Komposition und Interoperabilität von IT-Systemen

Johannes Reich, johannes.reich@sap.com

9. Januar 2021

Zusammenfassung

In diesem Artikel diskutiere ich Interoperabilität von berechenbaren Systemen mithilfe des mathematischen Konzepts der Komposition. Die zentrale Vorstellung ist, dass Systeme durch Interaktion komponieren und diese Komposition mathematisch als Operator, also als Funktion auf Systemen beschreibbar ist. Daraus ergibt sich der Begriff des Interface eines Systems, als Zusammenfassung aller seiner auf die Komposition bezogenen Angaben, auf natürliche Art und Weise ebenso wie der Begriff der Komponente als eines Systems, das für eine bestimmte Komposition konstruiert wurde.

Ich unterscheide zwei grundsätzlich verschiedene Kompositionen von Systemen. Zunächst die hierarchische Komposition, die Systeme als Ganzes erfasst und zur Bildung von Supersystemen führt. Sie basiert auf den Kompositionsregeln berechenbarer Funktionen und führt zu den Interfaces der Operationen (+Events) sowie zu Komponentenhierarchien. Demgegenüber steht eine zweite Form der Komposition, die Systeme nur partiell im Sinne einer Projektion erfasst und zu vergleichsweise loser Kopplung führt, mit dem Interface der Protokolle. Damit sind Interfaces, die Operationen repräsentieren, per Definitionem nur für hierarchische Kompositionen geeignet und keine wesentliche Hilfe in der Darstellung netzwerkartiger "horizontaler" Interaktionen.

Um die Anwendbarkeit des Kompositionskonzepts zu demonstrieren, werden verschiedene System-, Interface- und Komponentenmodelle aus der Literatur diskutiert, u.a. das Komponentenmodell verteilter Objekte, sowie die Interfacekonzepte von SOA und REST.

1 Einleitung

Politik und Informatik schätzen die Fähigkeit, Interoperabilität von IT-Applikationen effizient zu erreichen, mittlerweile als eine der Schlüsselqualitäten ein, um die gewünschten Netzwerkeffekte im Internet der Dinge, etwa in der vernetzten Produktion, auszuschöpfen [16]. Die Bundesregierung [7] nennt Interoperabilität neben Souveränität und Nachhaltigkeit sogar als dritte Säule der Gestaltung digitaler Ökosysteme.

Wer nun erwartet, von der Informatik zu der Frage, wie man interoperable IT-Komponenten möglichst effizient baut, klare Antworten zu erhalten, sieht sich gegenwärtig mit einer kaum mehr überschaubaren Anzahl verschiedener Theorie- und Technologieansätze konfrontiert. Die verschiedenen zu Grunde liegenden Modelle, in Verbindung mit unterschiedlichen — oder schlimmer — mit denselben Terminologien über Agenten, Programme, Prozesse, Kalküle, Dienste, Zwillinge, Schalen etc. beeinträchtigen das gegenseitige Verstehen — einen Zustand, den Leslie Lamport als "Whorfian syndrome" bezeichnet hat [21].

Mein Vorschlag diesem Syndrom zu begegnen ist, für die Diskussion der Interoperabilität von IT-Systemen den Begriff der Komposition in den Vordergrund zu stellen: Zwei (oder mehr) Systeme sind interoperabel immer bezogen auf eine definierte Komposition. Den verschiedenartigen Interaktionen von IT-Systemen entsprechen verschiedenartige Kompositionen. Das unterschiedliche Kompositionsverhalten war auch der Hauptgrund für David Harel und Amir Pnueli in ihrer richtungsweisenden Arbeit [18] "reaktive" gegenüber "transformationalen" (diskreten) Systemen zu unterscheiden. Stavros Tripakis [27] hält Komposition und Kompositionalität für die vielleicht wichtigsten Konzepte modernen Systemdenkens.

2 Die Komposition von Systemen

Mathematisch bedeutet Komposition aus zwei oder mehr mathematischen Objekten mithilfe einer mathematischen Abbildung eines zu machen. So können wir aus zwei Funktion $f, g : \mathbb{N} \to \mathbb{N}$, die die natürlichen Zahlen auf sich selbst abbilden mit Hilfe des Konkatenationsoperators $\circ$ eine Funktion $h = f \circ g$ erklären durch $h(n) = f(g(n))$.

Wenden wir diese Vorstellung auf unsere interagierende Systeme an, die wir mit $\mathcal{S}_1, \ldots, \mathcal{S}_n$ benennen, so können wir, unabhängig von der konkreten Darstellung dieser Systeme ihre Komposition zu einem Supersystem mittels eines entsprechenden Kompositionsoperators $C_{\mathcal{S}}$ als partielle Funktion[1] für Systeme erklären:

$$\mathcal{S}_{ges} = C_{\mathcal{S}}(\mathcal{S}_1, \ldots, \mathcal{S}_n). \tag{1}$$

Den ersten Nutzen, den wir aus dieser Definition ziehen können, ist, dass wir nun die Eigenschaften des Supersystems klassifizieren können in solche, die sich vergleichsweise einfach aus den gleichen Eigenschaften der Teilsysteme ergeben und solche, die sich aus anderen Eigenschaften der Teilsysteme ergeben:

Definition 1 *Ich nenne eine Eigenschaft $E(\mathcal{S}_{ges})$ eines zusammengesetzten Systems $\mathcal{S}_{ges}$ "(homogen) kompositional" bezüglich der Komposition $C_{\mathcal{S}}$, wenn ein Operator C_E existiert, so dass sich $E(S_{ges})$ als $C_E(E(\mathcal{S}_1), \ldots, E(\mathcal{S}_n))$ ergibt, also gilt:*

$$E(C_{\mathcal{S}}(\mathcal{S}_1, \ldots, \mathcal{S}_n)) = C_E(E(\mathcal{S}_1), \ldots, E(\mathcal{S}_n)) \tag{2}$$

[1] Partiell bedeutet, dass diese Funktion nicht für alle möglichen Systeme erklärt ist, d.h. nicht jedes System ist für jede Komposition geeignet.

Andernfalls nenne ich sie "emergent".

Ein einfaches Beispiel einer homogen kompositionalen Eigenschaft physikalischer Systeme ist ihre Masse: Die Masse eines Gesamtsystems ist gleich der Summe der Massen der Einzelsysteme. Ein einfaches Beispiel einer emergenten Eigenschaft eines physikalischen Systems ist die Resonanzfrequenz eines Schwingkreises aus Spule und Kondensator. Die Resonanzfrequenz des Schwingkreis ergibt sich nicht aus den Resonanzfrequenzen von Spule und Kondensator, weil diese keine solche Resonanzfrequenz aufweisen, sondern aus ihrer Induktivität und Kapazität.

Eine der in der Informatik wichtigsten Eigenschaft von Systemen ist die Berechenbarkeit ihrer Systemfunktion. Basierend auf Überlegungen von Kurt Gödel konnte Stephen Kleene in seiner bahnbrechenden Arbeit [20] zeigen, das diese Eigenschaft in der Tat per Konstruktionem kompositional ist. Ausgehend von gegebenen Elementaroperationen (Nachfolger, Konstante und Identität) lassen sich nämlich alle weiteren berechenbaren Operationen durch die folgenden 3 Regeln konstruieren (F_n sei die Menge aller Funktionen auf den natürlichen Zahlen mit Arität n):

1. *Comp:* Sei $g_1, \ldots, g_n \in F_m$ und $h \in F_n$ berechenbar, dann ist $f = h(g_1, \ldots, g_n)$ wiederum berechenbar.

2. *PrimRec:* Sind $g \in F_n$ und $h \in F_{n+2}$ berechenbar und $a \in \mathbb{N}^n$, $b \in \mathbb{N}$ dann ist die Funktion $f \in F_{n+1}$, gegeben durch $f(a,0) = g(a)$ und $f(a,b+1) = h(a,b,f(a,b))$ wiederum berechenbar.

3. *μ-Rec:* Sei $g \in F_{n+1}$ berechenbar und $\forall a \exists b$ so dass $g(a,b) = 0$ und die μ-Operation $\mu_b[g(a,b) = 0]$ ist definiert als das kleinste b mit $g(a,b) = 0$. Dann ist $f(a) = \mu_b[g(a,b) = 0]$ wiederum berechenbar.

Comp besagt, dass mit gegebenen berechenbaren Funktionen auch ihre nacheinander sowie ihre parallele Anwendung wiederum eine berechenbare Funktion ist. *PrimRec* definiert die einfache Rekursion, die sich in imperativen Programmiersprachen als FOR-Schleife mit einer im Vorhinein definierten Anzahl an rekursiven Schritten wiederfindet. Die dritte Regel *μ-Rec* besagt, dass eine rekursive Berechnung auch als iterative Lösung eines berechenbaren Problems, im Falle der natürlichen Zahlen einer Nullstellensuche, vorliegen kann, bei dem man nicht von vornherein sagen kann, wie viele Schritte man zur ersten Lösung brauchen wird, ja u.U. noch nicht einmal weiß, ob eine solche Lösung überhaupt existiert. Dem entspricht in imperativen Programmiersprachen die WHILE-Schleife.

2.1 Der Begriff des Interfaces und der Komponente

Der zweiten Nutzen, den wir aus unser Definition der Komposition ziehen können, ist eine klare Definition des Begriffs des Interfaces und der Komponente. Wir benötigen nämlich nun einen Begriff, der all das zusammenfasst, was ein

Kompositionsoperator von einem System wissen muss. [27, 14] verwenden dafür den Begriff des *Interface*, einem Vorschlag, dem ich mich gerne anschließe. Damit wird die Frage, was eigentlich ein Interface ist, entscheidbar.

Das Rezept sieht so aus, dass diejenige, die behauptet, ein mathematisches Objekt sei ein Interface, als erstes ein Systemmodell vorweisen muss, zweitens zeigen muss, wie ihr Kompositionsoperator aussieht, damit drittens nachvollziehbar wird, welche Anteile des Systemmodells zum Interface gehören.

Eine Komponente lässt sich als ein System verstehen, das für eine bestimmte Komposition vorgesehen ist und daher entsprechend wohldefinierte Interfaces aufweist, die, per Definitionem, die intendierte Komposition ausdrücken.

2.2 Ersetzbarkeit und Kompatibilität

Der dritte Nutzen ergibt sich in der Möglichkeit, mittels unseres Kompositionskonzepts Ersetzbarkeit und Abwärts- und Aufwärtskompatibilität zu definieren.

Zu diesem Zweck betrachten wir die beiden Systeme $\mathcal{A}$, $\mathcal{A}'$. Der Kompositionsoperator C, macht aus $\mathcal{A}$ und einem weiteren System $\mathcal{B}$ das Supersystem $\mathcal{S} = C(\mathcal{A}, \mathcal{B})$. Das System $\mathcal{A}$ kann durch das System $\mathcal{A}'$ in dieser Komposition sicherlich ersetzt werden, wenn auch gilt $\mathcal{S} = C(\mathcal{A}', \mathcal{B})$.

Stehen nun aber $\mathcal{A}'$ und $\mathcal{A}$ in einem zu konkretisierenden Sinne in einer Erweiterungsrelation, notiert als $\mathcal{A} \sqsubseteq \mathcal{A}'$ und macht ein weiterer Operator C' aus $\mathcal{A}'$ und wiederum $\mathcal{B}$ das Supersystem $\mathcal{S}' = C'(\mathcal{A}', \mathcal{B})$, so dass die Eigenschaft der Erweiterung erhalten bleibt, also auch $\mathcal{S} \sqsubseteq \mathcal{S}'$ gilt, dann sprechen wir unter den folgenden Bedingungen von "*Kompatibilität*"

Bringt C' auf das alte System $\mathcal{A}$ angewandt immer noch das alte System $\mathcal{S} = C'(\mathcal{A}, \mathcal{B})$ hervor, dann verhält sich $\mathcal{A}$ im Kontext der Komposition C' "*aufwärtskompatibel*". Gilt jetzt $\mathcal{S} = C(\mathcal{A}', \mathcal{B})$, lässt sich also $\mathcal{A}$ durch $\mathcal{A}'$ im Kontext C ersetzen, verhält sich $\mathcal{A}'$ im Kontext C "*abwärtskompatibel*".

Ein einfaches Beispiel ist eine rote Leuchtdiode $\mathcal{A}$ und eine Fassung $\mathcal{B}$, die zu einer rot leuchtenden Lampe $\mathcal{S} = C(\mathcal{A}, \mathcal{B})$ komponieren. Eine neue Leuchtdiode A', die die alte um die Fähigkeit erweitert, bei Stromumkehr grün zu leuchten, heißt abwärtskompatibel zu $\mathcal{A}$, wenn sie sich mit derselben Fassung ebenfalls zu einer ausschließlich rot leuchtenden Lampe machen lässt.

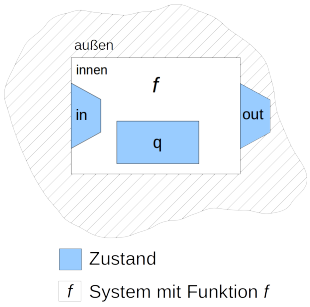
Eigentlich ist die zweifarbige LED natürlich für eine Lampenschaltung vorgesehen, die diese Zweifarbigkeit unterstützt. Da die alte Leuchtdiode in die unveränderte Fassung passt, ergibt sich trotz neuer Schaltung, die alte, einfarbig rote Lampe. Die alte LED verhält sich in diesem Kontext aufwärtskompatibel.

3 Die Anwendung des Kompositionsmodells

Zur Strukturierung der weiteren Diskussion schauen wir uns zunächst zwei grundsätzlich unterschiedliche Kompositionen an. Zum einen die Komposition von (einfacheren) Systemen zu (komplexeren) Supersystemen entlang der Regeln zur Konstruktion berechenbarer Funktionen und zum anderen die Komposition von Rollen zu Protokollen.

3.1 Die hierarchische Komposition von Systemen zu Supersystemen

Systeme grenzen ein Inneres vom Rest der Welt, der Umgebung ab [18, 10, 27]. Zur Demonstration des Vorgehens wähle ich ein einfaches Systemmodell das gemäß Abb. 1 die gesamte Eingabe und den inneren Zustand in einem Zeitschritt mittels einer (berechenbaren) Systemfunktion f auf eine Ausgabe und einen neuen inneren Zustand abbildet.



$$\begin{pmatrix} q(t') \\ out(t') \end{pmatrix} = \begin{pmatrix} f^{int}(q(t), in(t)) \\ f^{ext}(q(t), in(t)) \end{pmatrix}$$

Abbildung 1: Ein einfaches System mit seinen drei, ggfs. vektorwertigen Zustandsfunktionen $(in, out, q) : T \to I \times O \times Q$, die die Systemzeit T auf ihre jeweilige Wertemengen I, O, Q abbilden. Der Zusammenhang dieser Werte zu jedem Zeitpunkt wird durch die Systemfunktion f hergestellt, wie rechts dargestellt. t' ist der Nachfolgezeitpunkt von t.

Solche Systeme werden durch Zustände gekoppelt, die für die einen Systeme die Ausgabe und für die anderen Systeme die Eingabe repräsentieren, und die ich Shannonzustände nenne. Aus informatischer Sicht entspricht das der Informationsübertragung zwischen den Teilsystemen.

Es ist nicht schwer zu sehen [24], dass die sequentielle und die parallele Kopplung einfacher Systeme im Sinne einer Pipe zusammen der Komposition berechenbarer Funktionen gemäß Regel *Comp* entspricht. Zur Darstellung der Rekursion muss man das Systemmodell dahingehend erweitern, dass einzelne Eingabekomponenten unberücksichtigt bleiben können, was durch das leere Zeichen ϵ in Eingabekomponenten ausgedrückt werden kann. In Abb. 2 ist eine Komposition dreier Systeme, $\mathcal{S}_1$, $\mathcal{S}_2$ und $\mathcal{S}_3$, dargestellt, die zusammen ein Supersystem $\mathcal{S}_{ges}$ bilden mit der Systemfunktion $f_{ges}(x) = 2x + 5$ (Beispiel aus [26]).

Offensichtlich enthält die Darstellung des Supersystems $\mathcal{S}_{ges}$ ohne seine innere Struktur alle die Angaben, die man benötigt, um seinerseits dieses System wieder mit weiteren Systemen auf diese Art und Weise zu komponieren. Seine Systemfunktion ist daher notwendigerweise Teil seines Interfaces. D.h. in dieser Komposition verbirgt ein Interface die innere Systemstruktur, also die Art und Weise, wie die Systemfunktion berechnet wird — sozusagen den Algorithmus — aber nicht die Systemfunktion als solches.

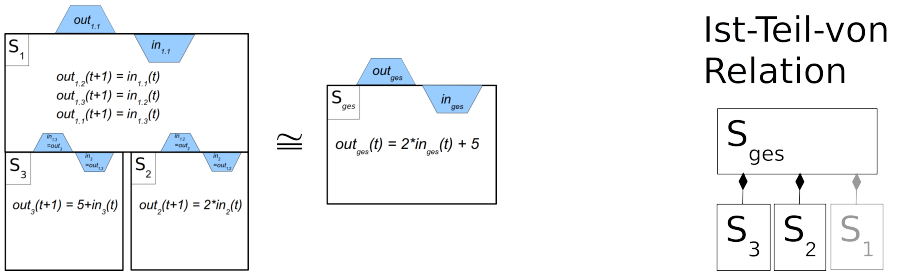


Abbildung 2: Links wird die Komposition dreier Systeme $\mathcal{S}_1$, $\mathcal{S}_2$ und $\mathcal{S}_3$ zu dem System $\mathcal{S}_{ges}$ mit der Funktion $f_{ges}(x) = 2x + 5$ dargestellt. Rechts wird die dabei entstehende "Ist-Teil-von"-Beziehung zwischen den Teilsystemen $\mathcal{S}_1$, $\mathcal{S}_2$ und $\mathcal{S}_3$ und dem Supersystem $\mathcal{S}_{ges}$ gezeigt, die durch die Interaktion entstehen. Das System $\mathcal{S}_1$ ist ausgegraut, weil es in dieser Art der Darstellung häufig weggelassen wird.

Diese Art der Komposition findet sich in modernen imperativen Programmiersprachen im Konstrukt der "*Operation*", bzw. des "*Objekts*". In der Programmiersprache C sähe diese Komposition folgendermaßen aus:

```
int s2(int n) {return(2*n);}
int s3(int n) {return(n+5);}
int s(int n) {return(s3(s2(n)));}
```

In dieser Beschreibung taucht das System $\mathcal{S}_1$ nicht explizit auf, weswegen es im rechten Teil der Abb. 2 ausgegraut ist. Operationen sind demnach tatsächlich mehr als nur Systemfunktionen, sondern sie sind Teil eines Kompositionsmechanismus für die hierarchische Komposition von Systemen, der ihre interne Struktur nach außen verbirgt. Eine einfache Folge ist, dass man mit dem sogenannten Aufruf einer gewöhnlichen Operation, deren Aufgabe in der Abbildung ihrer Eingangsdaten auf ihre Ausgangsdaten liegt, nicht aus seinem System hinausgelangen kann. Dies geht nur mit Operationen mit "Transportsemantik", d.h. mit Operationen, deren Korrektheit nur im Transport und nicht in der Verarbeitung liegt.

3.2 Lose Kopplung: Protokolle

Eine ganz andere Form der Kopplung von Systemen beruht darauf, diese nicht vollständig, sondern nur in Teilen in die Komposition einfließen zu lassen. Ich spreche daher auch von "*loser*" Kopplung. Dabei sind mit Teilen nicht "Teilsysteme" gemeint — diese würde ja wieder zu Supersystemen komponieren —, sondern Teile, die für sich alleine eben keine Systeme darstellen. Entsprechend komponieren diese Teile, ich nenne sie "*Rollen*", nicht zu Supersystemen, sondern zu etwas, was in der Informatik als "*Protokoll*" bezeichnet wird. Diese Rollen lassen sich als Projektionen der Systeme auf die Interaktionen, die durch

Protokolle beschrieben werden, verstehen [24]. Wir müssen unsere Kompositionsrelation (1) daher auf Rollen $\mathcal{R}_i$, die zu einem Protokoll $\mathcal{P}$ komponieren erweitern.

$$\mathcal{P} = C_{\mathcal{R}}^{Prot}(\mathcal{R}_1, \ldots, \mathcal{R}_n) \tag{3}$$

Die Systeme, für die diese Komposition passt, werden in der Literatur "*reaktiv*" [18] oder "*interaktiv*" [10, 24] genannt, oder auch als Prozesse bezeichnet. In Interaktionen weisen diese Systeme ein nichtdeterministisches, zustandsbehaftetes Verhalten auf und lassen sich aus diesem Grund, bezogen auf eine Interaktion, gar nicht mit einer Funktion beschreiben, die Eingabewerte auf Ausgabewerte abbildet.

Ein einfaches Beispiel für ein Protokoll ergibt sich aus dem in Abb. 3 dargestellten Problem der einspurigen Eisenbahnbrücke, die sich zwei Züge, $\mathcal{Z}_1$ und $\mathcal{Z}_2$, teilen müssen (s. a. [2]). Zu diesem Zweck interagieren beide Züge über ein Protokoll mit einem Kontroller $\mathcal{C}$, der sicherstellt, dass höchstens ein Zug gleichzeitig auf der Brücke ist.

Es gibt viele Formalismen, um Protokolle zu beschreiben (s.a. [19, 3, 24]). Ich verwende sogenannte Transducer, also Automaten, deren Transitionen einen Startzustand p und ein Eingabezeichen i auf einen Zielzustand q und ein Ausgabezeichen o abbilden, sich also darstellen lassen als $p \xrightarrow{i/o} q$. Sie eignen sich meiner Meinung nach besonders gut, weil ihre Kopplung genauso wie bei den einfachen Systemen durch die "ausgetauschten" Zeichen erfolgt, sich mit ihnen gut die Verwandtschaft von Protokollen zu Spielen zeigen lässt [23, 24] und sich auch ein interessantes Modell für die Bedeutung der ausgetauschten Zeichen ergibt [25].

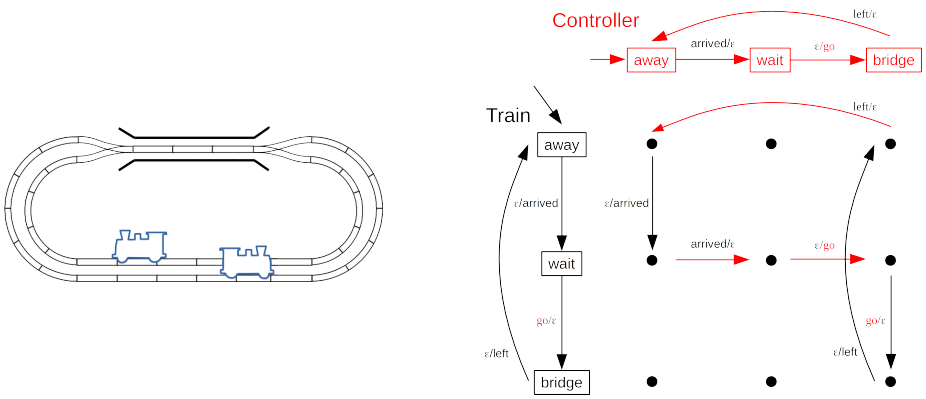


Abbildung 3: Die linke Abbildung zeigt zwei Züge, die sich eine gemeinsame Brücke teilen. Es darf sich höchstens ein Zug gleichzeitig auf der Brücke aufhalten. Um dies zu gewährleisten, kommunizieren beide Züge mit einem Kontroller. Die rechte Abbildung zeigt das Zustandsdiagramm des Protokolls, das Zug (schwarz) und Kontroller (rot) miteinander sprechen.

Die Rollen von Zug und Controller im Protokoll sind nur ein winziger Ausschnitt aus diesen tatsächlichen Systemen. Sowohl für den Zug als auch für den Kontroller wähle ich ein Modell aus drei Zuständen, $Q_{Z_{1,2}/C} = \{away, wait, bridge\}$. Das Eingabealphabet der Züge $I_{\mathcal{Z}_{1,2}} = \{go\}$ ist das Ausgabealphabet des Kontrollers $O_{\mathcal{C}}$ und ebenso ist das Ausgabealphabet der Züge $O_{\mathcal{Z}_{1,2}} = \{arrived, left\}$ gleich dem Eingabealphabet des Kontrollers $I_{\mathcal{C}}$. Initial sind Zug und Kontroller im Zustand "*away*". Kommt der Zug an, signalisiert er dem Kontroller "*arrived*", der in seinen *wait*-Zustand wechselt. Wenn der Zug aus Sicht des Kontrollers die Brücke passieren kann, signalisiert der Kontroller dies mit "*go*", während ihm der Zug mit "*left*" signalisiert, dass er die Brücke verlassen hat.

Das Protokoll zwischen Zug und Kontroller weist die wichtigen Eigenschaft der Konsistenz [24] auf: Es ist vollständig, da keine zusätzlichen externen Zeichen vorkommen, es ist wohlgeformt, in dem alle gesendeten Zeichen auch geeignet empfangen werden können. Es ist unterbrechbar, weist also keine unendlich langen Interaktionsketten auf und erfüllt seine Akzeptanzbedingung, dass es jeweils die drei Zug- und Kontrollerzustände unendlich oft durchläuft.

Werden die Protokolle komplexer, lassen sich Transitionen zu Äquivalenzklassen zusammenfassen, wie u.a. in [24] dargestellt, was zum Dokumentenbegriff führt.

3.3 Andere Systemmodelle

Im weiteren gebe ich eine Auswahl an Systemmodellen, die jeweils mit eigenen Kompositionsoperatoren einhergehen.

3.3.1 Blockdiagramme als Systeme

Hierarchische Blockdiagramme, wie sie etwa die Simulationsumgebungen Simulink und Ptolemy als "kleinsten gemeinsamen Nenner" anbieten, bestehen im Wesentlichen aus sogenannten Ports im Sinne getypter Variablen, die unseren Shannon-Zustandsfunktionen entsprechen, mittels derer die Blöcke verschaltet werden können [27, 14]. Ohne Bezug auf eine Abbildung gibt es jedoch kein Kriterium, dass die Zusammenfassung der Ports eigentlich rechtfertigt. Insbesondere bleibt dabei auch völlig unbestimmt, ob wichtige Eigenschaft, wie eine gewisse Lebendigkeit, durch Komposition erhalten bleiben oder nicht.

3.3.2 Manfred Broys Systemmodell

Ein Beispiel für eine Modell interaktiver Systeme, das allein auf Ein- und Ausgabehistorien beruht und auf die explizite Beschreibung eines internen Zustands verzichtet, liefert Manfred Broy [10]. Die Menge der (getypten) Ein- und Ausgabekanäle bezeichnet er als "syntaktisches Interface" und die Relation zwischen den Ein- und Ausgabehistorien als "semantisches Interface". Problematisch an diesem Ansatz ist, dass sich die Ein- und Ausgabehistorien zusammengesetzter Komponente nicht ausschließlich aus den Ein- und Ausgabehistorien der Teilkomponenten ergeben [9]. Stattdessen sind weitere Annahmen über Kausalität

im Sinne einer partiellen Ordnung der Ereignisse der Historien untereinander notwendig, die auch Manfred Broy entsprechend einführen muss.

3.3.3 Das Systemmodell BIP

Die Gruppe um Joseph Sifakis hat das Komponentenmodell "Behavior, Interaction, and Priority (BIP)" entwickelt [4] um interaktive Systeme zu beschreiben. Atomare Komponenten werden als finite Automaten beschrieben. Die Komposition verschiedener Komponenten entsteht durch Konnektoren, die Interaktionen beschreiben in dem Sinne, dass gegebene Bedingungen zu erfüllen sind unter denen Variablenwerte in vorgegebener Richtung zwischen mit ihren Ports registrierten Komponenten ausgetauscht werden. Dabei lassen sich Prioritäten angeben, mit denen unter zwei Interaktionen ausgewählt werden kann, falls beide ermöglicht wären.

Dieses Modell ist nach meinem Verständnis dem in [24] vorgestellten Modell für interaktive Systeme mit seiner Aufteilung des Zustands und seiner Regelorientierung recht ähnlich. Aber es wird nicht hinreichend klar zwischen dem Informationstransport und der Verarbeitung unterschieden, weil Berechnungen sowohl in den Automaten, als auch in den Konnektoren durchgeführt werden.

3.3.4 Synchrone reaktive Systeme

In der Verschaltung digitaler Systeme, wie logischer Gatter oder Flipflops, beziehen sich Ein- und Ausgabe u. U. auf "denselben" Zeitpunkt und nur das Update eines eventuell vorhandenen inneren Zustands bezieht sich auf den nächsten Zeitpunkt.

$$\begin{aligned} q(t') &= f^{int}(in(t), q(t)) \\ out(t) &= f^{ext}(in(t), q(t)) \end{aligned}$$

Dieses Systemmodell ist Grundlage der synchron reaktiven Systeme [5, 2] und hat seinen Niederschlag in einer ganzen Reihe von Hardwarebeschreibungssprachen gefunden, wie etwa Lustre, Esterl, Signal oder VHDL/Verilog.

3.4 Andere Interface und Komponentenmodelle

Unter dem Blickwinkel der Komposition müssen die Interface- und Komponentenmodelle wenigstens so vielfältig sein, wie die Systemmodelle mit ihren verschiedenen Kompositionskonzepten. Die Vorstellungen von Manfred Broy hatte ich schon beschrieben. Luca de Alfaro und Thomas Henzinger gehen in [14] vom Systemmodell des Blockdiagramms aus, das nicht primär auf das Abbildungsverhalten zielt. Ihr Interfacekonzept scheint mir eher eine Spezifikation zu sein, wofür spricht, dass ihr Interface im Gegensatz zu ihrer Komponente Anforderungen an die Umgebung stellen kann, etwa dass eine bestimmte Eingangsgröße ungleich Null sein muss. Das erklärt vielleicht auch ihre Einschätzung, "Nondeterminism in interfaces ... seems unnecessary", die, verstanden als Interface in meinem Sinn, falsch wäre.

Den Erfolg des Interfacekonzepts der Operation lässt sich daran ablesen, dass inzwischen hierarchisch komponierende Komponentenmodelle allgegenwärtig sind. Für viele Programmiersprachen gibt es mittlerweile Paketmanager, von denen jeder mehr als 100.000 Pakete verwaltet [12].

Problematisch wird es, wenn die Auffassung vertreten wird, dass alleine den Operationen ein Interfacecharakter zukommt (z.B. aktuell [8]). In ihrem Überblick über Komponentenmodelle unterscheiden Ivica Crnkovic et al. [13] zwischen "operationsbasierter" und "portbasierter" Schnittstellenunterstützung und zeigen damit auf, dass viele der derzeit wichtigen Komponentenmodelle tatsächlich die Deklaration von Protokollschnittstellen nicht unterstützen.

3.4.1 Verteilte Objekte

Verteilte Objektmodelle wurden unter der Vorstellung entwickelt, dass die Kapselung des internen Zustands durch eine vorab definierte Menge an Operationen im Sinne eines abstrakten Datentyps diesen Zustand gegenüber der Außenwelt "verbirgt" und ein solches Objekt "autonom" sei, womit eine "lose" Kopplung zwischen Applikationen erreicht werden könne [11]. Entsprechende Komponentenmodelle sind etwa CORBA, DCOM oder auch OPC-UA.

Tatsächlich aber lässt sich nach unserer Definition nur die kompositionale Struktur hinter einem objektorientierten Interface verbergen, nicht aber die Logik der jeweiligen Abbildung. Damit werden entfernte Objekte logisch gesehen genauso zu einem Teil der "einen Applikation" wie lokale Objekte und von "loser Kopplung" kann keine Rede sein.

3.4.2 Service orientierte Architektur (SOA)

OASIS definiert eine SOA eher vage als "paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains." [22]. Ein "Service" wird definiert als "The performance of work (a function) by one for another." und als "mechanism by which needs and capabilities are brought together". Eine SOA wird aktuell etwa für die Industrie 4.0 [15] oder in der NATO [1] (Vol. 2) propagiert.

Tatsächlich geht keine der Servicedefinitionen auf dessen Transformationsverhalten ein, etwa ob er eine Funktion zu repräsentieren hat, also ein deterministisches Verhalten zeigen sollte, oder nicht [28]. Damit sind SOA-Interfaces nach dem vorgestellten Modell der Komposition nicht wohldefiniert.

Problematisch ist die Namensgebung als "Dienst" v.a. weil in dem Gebiet der Ökonomie sich eine "Dienstleistung" gerade nicht als einfache Funktion darstellt, wie es die SOA mit ihrer WSDL-Interfacesyntax suggeriert. Um das Beispiel aus [26] aufzugreifen: Zum Streichen einer Wand in einem neu gebauten Haus muss man Angebote einholen, ein Angebot annehmen, Termine vereinbaren und ggfs. neu arrangieren, das Ergebnis überprüfen und im Falle der Annahme die Rechnung bezahlen und schließlich die Dokumente für die Steuererklärung aufbewahren — ein Verhältnis voller Zustand, Asynchronität und Nichtdeterminismus.

Eben aus diesem Grund beschreibt die Ökonomie diese Interaktionen mit Spielen, deren informatisches Pendant die Protokolle sind [23, 24].

3.4.3 Representational State Transfer (REST)

Der Representational State Transfer (REST) [17] kann als der Versuch angesehen werden, die Prinzipien der zustandslosen Kommunikation zusammen mit der semantischen Agnostik - beides Prinzipien des sehr erfolgreichen Hypertext Transfer Protokolls (HTTP) - auf die Interaktionen von Netzwerkanwendungen zu übertragen. Gegenwärtig wird es oft als eine einfachere Variante von SOA positioniert. Ein REST-Aufruf soll dem Prinzip der Adressierbarkeit genügen, dass jede Ressource eine eindeutige URI haben muss, und der Zustandslosigkeit, dass jede REST-Nachricht alle Informationen enthalten soll, die für die von ihr initiierte Verarbeitung notwendig sind.

In gewisser Weise beruht REST meiner Meinung nach auf einem Missverständnis über die Rolle des Zustands in verteilten Applikationen. Wollte ich etwa einer Bank bei der Überweisung alle Informationen mitgeben, die diese für die weitere Verarbeitung benötigt, dann wüsste sie meinen Kontostand nicht. Ein interessanter Gedanke. Das eigentliche Transformationsverhalten ist explizit nicht Teil der Semantik eines REST-Aufrufs, ebenso wenig eine eventuelle Beziehung zwischen verschiedenen REST-Aufrufen. Entsprechend repräsentieren REST-"Interfaces" eher generische Transportfunktionen, was auch ihre vermeintliche "Vielseitigkeit" erklärt. Überträgt man die erfolgreichen "Prinzipien" der Informationsübertragung auf die Informationsverarbeitung, erhält man eben nur Informationsübertragung.

4 Zusammenfassung und Ausblick

Das Kompositionskonzept gibt uns mit vergleichsweise geringem mathematischen Aufwand einen mächtigen Begriff in die Hand, um über Interoperabilität zu reden. Es ergeben sich recht zwanglos der Begriff des Interface, der Komponente, sowie der Begriff der Ersetzbarkeit und der Kompatibilität, ohne dass wir uns auf die konkrete Struktur eines Systemmodells beziehen müssen.

Je nachdem, unter welchem Kompositionskonzept wir die Welt betrachten, sieht sie scheinbar gänzlich anders aus: In der Welt der vollständigen Systeme dominiert die Systemfunktion und die hierarchische Komposition. Für den Informatiker ist dies die Welt der Berechenbarkeit, in der wir die Verarbeitbarkeit der Informationen durch ein Datentypsystem garantieren, Operationen im Wesentlichen mit Verben in Imperativform benennen und die explizite Darstellung von Kommunikation eliminieren können.

In der Welt der Interaktionsnetzwerke, in der viele Systeme vergleichsweise unabhängig agieren, eigene Entscheidungen treffen und kein System das Verhalten aller anderen Systeme bestimmt, dominieren die Protokolle. Für den Informatiker ist dies die Welt der nichtdeterministischen Zustandsautomaten, die ihre Zustandsübergänge für die Empfänger so dokumentieren, dass sie von

diesen "verstanden" werden können. Der Informationsaustausch spielt in der Darstellung eine zentrale Rolle, die ausgetauschten Informationen erhalten Dokumentencharakter und werden von uns mit Hauptwörtern bezeichnet: Bestellung, Rechnung, Überweisung, etc.

Beide Weltsichten sind nicht scharf voneinander abgegrenzt: Was in einer Interaktion nichtdeterministisch erscheint, kann durch eine weitere Interaktion erzwungen sein. Ebenso kann eine Operation Daten nur transportieren und nicht "verarbeiten"; bzw. eine eigentlich vorgesehene Verarbeitung kann ausnahmsweise auch einmal schief gehen, und eine Modifikation der Komposition provozieren. Die kooperative Aufgabe, für die ein Protokoll steht, kann auf einer höheren Ebene als Funktion, ggfs. mit Ausnahmen betrachtet werden. Usw.

Das bedeutet, dass das Softwareengineering beim Bau von Applikationen der Spannung ausgesetzt ist, einerseits Systeme über Funktionen zu beschreiben, aber andererseits diese Systeme nur durch nichtdeterministische, zustandsbehaftete Interaktionen in die gewünschten Interaktionsnetzwerke einbinden zu können. Ignoriert man diese Spannung und beschreibt interaktive Systeme tatsächlich mit einer explizit ausformulierten Systemfunktion, dann verliert man leider jede Garantie, dass kleine Änderungen in einzelnen Interaktionen auch nur kleine Änderungen in der Struktur der Applikation zur Folge haben wird.

Dies führt zu der naheliegenden Forderung durch eine "*interaktionsorientierte Architektur*" hier einen Ausgleich zu finden. Den Ansatz dazu liefert meiner Meinung nach die Inhomogenität der Komposition von Rollen zu Protokollen (3), also dass das Resultat der Komposition von Rollen selbst keine Rolle, sondern ein Protokoll ist. Deswegen müssen wir uns für die Komposition von (einfachen) Protokollen zu (komplexeren) Protokollen nach einer weiteren Kompositionsregel umschauen, die sich wiederum auf die Rollen beziehen muss. Wenn wir die Kompositionsrelation (3) als "*äußere*" Komposition ansehen, dann wäre die gesuchte Kompositionsregel zur Verknüpfung der Rollen eine "*innere*" Komposition im Sinne einer Rollenkoordination [24]. Der Aufbau einer Applikation unter Erhalt der Rollen hätte zur Folge, dass die Korrektheit der Implementierung einer Protokollrolle nicht von Änderungen in anderen Rollen beeinflusst wird.

Das Verständnis von Komposition in den Mittelpunkt der Debatte über die effiziente Herstellung von Interoperabilität zu stellen, impliziert unmittelbar die Aufforderung, die Protokolle wieder stärker in den Mittelpunkt der Vernetzung zu rücken [6]. Könnte die Informatik ein hinreichend einheitliches Verständnis für diese Form des Interfaces lose gekoppelter Applikationen erreichen, dann ließe sich vielleicht sogar der Trend des Internets der letzten 20 Jahre, sich von einer Welt der öffentlichen Protokolle in eine Welt der proprietären Plattformen zu transformieren, wieder rückgängig machen. Dazu könnte ebenfalls beitragen, wenn dem öffentlichen und multilateralen "Sprach"-Charakter der Protokolle folgend, der Gesetzgeber hier ordnungspolitisch die Offenlegung und freie Verwendung vorsieht.

Dank: Mein besonderer Dank geht an Arend Rensik, dessen Vortrag "Compositionality huh?" im Rahmen des Dagstuhl Workshop "Divide and Conquer: the

Quest for Compositional Design and Analysis" im Dezember 2012 mich letztlich zu diesem Denkansatz inspiriert hat.

Literatur

[1] NATO Standard Allied Data Publication ADatP-34, Edition M, Version 1 NATO Interoperability Standards and Profiles, 2020.

[2] R. Alur. *Principles of Cyber-Physical Systems.* MIT Press, 2015.

[3] C. Baier and J.-P. Katoen. *Principles of Model Checking.* The MIT Press, 2008.

[4] A. Basu, S. Bensalem, M. Bozga, P. Bourgos, and J. Sifakis. Rigorous system design: the bip approach. In *International Doctoral Workshop on Mathematical and Engineering Methods in Computer Science*, pages 1–19. Springer, 2011.

[5] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. Le Guernic, and R. De Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[6] Bitkom. *Vorschlag zur systematischen Klassifikation von Interaktionen in Industrie 4.0 Systemen – Hinführung zu einem Referenzmodell für semantische Interoperabilität.* White paper, 2020.

[7] *Leitbild 2030 für Industrie 4.0, Digitale Ökosysteme global gestalten.* Bundesministerium für Wirtschaft und Energie (BMWI), 2019.

[8] *Details of the Asset Administration Shell. Part 2 – Interoperability at Runtime – Exchanging Information via Application Programming Interfaces (Version 1.0RC01).* Bundesministerium für Wirtschaft und Energie (BMWI), 2020.

[9] J. D. Brock and W. B. Ackerman. Scenarios: A model of non-determinate computation. In *International Colloquium on the Formalization of Programming Concepts*, pages 252–259. Springer, 1981.

[10] M. Broy. A logical basis for component-oriented software and systems engineering. *Comput. J.*, 53(10):1758–1782, 2010.

[11] R. S. Chin and S. T. Chanson. Distributed, object-based programming systems. *ACM Computing Surveys (CSUR)*, 23(1):91–124, 1991.

[12] R. Cox. Surviving software dependencies. *Communications of the ACM*, 62(9):36–43, 2019.

[13] I. Crnkovic, S. Sentilles, A. Vulgarakis, and M. R. V. Chaudron. A classification framework for software component models. *Software Engineering, IEEE Transactions on*, 37(5):593–615, 2011.

[14] L. De Alfaro and T. A. Henzinger. Interface theories for component-based design. In *International Workshop on Embedded Software*, pages 148–165. Springer, 2001.

[15] DIN. SPEC 91345:2016-04 Referenzarchitekturmodell Industrie 4.0 (RAMI4.0), 2016.

[16] *Deutsche Normungsroadmap Industrie 4.0, Version 4.* DIN e.V. und DKE, 2020.

[17] R. Fielding. *Architectural Styles and the Design of Network-based Software Architectures.* PhD thesis, University of California, Irvine, 2000.

[18] D. Harel and A. Pnueli. On the development of reactive systems. In K. R. Apt, editor, *Logics and Models of Concurrent Systems*, pages 477–498. Springer-Verlag, New York, 1985.

[19] G. J. Holzmann. *Design and validation of computer protocols.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[20] S. Kleene. General recursive functions of natural numbers. *Mathematische Annalen*, 112(5):727–742, 1936.

[21] L. Lamport. Computer science and state machines. In *Concurrency, Compositionality, and Correctness*, pages 60–65. Springer, 2010.

[22] OASIS. Reference model for service oriented architecture 1.0. `http://docs.oasis-open.org/soa-rm/v1.0/`, 2006. Aufgerufen am 2015-01-17.

[23] J. Reich. The relation between protocols and games. In S. Fischer, E. Maehle, and R. Reischuk, editors, *39. Jahrestagung der GI, Lübeck*, volume 154 of *LNI*, pages 3453–3464. GI, 2009.

[24] J. Reich. Composition, cooperation, and coordination of computational systems. *preprint arXiv:1602.07065*, 2016/2020.

[25] J. Reich. A theory of interaction semantics. *preprint arXiv:2007.06258*, 2020.

[26] J. Reich and T. Schröder. A simple classification of discrete system interactions and some consequences for the solution of the interoperability puzzle. *ifac 2020*, 2020.

[27] S. Tripakis. Compositionality in the Science of System Design. *Proceeding of the IEEE*, 104:960 – 972, 2016.

[28] Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. `http://www.w3.org/TR/wsdl20/`, 2007. Aufgerufen am 2015-01-17.