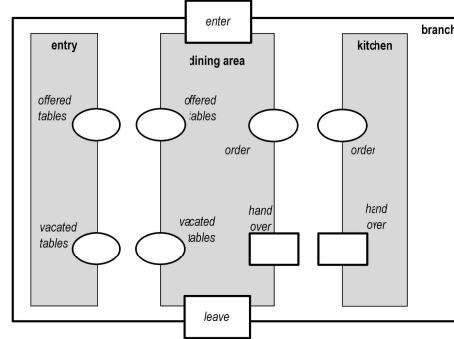


Abb. 2: eine Filiale, bestehend aus drei Modulen

Schnittstellen, die linke Schnittstelle  $*M$  und die rechte Schnittstelle  $M^*$ . Eine Schnittstelle kann Petrinetz-Plätze und -Transitionsen enthalten. Graphisch wird ein Modul rechteckig dargestellt, mit den Elementen der linken Schnittstelle auf dem linken oder oberen Rand des Rechtecks und den Elementen der rechten Schnittstelle auf dem rechten oder unteren Rand.

Abb. 1 zeigt das Schema für die Filialen in größtmöglicher Abstraktion als ein HERAKLIT-Modul, *branch*. Modelliert werden zwei Aktivitäten der Restaurant-Gäste: das Betreten und das Verlassen einer Filiale; technisch als Petrinetz-Transitionen in  $*branch$  und  $branch^*$  mit den Beschriftungen *enter* bzw. *leave*.

Abb. 2 zeigt, dass jede Filiale drei Komponenten hat: den Eingang, den Gastraum und die Küche. Diese Komponenten werden wiederum als Module modelliert. Die linke Schnittstelle des Eingangs ist leer, die rechte Schnittstelle enthält zwei Petrinetz-Plätze. Die Beschriftungen dieser beiden Plätze stimmen mit den Beschriftungen der Plätze der linken Schnittstelle des Gastraums überein; gleich beschriftete Plätze werden später beim Komponieren der beiden Module miteinander verschmolzen. Die beiden Transitionen der Schnittstellen des *branch*-Moduls liegen auch in den Schnittstellen des Gastraums. Die rechte Schnittstelle des Gastraums enthält einen Platz und eine Transition; entsprechend beschriftete Elemente enthält die linke Schnittstelle des Küchen-Moduls.

Die Beschriftungen der Schnittstellen-Elemente weisen auf das Verhalten und die Funktionalität der Module hin: am Eingang wird jedem Gast ein freier Tisch zugewiesen. Im Gastraum kann der Gast seine Bestellung aufgeben, später das bestellte Gericht entgegennehmen und an seinem Tisch verspeisen.

## 4.2 Statische Komponenten der Filialen

In jeder Filiale spielen vier Mengen eine zentrale Rolle: die Tische, die Gäste, die Speisekarte mit ihren einzelnen Einträgen und die zubereiteten Gerichte. In jeder Filiale kann jede dieser

set symbols	function symbols
Clients	$f: \text{Meal items} \rightarrow \text{Menu}$
Tables	$g: \text{Meals} \rightarrow \text{Orders}$
Menu (order items)	
Meal items	properties for $A \subseteq \text{Meal items}$ $g(A) = \{f(a) \mid a \in A\}$
derived symbols	
Orders: subsets of Menu	
Meals: subsets of Meal items	

Abb. 3: Signatur  $\Sigma_0$ 

variables	sets	functions
c: Clients	Clients: all persons with an id card	$f: \text{Meal items} \rightarrow \text{Menu}$
t: Tables	Tables: $\{t_1, t_2, t_3, t_4\}$	$f(\text{rice}) = \text{rice}$
X: Orders	Menu: $\{\text{rice}, \text{meat}, \text{salad}\}$	$f(\text{meat}) = \text{meat}$
y: Meal items	Meal items: $\{\text{rice, rice, meat, salad}\}$	$f(\text{salad}) = \text{salad}$
Y: Meals		
	derived symbols	g: Meals $\rightarrow$ Orders
	Orders: $\{\text{rice, meat}, \{\text{rice, salad}\}\}$	$g(\text{rice, meat}) = \{\text{rice, meat}\}$
	Meals: $\{\{\text{rice, meat}\}, \{\text{rice, salad}\}\}$	$g(\{\text{rice, salad}\}) = \{\text{rice, salad}\}$

Abb. 4:  $\Sigma$ -Struktur  $S_0$ 

Mengen jeweils aus anderen Elementen bestehen. Beispielsweise arbeitet eine kleine Filiale mit weniger Tischen und einer anderen Auswahl auf der Speisekarte als eine große Filiale.

Um dennoch für alle Filialen die Abläufe gleich zu formulieren, verwenden wir eine Signatur,  $\Sigma_0$ , die Abb. 3 zeigt. Sie enthält die vier Symbole *Clients*, *Tables*, *Menu*, *Meal items* für die vier oben beschriebenen Mengen, dazu zwei Symbole (*Orders* und *Meals*) für Teilmengen und zwei Symbole ( $f$  und  $g$ ) für Funktionen. *Orders* steht für Bestellungen (eine Bestellung ist eine Teilmenge der Einträge in die Speisekarte), und *Meals* steht für Gerichte (ein Gericht ist eine Teilmenge zubereiteter Speisen). Die Funktion  $f$  ordnet jeder Speise den entsprechenden Eintrag in der Speisekarte zu;  $g$  ordnet jedem Gericht seine Bestellung zu.

In Abschnitt 3.2 wurde diskutiert, wie eine Signatur auf vielfältige Weise instanziert werden kann. Jede konkrete Filiale ist durch eine solche Instanzierung der Signatur  $\Sigma_0$  charakterisiert. In der Fallstudie diskutieren wir die Instanzierung  $S_0$  der Abb. 4.

### 4.3 Das Verhalten der drei Module auf Schema-Ebene

Zunächst modellieren wir das Verhalten jedes einzelnen der drei Module aus Abb. 2. Abb. 5 zeigt das Verhalten der einzelnen Module auf der Schema-Ebene, also auf Basis der Signatur  $\Sigma_0$ , und nicht einer einzelnen Instanzierung. Als mathematisches Konzept repräsentiert ein Platz ein Prädikat, das auf eine Menge von Objekten zutrifft. Diese Menge kann durch den Eintritt von Transitionen wachsen und schrumpfen.

Das *entry*-Modul in Abb. 5 verwaltet die Tische. Zunächst stellt sich hier die Frage nach einer angemessenen Anfangsmarkierung: Für eine gegebene Instanzierung, also eine konkrete Filiale, enthält der Platz *free tables* anfangs alle Tische als Marken. Im Beispiel der Struktur  $S_0$  aus Abb. 4 sind das die vier Marken  $t_1, t_2, t_3, t_4$ . Auf der schematischen Ebene kennen wir nur das Symbol *Tables*, für das jede Instanzierung eine Belegung mit einer Menge von Tischen frei wählen kann. Die zunächst naheliegende Idee, den Platz *free tables* anfangs mit dem Symbol *Tables* zu beschriften, greift allerdings zu kurz: Bei einer Instanzierung des Symbols *Tables* mit einer Menge, beispielsweise  $\{t_1, t_2, t_3, t_4\}$ , würde anfangs diese Menge als eine einzige Marke entstehen. Wir wollen aber die vier Elemente dieser Menge als einzelne Marken. Das wird mit *elm(Tables)* notiert.

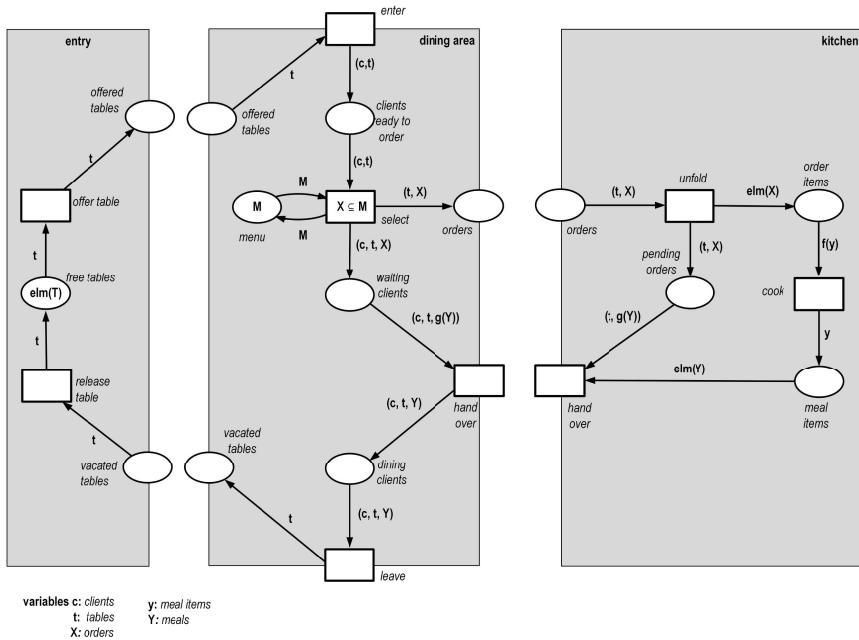


Abb. 5: Verhalten der Module auf der Schema-Ebene

Mathematisch steckt dahinter, dass ein Platz ein Prädikat darstellt, das aktuell auf die Elemente im Platz zutrifft. Der Platz *free tables* mit der Inschrift *elm(Tables)* steht also für den logischen Ausdruck:

$$\forall t \in Tables : free\ tables(t) \quad (1)$$

Die Instanzierung  $S_0$  erzeugt damit in der Anfangsmarkierung den logischen Ausdruck „ $\forall t \in \{t_1, t_2, t_3, t_4\} : free\ tables(t)$ “. Damit liegen auf dem Platz *free tables* anfangs die vier Marken  $t_1, t_2, t_3, t_4$ .

In einer gegebenen Instanzierung  $S$  kann im Gastraum-Modul die Transition *enter* eintreten, sobald eine Marke auf *offered tables* vorliegt, indem die Variable  $t$  mit dieser Marke belegt wird. Die Belegung der Variablen  $c$  kann frei gewählt werden aus der Menge, mit der die Instanzierung  $S$  das Mengensymbol *Clients* instanziert. Die Idee dahinter: Das Modul kann mit einem anderen Modul aus seiner Umgebung komponiert werden, indem die Belegung von  $c$  mit einem Gast einen inhaltlichen Sinn hat.

Bei der Transition *select* ist das Verhältnis zwischen dem Symbol *Menu* und der Variablen  $X$  interessant: Eine Instanzierung  $S$  des Gastraum-Moduls legt *Menu* als eine Speisekarte, also eine Menge einzelner Einträge fest. Hingegen wird beim Eintreten von *select* die Variable  $X$  jedes mal neu belegt. Die Bedingung  $X \subseteq Menu$  der Inschrift von *select* garantiert, dass die Belegung von  $X$  („Bestellung“) nur Einträge enthält, die in der Speisekarte aufgeführt sind.

Eine *order* besteht aus einem Tisch und einer Bestellung. Jede Marke auf dem Platz *waiting* besteht aus einem Kunden  $c$ , seinem Tisch  $t$ , und seiner Bestellung  $X$ . Mit der Transition *hand over* wird dem Gast das bestellte Gericht ausgehändigt.

Im *kitchen*-Modul zerlegt die Transition *unfold* jede eingehende Bestellung in ihre einzelnen Einträge, formuliert mit Hilfe des *elm*-Operators, analog zur Verwendung im *entry*-Modul. Für jeden dieser Einträge  $f(y)$ , also jede Benennung einer Speise, wird die Speise  $y$  gekocht (Transition *cook*). Entsprechend einer vorliegenden Bestellung auf dem Platz *pending orders* werden dann Speisen als  $Y$  zusammengestellt und als Gericht an den Kunden übergeben.

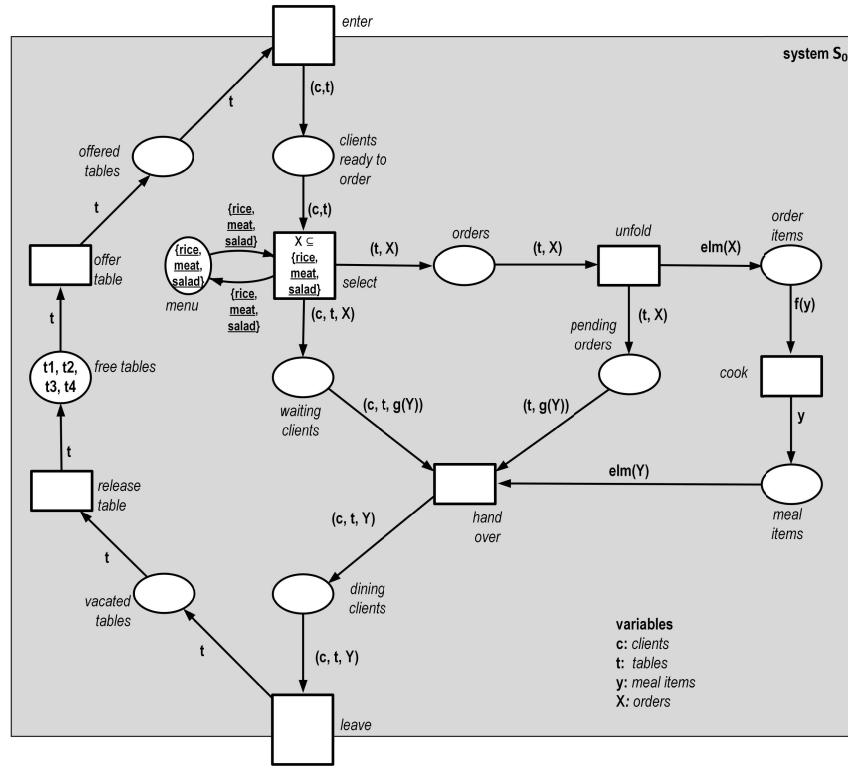
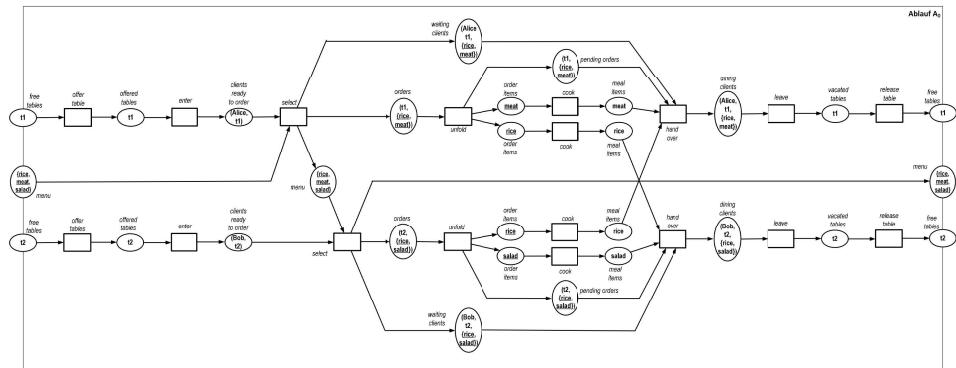
#### 4.4 Die Instanziierung $S_0$ und ihre Verhalten

Abb. 6 zeigt ein Modul,  $S_0$ , das aus den Modulen von Abb. 5 in zwei Schritten entsteht: erstens werden die drei Module aus Abb. 5 zu einem einzigen Modul, System  $S_0$ , komponiert; zweitens wird die Signatur  $\Sigma_0$  von Abb. 5 mit der Struktur  $S_0$  instanziert. Insbesondere enthält der Platz *free tables* jetzt vier Marken, und der Platz *menu* drei Marken.

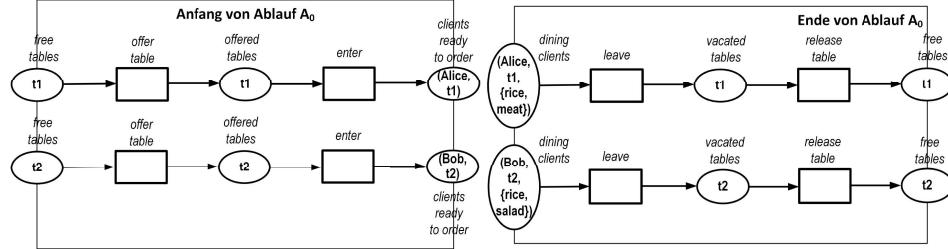
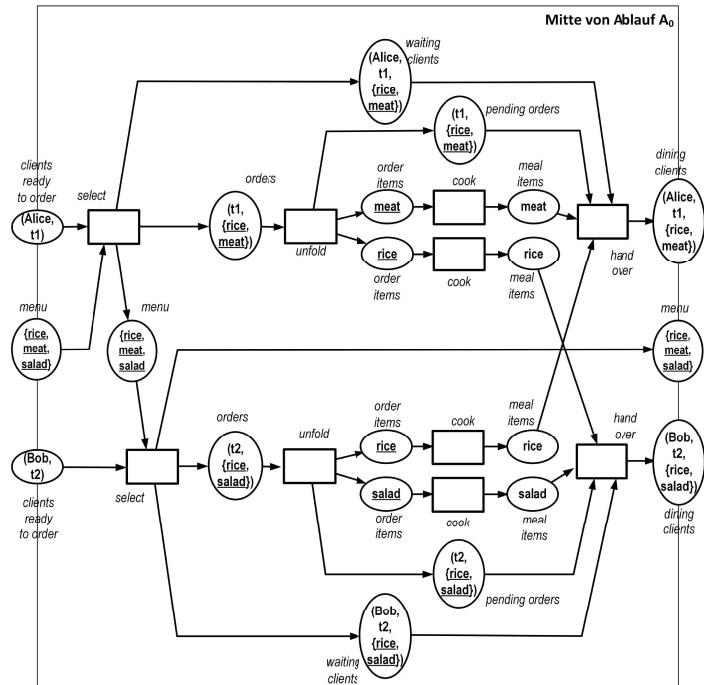
In der Instanziierung  $S_0$  kann man nun den Eintritt einzelner Transitionen dokumentieren. Im Anfangszustand kann beispielsweise die Transition *offer table* mit der Belegung  $t = t_1$  eintreten und dabei die Marke  $t_1$  auf den Platz *offered tables* verschieben. Danach kann *enter* mit der Belegung  $t = t_1$  und einem frei gewählten Kunden für  $c$  eintreten. Unabhängig davon kann *offer table* auch mit  $t = t_2$  eintreten, etc. So kann beispielsweise ein Ablauf entstehen, in dem der Gast *Alice* aus der Speisekarte eine Bestellung für Reis und Fleisch zusammenstellt, für die in der Küche die entsprechenden Speisen gekocht und an Alice übergeben werden. Daneben kann *Bob* ein Gericht mit Reis und Salat bestellen und ausgehändigt bekommen. Solches einzelnes Verhalten wird üblicherweise als Sequenz aus globalen Zuständen und eintretenden Transitionen beschrieben, beginnend mit der Anfangsmarkierung.

HERAKLIT modelliert ein einzelnes Verhalten als verteilten Ablauf (vgl. Abschnitt 3.3); formal gefasst als ein Modul. Damit können verteilte Abläufe einfach komponiert werden. Am laufenden Beispiel wird zudem deutlich, dass verteilte Abläufe Einzelheiten der Zusammenhänge innerhalb eines Ablaufs sehr viel expliziter zeigen als es mit sequentiellen Abläufen möglich ist.

Abb. 7 zeigt einen verteilten Ablauf,  $A_0$ , des Systems  $S_0$  in Abb. 6. Zur besseren Lesbarkeit komponieren wir diesen Ablauf aus drei Teilen: Das Modul *Anfang* von  $A_0$  in Abb. 8 besteht aus zwei Strängen. Der obere beginnt mit dem Eintritt der Transition *offer table* aus Abb. 6 im Modus  $t = t_1$ . Dadurch enthält der Platz *offered tables* die Marke  $t_1$ . Diese Marke wiederum aktiviert die Transition *enter* mit der Belegung  $t = t_1$ , und einer frei wählbaren Belegung von  $c$ . Der Ablauf  $A_0$  wählt  $c = Alice$  und erzeugt damit die Marke  $(Alice, t_1)$  auf dem Platz *clients ready to order*. Unabhängig von diesem Strang beschreibt der untere Strang Schritte der Marke  $t_2$  und das Entstehen der Marke  $(Bob, t_2)$  auf dem Platz *clients ready to order*.


 Abb. 6: System  $S_0$ 

 Abb. 7: Ablauf  $A_0$ 

Das Modul *Mitte von  $A_0$*  in Abb. 10 ergänzt in seiner linken Schnittstelle die rechte Schnittstelle von *Anfang von  $A_0$*  um den Platz *menu* mit der Marke  $\{rice, meat, salad\}$ .

Abb. 8: Anfangsstück des Ablaufs  $A_0$ Abb. 9: Endstück des Ablaufs  $A_0$ Abb. 10: Mittelstück des Ablaufs  $A_0$ 

Diese Marke repräsentiert das einzige verfügbare Exemplar der Speisekarte. Alice und Bob schauen sie nacheinander an, und erzeugen jeweils eine *order*,  $(t_1, \{rice, meat\})$  und  $(t_2, \{rice, salad\})$ . An der Transition *unfold* beginnt im Modul *System S<sub>0</sub>* ein Pfeil mit der Inschrift *elm(X)*. Mit  $t = t_1$  und  $X = \{rice, meat\}$  zerlegt *elm(X)* im Modul den Anteil  $\{rice, meat\}$  der Marke  $(t_1, \{rice, meat\})$  in die Bestandteile *rice* und *meat*, analog zur Inschrift des Platzes *free tables* im Schema von Abb. 5. Die entsprechenden Speisen *rice* und *meat* werden einzeln gekocht. Mit der Bestellung des Tisches  $t_2$  wird entsprechend verfahren. In beiden Fällen wird *rice* bestellt. Da die Bestellungen nebenläufig vorliegen,

können die beiden gekochten *rice*-Portionen nicht eindeutig den Bestellungen zugeordnet werden. Im Ablauf  $A_0$  werden sie vertauscht.

Das Modul *Ende von Ablauf*  $A_0$  in Abb. 9 beendet nun die beiden Ablauf-Teile in offensichtlicher Weise. Die Komposition *Anfang von Ablauf*  $A_0 \bullet$  *Mitte von*  $A_0 \bullet$  *Ende von Ablauf*  $A_0$  der drei Module liefert genau das Modul *Ablauf Ablauf*  $A_0$  in Abb. 7.

## 5 Bezug von HERAKLIT zu anderen Modellierungsinfrastrukturen

Ein *Framework* zur Modellierung beschreibt wesentliche Konzepte zur Modellierung. Darauf aufbauend bietet eine *Modellierungsinfrastruktur* korrespondierende Modelle, Methoden, Techniken und Werkzeugen. Im Vergleich mit anderen Disziplinen wird generell *in der Praxis* der (Wirtschafts-) Informatik wenig mit Modellen gearbeitet. Es sind zahlreiche Frameworks vorgeschlagen worden, aber keines hat sich bisher wirklich durchgesetzt.

Ein Beispiel sind Diagramme der *Unified Modeling Language* (UML, [Ob17]) für Software-Systeme. Die verschiedenen Diagrammtypen veranschaulichen einige Aspekte eines Softwaresystems, aber unterstützen Analysefragen nur wenig. Ein anderes Beispiel sind die in der Wirtschaftsinformatik beliebten Diagramme der *Business Process Model and Notation* (BPMN, [Ob14]) oder *ereignisgesteuerte Prozeßketten* (EPK, [KNS92]). Solche Diagramme beschränken sich auf die Identifikation abstrakter Aktivitäten und die Darstellung des Kontrollflusses. Sie unterstützen Abstraktion und Komposition, helfen aber wenig beim Umgang mit konkreten oder abstrakten Daten und Gegenständen bei der Beschreibung des Verhaltens und beim Nachweis der Korrektheit von Verhaltensmodellen gegenüber einer Spezifikation. Die *Architektur integrierter Informationssysteme* (ARIS, [Sc01]) erlaubt zwar eine integrierte Beschreibung von Daten, Funktionen und Abläufen. Allerdings fehlen leistungsfähige Modularisierungskonzepte. Auch sind ARIS-Modelle nur bedingt formalisiert. Weit verbreitet sind Petrinetze; in ihrer klassischen Form modellieren sie allerdings lediglich verteilten Kontrollfluss. Zahlreiche Varianten und Verallgemeinerungen (insbesondere *Coloured Petri Nets* (CPN, [JK09])) integrieren konkrete Daten (in der Sprechweise von HERAKLIT: einzelne Strukturen). Typische Vorschläge für hierarchische Petrinetze ersetzen einzelne Transition durch ganze Netze. Statecharts [Ha87] verwenden eigenständige Modul- und Kompositionskonzepte; dabei werden gleich gelabtete Übergänge endlicher Automaten verschmolzen. [Gr20] schlagen die statechart-basierte Sprache *Gamma* vor und diskutieren eine Vielzahl von Kompositionsooperatoren.

Schließlich gibt es noch eine Reihe von Frameworks, die letztlich als Strukturen einer Signatur, oder als eine Signatur auffassbar sind. Dazu gehören Abstract State Machines (ASM, [Gu00]), event-B [Ab05] und Z [Sp92]. Andere Frameworks orientieren sich an der Prädikatenlogik, beispielsweise TLA [La02], FOCUS [Br97, BS01] und Aloy [Ja87].

Alle diese Frameworks werden zur Analyse und Simulation von Softwarewerkzeugen unterstützt und wurden in größeren Software-Entwicklungsprojekten eingesetzt. Allerdings

hat sich keine wirklich durchgesetzt. Das liegt vorwiegend daran, dass bisher aus Modellen nicht besonders viel Nutzen gezogen werden kann. Derzeitige Frameworks fokussieren jeweils nur spezielle Bereiche eines Systems. Für kleine und mittelgroße Systeme sind die genannten Frameworks mehr oder weniger hilfreich. Aber da, wo man es besonders braucht, bei wirklich großen Systemen, fehlen systematische, strukturierende und abstrahierende Prinzipien zur Modellierung. Komplexe Systeme werden bestenfalls in Teilen modelliert, oft mit ganz unterschiedlichen Frameworks, die nur mühsam zusammenpassen. Es gibt keine umfassenden Modellsichten, in die dann ganz unterschiedliche Teilmodelle integriert werden können.

Im Vergleich mit den genannten Frameworks ist HERAKLIT breiter aufgestellt, indem Petrinetze, Konzepte algebraischer Spezifikation, und ein universeller Kompositionsoperator miteinander integriert sind. Keine der erwähnten Frameworks erreicht die Ausdruckskraft von HERAKLIT. Mit keiner der erwähnten Frameworks können alle Aspekte der Fallstudie des vierten Abschnitts modelliert werden.

Mit dem in der Logik grundlegenden und in der algebraischen Spezifikation bewährten Konzept einer Signatur und ihren unterschiedlichen Instanziierungen beschreibt HERAKLIT auch dynamisches Verhalten, sowohl auf der schematischen Ebene als auch für einzelne Instanziierungen. Das war auch die Intention von Prädikat-Transitionsnetzen [GL81].

HERAKLIT legt Wert auf Ausdrucksmittel für eine integrierte Modellierung lebensweltlicher und formaler Sachverhalte. Dazu nutzt HERAKLIT die wissenschaftstheoretischen Diskussionen um den Bezug zwischen der Welt und ihrer formalen Fassung, seit Aristoteles, die dann in die Prädikatenlogik eingeflossen ist [Su57]. HERAKLIT schlägt vor, dynamische Aspekte in die Prädikatenlogik einzubauen.

Wie bereits in der Einleitung erwähnt, hat Dijkstra vorgeschlagen, informelle und formale Argumentationen strikt zu trennen und zwischen ihnen eine „Firewall“ zu errichten. Diese Mauer ist nicht hilfreich zum Verständnis Informatik-basierter Systeme. Vielmehr braucht es eine Brücke, um zwischen der informellen Lebenswelt der Anwendung und der formalen Welt der Technik zu vermitteln. HERAKLIT bietet hierzu ein passendes Framework.

## 6 Ausblick

Die formalen Grundlagen und die Prinzipien des Einsatzes von HERAKLIT liegen vor; zahlreiche Fallstudien zeigen den Nutzen des umfassenden Ansatzes von HERAKLIT. Für die Verwendung im industriellen Maßstab und als bessere Alternative zu derzeit verwendeten Frameworks zur Modellierung fehlen insbesondere noch Softwarewerkzeuge zur Unterstützung des Entwurfs großer Modelle, und auf HERAKLIT zugeschnittene Analyseverfahren. Für spezielle Anwendungsbereiche werden im Lauf der Zeit Ausprägungen mit verfeinerten Konzepten gebildet. In einigen Anwendungen ist auch die automatische Erzeugung von Programmcode für einige Module wünschenswert.

## Literaturverzeichnis

- [Ab05] Abrial, J.-R.: The B-Book – Assigning Programs to Meanings. Cambridge University, 2005.
- [Bo01] Bowen, Jonathan P: Z: A formal specification notation. In: Software specification methods, S. 3–19. Springer, 2001.
- [Br97] Broy, Manfred: Compositional Refinement of Interactive Systems. *Journal of the ACM*, 44(6):850–891, 1997.
- [BS01] Broy, Manfred; Stolen, Ketil: Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement. Springer, 2001.
- [Di89] Dijkstra, E. W.: Reply to comments. *Commun. ACM*, 32(12):1414, 1989.
- [FR21a] Fettke, Peter; Reisig, Wolfgang: Handbook of HERAKLIT. HERAKLIT working paper, v1.1, September 20, 2021, <http://www.haraklit.org>, 2021.
- [FR21b] Fettke, Peter; Reisig, Wolfgang: Modelling Service-Oriented Systems and Cloud Services with HERAKLIT. In (Zirpins, Christian; Paraskakis, Iraklis; Andrikopoulos, Vasilios; Kratzke, Nanc; Pahl, Claus; El Ioini, Nabil; Andreou, Andreas S.; Feuerlicht, George; Lamersdorf, Winfried; Ortiz, Guadalupe; Van den Heuvel, Willem-Jan; Soldani, Jacopo; Villari, Massimo; Casale, Giuliano; Plebani, Pierluigi, Hrsg.): Advances in Service-Oriented and Cloud Computing. Springer International Publishing, Cham, S. 77–89, 2021.
- [GL81] Genrich, Hartmann J.; Lautenbach, Kurt: System modelling with high-level Petri nets. *Theoretical Computer Science*, 13:109–135, 1981.
- [Gr20] Graics, Bence; Molnár, Vince; Vörös, András; Majzik, István; Varró, Dániel: Mixed-Semantics Composition of Statecharts for the Component-Based Design of Reactive Systems. *Software and Systems Modeling*, 19(6):1483–1517, 2020.
- [Gu00] Gurevich, Yuri: Sequential abstract-state machines capture sequential algorithms. *ACM Transactions on Computational Logic*, 1:77–111, 2000.
- [Ha87] Harel, David: Statecharts: A Visual Formalism for Complex Systems. *Science of Computer Programming*, 8(3):231–274, 1987.
- [Ja87] Jackson, Daniel: Alloy: A Language and Tool for Exploring Software Designs. *Communications of the ACM*, 62(9):66–76, 1987.
- [JK09] Jensen, Kurt; Kristensen, Lars M.: Coloured Petri Nets: Modelling and Validation of Concurrent Systems. Springer, 2009.
- [Jo91] Jones, Clifford B.: Systematic software development using VDM. Prentice Hall, 2. Auflage, 1991.
- [KNS92] Keller, Gerhard; Nüttgens, Markus; Scheer, August-Wilhelm: Semantische Prozeßmodellierung auf der Grundlage „Ereignisgesteuerter Prozeßketten (EPK)“. Bericht 89, Veröffentlichungen des Instituts für Wirtschaftsinformatik (IWI) an der Universität des Saarlandes, 1992.
- [La02] Lamport, Leslie: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley, 2002.

- [Ob14] Object Management Group: Business Process Model and Notation (BPMN): Version 2.0.2. Bericht formal/2013-12-09, Object Management Group, 2014.
- [Ob17] Object Management Group: OMG Unified Modeling Language (OMG UML): Version 2.5.1. Bericht formal/2017-12-05, Object Management Group, 2017.
- [Pe77] Petri, C. A.: Non-Sequential Processes. Bericht ISF-77-5, Gesellschaft für Mathematik und Datenverarbeitung, St. Augustin, Federal Republic of Germany, 1977.
- [Re19] Reisig, Wolfgang: Associative composition of components with double-sided interfaces. *Acta Informatica*, 56(3):229–253, 2019.
- [Sc01] Scheer, August-Wilhelm: ARIS: Modellierungsmethoden, Metamodelle, Anwendungen. Springer, 4. Auflage, 2001.
- [Sp92] Spivey, John Michael: The Z Notation: A reference manual. Prentice Hall, 2. Auflage, 1992.
- [ST12] Sanella, Donald; Tarlecki, Andrzej: Foundations of Algebraic Specification and Formal Software Development. Springer, 2012.
- [Su57] Suppes, Patrick: Introduction to Logic. Van Nostrand Reinhold, 1957.