Autonome Systeme und neue Paradigmen

Chemical Computing

02.07.2007

Übersicht

- Chemical Computing
  - Neues Paradigma: Strukturell aktiver Code
- Autonomic Computing und Verwandte
  - Aufbau von autonomen Systemen
- Selbst-Eigenschaften und Emergenz

Neue Vorbilder

- Bislang: bekannte Muster zur Erstellung von Anwendungen
  - Analyse der Problemstellung
  - Finden von Teilproblemen
  - Lösen dieser Teilprobleme durch bekannte Algorithmen
  - Zusammenfügen der Teillösung zu einer Gesamtlösung
- Problem
  - Es ist nicht klar, ob man alle Teilprobleme findet oder unzulässig verallgemeinert
  - Lösung funktioniert nicht wie erwartet

Brüchiger Code

- „Brittleness“ - Brüchigkeit
  - Der Code hat extrem geringen Redundanzgrad
  - Wenn es zu einem Fehler kommt (zur Erstellungs- oder Laufzeit), dann funktioniert die Lösung nicht mehr
  - Bisherige Anwendungsentwicklung ist sogar auf möglichst kleinen Redundanzgrad (ideal 0) ausgelegt.

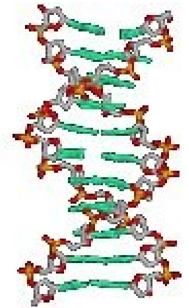
Einen auf Anhieb fehlerfreien Code zu schreiben ist ein Lotteriespiel.

Neue Inspiration

- Natürliche (biologische) Systeme
  - Gleichgewicht
  - Stabilität
  - sie funktionieren schlichtweg
- Natürliche Systeme bestehen aus autonomen Einheiten
  - und trotzdem (oder deswegen?) arbeiten sie alle zusammen

Biologically-inspired Computing

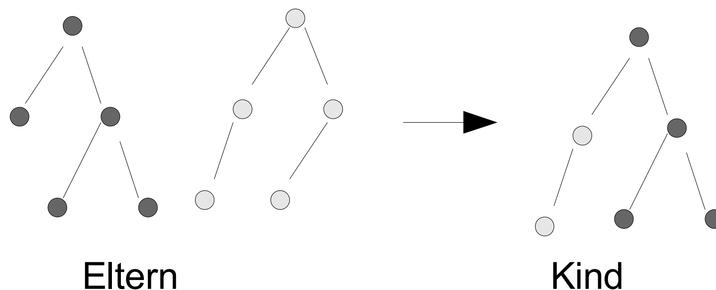
Beispiel: Genetische Programmierung



- Nachbildung der natürlichen Vorgänge der Evolution
- Evolvierende Programme
  - kein vorgegebener Code; stattdessen Maß, wie gut das Individuum in die aktuelle Umgebung passt („Fitness"-Funktion)
  - Basisoperationen
    - „Crossover": Codeteile zweier Programme werden ausgetauscht
    - „Reproduction": Code wird in die nächste Generation übertragen
    - „Mutation": Code wird an zufälligen Stellen geändert
    - „Architecture-altering operation": Änderung der Struktur (z.B. Erzeugung von Unterroutinen)

Genetische Programmierung

- Rekombination von „genetischem“ Material
  - Zahlreiche „nicht lebensfähige“ Individuen; sterben aus
  - Algorithmen setzen sich durch, welche das gesteckte Ziel gemäß der definierten Fitness besser erreichen
- Keine völlig willkürlichen Änderungen
  - Programme sollten zumindest kompilierbar/ausführbar bleiben
  - Einfach bei LISP-Programmen: Teilbäume umhängen



Genetische Programmierung

- Voraussetzung
  - Fitness muss eine Annäherung an das Optimum erkennbar machen
    - Eine 0-1-Fitness erlaubt diese Annäherung nicht
  - Beispiel: Fitness-Funktion, die angibt, an wie viele Knoten eine Information weitergegeben wurde
- GP zielt auf Programmentstehung ab
  - Höhere Abstraktionsschichten: Strategien entstehen lassen und vererben bzw. abwandeln
  - Vielleicht für Agenten einsetzbar?

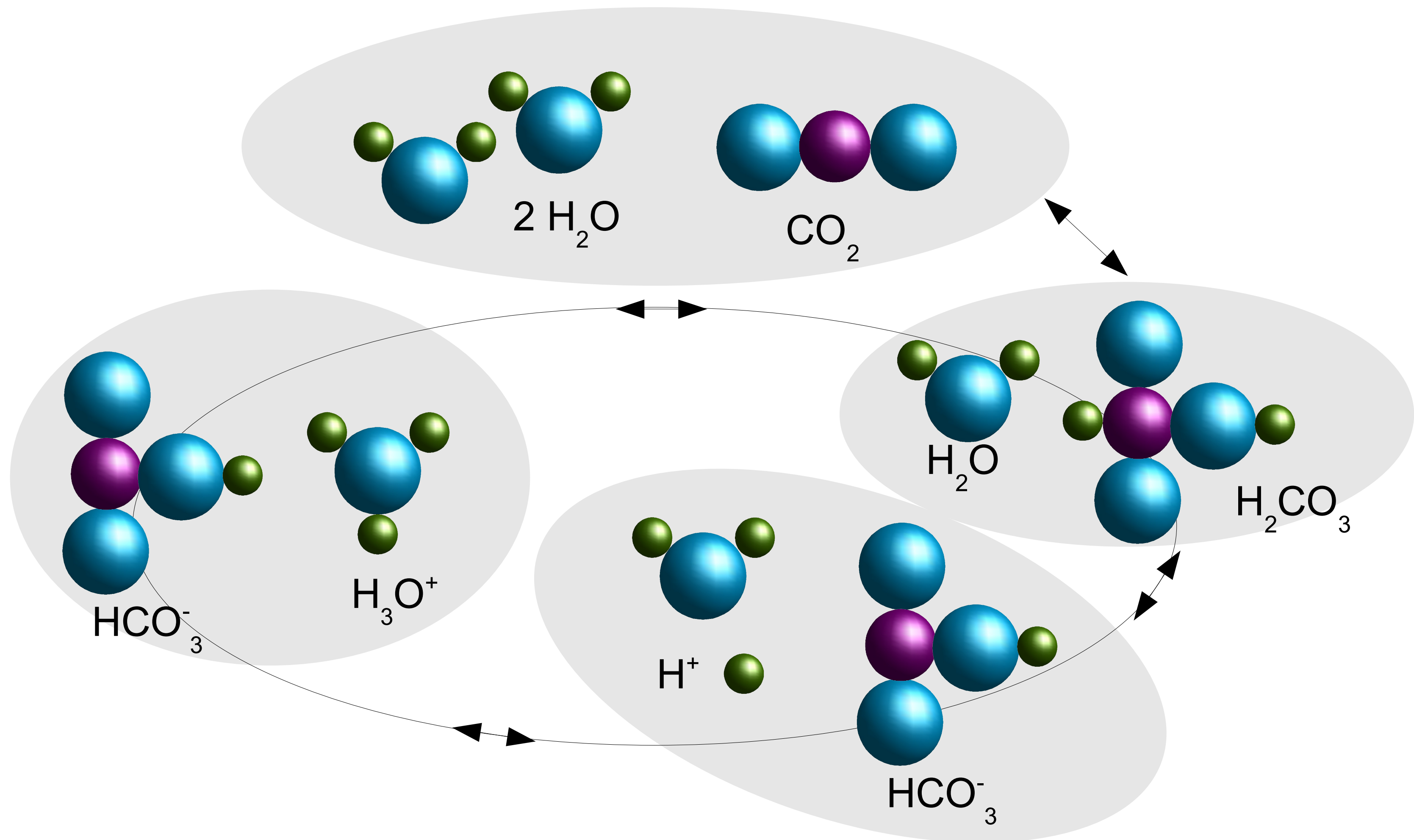
Chemical Computing

- Code, der miteinander „reagiert
- Christian Tschudin
  - bekannt von den „Messengers“
  - neues Konzept: „Fraglets“
    - Fragmente (von IP-Datagrammen)
    - Applet („kleine“ (-let) Applikationen, ausführbar)

Wieder ein neues Let?

- Fraglets als aktive Pakete
- Hintergrund
  - Chemische Verarbeitung (Chemical Computing)
  - Keine Trennung mehr von Code und Daten
  - Daten „reagieren“ miteinander

Chemische Verarbeitung



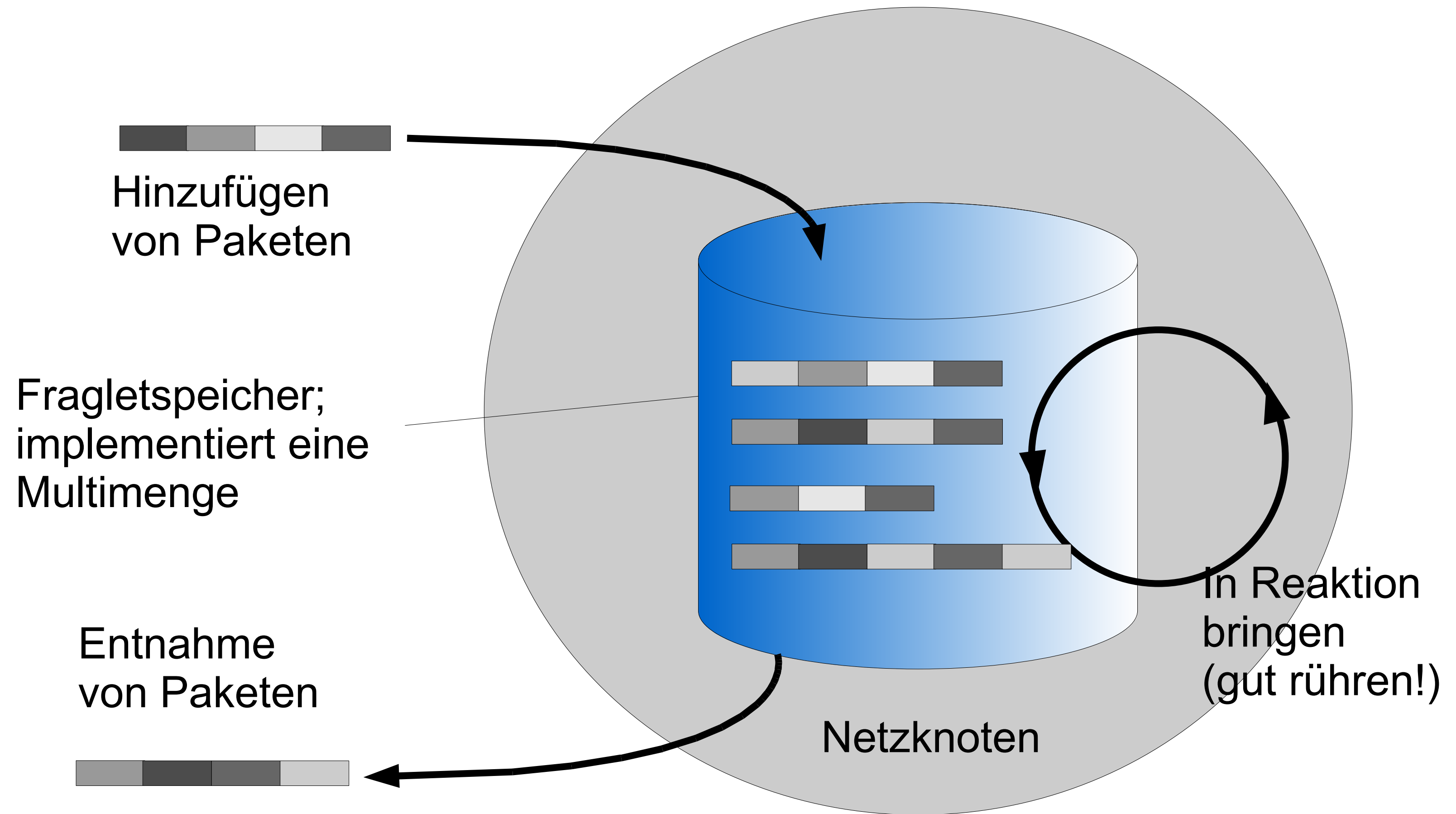
Chemische Verarbeitung

- Kein Anfang und kein Ende
- Gleichgewichtsreaktionen in beide Richtungen möglich
- Kein Aktor und kein Aktum
  - Nichts/niemand konstruiert Verbindungen
  - Jeder Bestandteil agiert auf anderen durch seine eigene Beschaffenheit, nicht aufgrund eines äußeren „Prozessors“

Fraglets und chemische Reaktion

- Fraglets konstruiert nach dem Vorbild chemischer Elemente und Reaktionen
- Nicht gedacht, um chemische Reaktionen zu simulieren
  - nur eingeschränkte Operationen
  - keine reversiblen Prozesse
- Synthese von Protokollen möglich
  - in Kombination mit genetischer Programmierung [4]
  - Perrig/Song: Protokollsynthese unter vorgegebenen Sicherheitsmerkmalen [3,5]

Das Fraglet-Modell



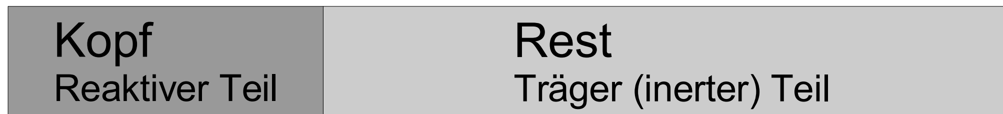
Fragletoperationen

- Zwei Klassen
  - eigene Umformung
  - Reaktion mit anderen Fraglets
- Prinzip
  - Stringübersetzungssystem: String Rewriting system
    - String-Matching
    - String-Ersetzung
  - Begrenzung auf einfache Matchingmuster
    - Aufwand gering halten

→ **Hardwarelösung** denkbar!

Fraglets: Operationen

Fraglet:



Notation: $f = [\ s_1 : s_2 : s_3 : \dots \text{Rest}\ ]$

s_i: Symbole

	Edukt	**Produkt**
nul	[**nul** : Rest]	- (Fraglet komplett tilgen)
dup	[**dup** : t : u : Rest]	[t : u : u : Rest]
exch	[**exch** : t : u : v : Rest]	[t : v : u : Rest]
new	[**new** : t : Rest]	[t : n_{i+1} : Rest]
split	[**split** : t : ... : * : Rest]	[t : ...] , [Rest]
send	$_A$[**send** : B : Rest]	$_B$[Rest]

n_{i+1} : neues, eindeutiges Symbol

wo das Fraglet gespeichert wird

Senden: nicht verlässlich

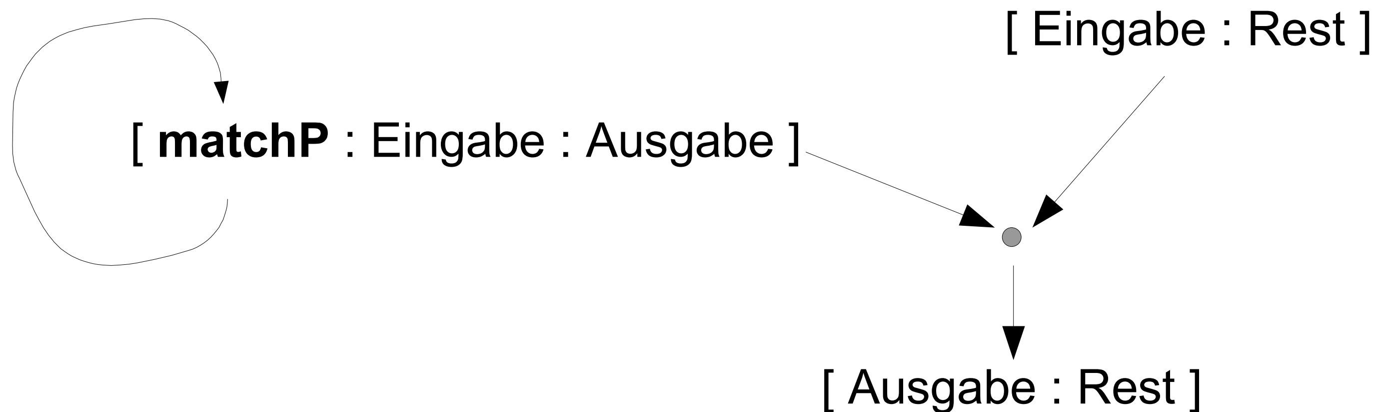
Fraglets: Reaktionen

	Edukte	**Produkte**
match	[**match** : s : $Rest_1$], [s : $Rest_2$]	[$Rest_1$: $Rest_2$]
matchP	[**matchP** : s : $Rest_1$], [s : $Rest_2$]	[**matchP** : s : $Rest_1$], [$Rest_1$: $Rest_2$]
matchS	[**matchS** : s : t : $Rest_1$], [s : t : $Rest_2$]	[$Rest_1$: $Rest_2$], [s : t : $Rest_2$]

matchP: Katalytische Reaktion

Einfache Reaktion

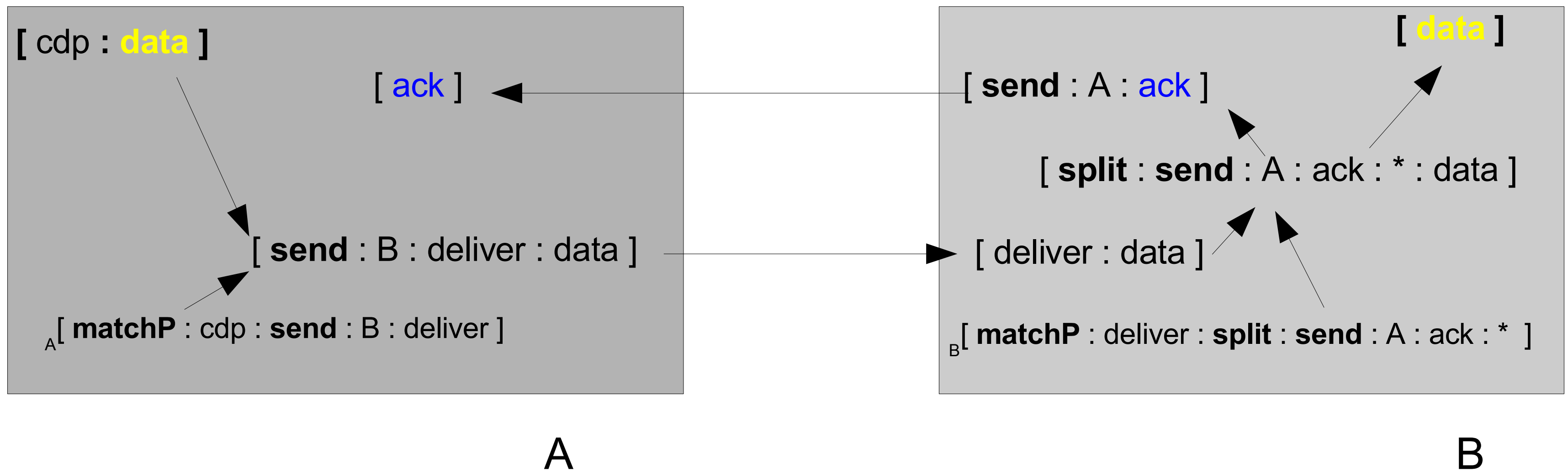
- Kopf ersetzen
  - Einfache matchP-Regel genügt



Einfaches Protokoll

- Confirmed Delivery Protocol

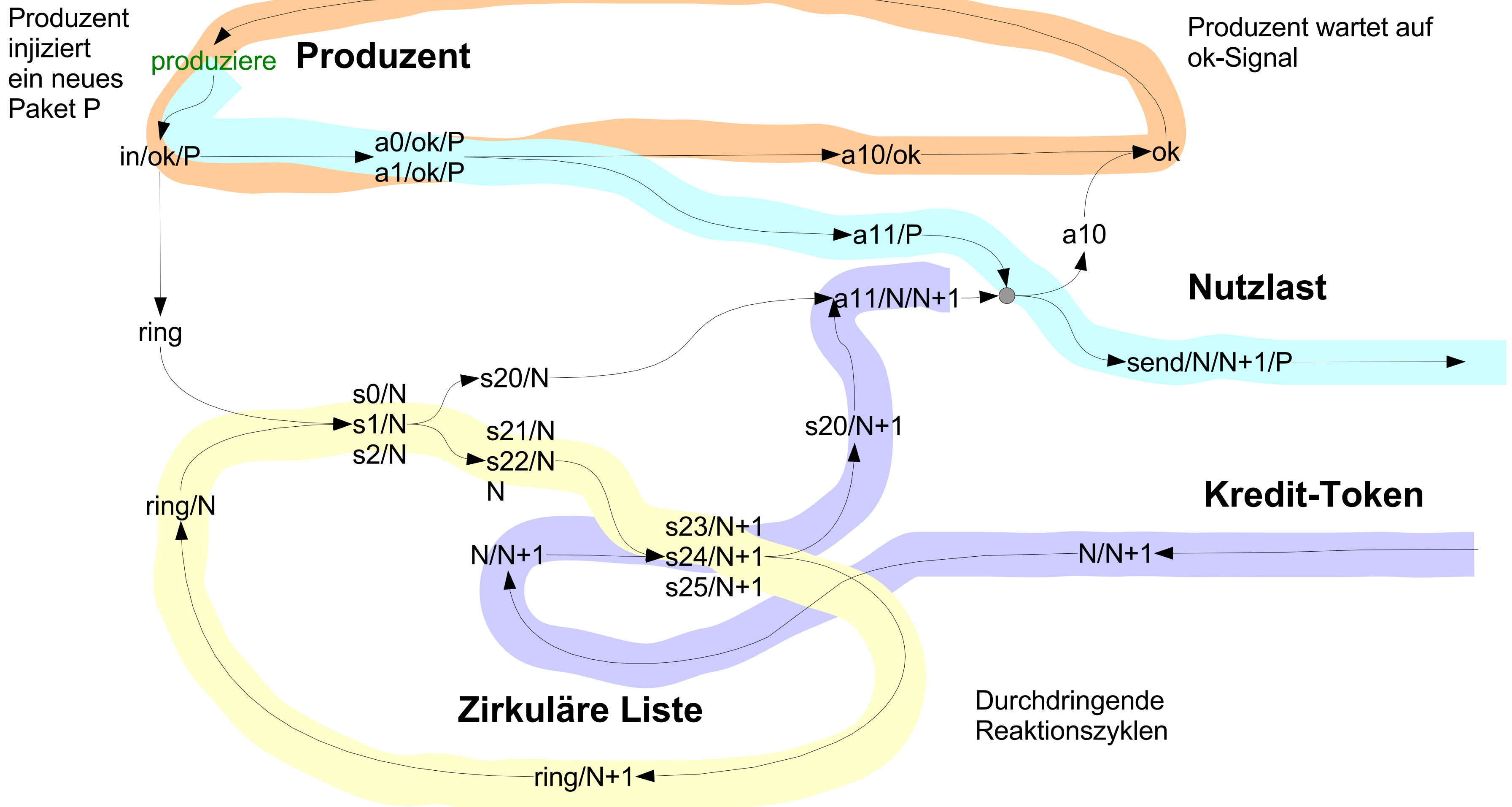
$_A$[**matchP** : cdp : **send** : B : deliver]
$_B$[**matchP** : deliver : **split** : **send** : A : ack : *]



Weitere Varianten

- Definition erforderte ein Fraglet am Ziel
  - Ohne Vorinstallation (vgl. Messenger) am Zielknoten:
    $_{A}$[**matchP** : cdp : **send** : B : **split** : **send** : A : ack : *]
- Auch komplexere Protokolle möglich
  - Kreditbasierte Flusssteuerung mit Paketordnung
  - Sender kann bis zu N Pakete versenden und wartet dann auf den ACK für das erste Paket
  - Mehrere Reaktionsschleifen
    - Produzentenschleife und Konsumentenschleife
    - Tokenschleife
  - 38 Fraglets involviert (20 beim Produzent, 18 beim Konsument)

Kredit-basierte Flusssteuerung



Produzent

Produzenten-Schleife: Erzeuge ein [in:...]-Fraglet
mit einem eindeutigen Namen als Nutzlast,
warte auf „ok“ vor dem erneuten Produzieren
$_A$[**matchP**: p_produce:**new**:p_request]
$_A$[**matchP**:p_request:in:ok]
$_A$[**matchP**:ok:**split**:p_produce:*:**nul**]
$_A$[p_produce]

ID-Ring; zurzeit 3 Kredite (ggf. vergrößern)
$_A$[ring:n0]
$_A$[n0:n1]
$_A$[n1:n2]
$_A$[n2:n0]

nimm „in“-Anfrage, speichere Parameter
(a10 callback, a11 Nutzlast)
$_A$[**matchP**:in:**split**:**match**:ring:**dup**:s0:*:**exch**:a0:a11]
$_A$[**matchP**:a0:**exch**:a1:*]
$_A$[**matchP**:a1:**split**:**match**:a10]
$_A$[**matchP**:s0:**exch**:s1:s21]
$_A$[**matchP**:s1:**exch**:s2:*]
$_A$[**matchP**:s2:**split**:s20]

Lasse den „Ring“-Zeiger fortschreiten;
Kopie des nächsten Wertes bei Marke „s26“
$_A$[**matchP**:s21:**matchS**:s22:**match**]
$_A$[s22:**dup**:s23]
$_A$[**matchP**:s23:**exch**:s24:s26]
$_A$[**matchP**:s24:**exch**:s25:*]
$_A$[**matchP**:s25:**split**:ring]

Weiterleitungslogik, zurzeit als passive Version.
Sende ein [n_i:n_{i+1}:Data]-Paket;
ein vorinstalliertes Fraglet beim Empfänger
nimmt es entgegen und macht weiter
$_A$[**matchP**:s20:**match**:s26:**match**:a11:**split**:a10:*:**send**:B]

Kredit-basierte Flusssteuerung

Konsument

Nächstes
erwartetes Token

N/N+1/P
r0/N/N+1/P
r1/N/N+1/P
r2/N/N+1/P
t0/N
r30/N+1/P
r31/N+1/P
r32/N+1/P
c0/P
out/ok/P
out
out
c1/ok/P
c2/ok/P
c3/ok/P
c4/ok
ok
t0
t1/N
r33/N+1
r34/N+1
r35/N+1
r36/N+1
r37/N+1
r39
N
t1/N+1
r38/N+1
r39/N+1
N+1
send/N/N+1

Konsument

Der „passive“ Empfänger wartet auf die
nächste erwartete Marke n_i. Dieses Empfänger-
Fraglet wird ständig neu erzeugt
$_B$[**match**:n0:r0:n0]

Isoliere Felder: n_i, n_{i+1} und Nutzlast
Rufe „out“-Lieferung auf
$_B$[**matchP**:r0:**exch**:r1:r30]
$_B$[**matchP**:r1:**exch**:r2:*]
$_B$[**matchP**:r2:**split**:t0]
$_B$[**matchP**:r30:**exch**:r30:c0]
$_B$[**matchP**:r31:**exch**:r32:*]
$_B$[**matchP**:r32:**split**:r33]
$_B$[**matchP**:c0:out:ok]

Erzeuge neuen Umordnungsstartcode
(passiver Startpunkt), speichern mittels r39
$_B$[**matchP**:r33:**dup**:r34]
$_B$[**matchP**:r34:**exch**:r35]
$_B$[**matchP**:r35:**exch**:r36:t1]
$_B$[**matchP**:r36:**exch**:r37:*]
$_B$[**matchP**:r37:**split**:**dup**:r38]
$_B$[**matchP**:r38:**exch**:r39:r0]

Konsumentenschleife; wartet auf ein Fraglet
der Form [out:handshakeTag:Nutzlast]
$_B$[**matchP**:out:**exch**:c1:nul]
$_B$[**matchP**:c1:**exch**:c2:*]
$_B$[**matchP**:c2:**split**]

Callback der Empfängerseite (Marke „ok“);
wird durch Konsumentenschleife aufgerufen. Damit:
- macht neuen Eintrag auf Empfängerseite frei (r39)
- sendet Kredit-Token zurück
$_B$[**matchP**:ok:**split**:**match**:r39:**match**:*:**match**:t0:
match:t1:**send**:A]

Funktionsweise

- Produzent generiert ständig neue Pakete
  - wartet jedoch auf „ok“-Token, bevor ein neues Paket abgeschickt wird
- Verwendung dreier Token (legt Reihenfolge fest)
  - Ring kann entsprechend vergrößert werden
- Nutzlast wird mit dem neuen Token an die Gegenseite geleitet
- Spezieller Kanal auf Empfängerseite sortiert die Pakete mittels des erwarteten Tokennames
  - Nutzt dazu ein wartendes Fraglet mit dem erwarteten Namen
  - Dann spaltet sich der Ablauf:
    - Nutzlast geht zum Konsumenten, der „ok“ signalisieren muss
    - Zweiter Pfad wartet auf „ok“ und schickt dann das Token zurück

Automatische Fraglet-Synthese

- Einsatz der genetischen Algorithmen
- Vorzüge der Fraglet-Umgebung
  - Fraglets sind rein stringbasiert, fast keine syntaktischen Bedingungen
  - Fraglets sind einfach und kurz gehalten
- Fortentwicklung durch Mixen von Fraglets
- Derzeitiger Ansatz
  - Zufallsmenge von Fraglets in einen Topf werfen
  - Fitnessfunktion berücksichtigt z.B. die Anzahl korrekt bestätigter Pakete oder Anzahl der Verarbeitungsschritte

Selbstheilende Protokolle?

Ansatz über Fraglets

- Grundidee
  - Fraglets haben keine einzelne Ausführungsentität – die Daten selbst reagieren
  - Redundanz lässt sich leicht erzeugen
  - System kann sich zwischen Redundanz und Effizienz flexibel bewegen

Redundanz | Effizienz

Redundante Systeme können genutzt werden und liegen nicht brach, sie rechnen stets parallel mit.

Protokollrobustheit

- Bislang
  – Robustheit in Bezug auf das Erkennen von Fehlern, die vom Transport verursacht wurden (Abbrüche, verlorene Pakete usw.)
- Jetzt
  – Fähigkeit, Operationsfehler zu „überleben“
  – Einbeziehen der Protokollverarbeitung
  – Im Idealfall sogar Erholung von Fehlerzuständen (Selbstheilung!)

Beispiel: Der Nachrichtenverdoppler

- Jede Nachricht x soll genau zwei Nachrichten z bewirken
- Ansatz
  - [**matchP** : x : z] schreibt x in z um, verdoppelt aber nicht
  - [**matchP** : x : **split** : z : * : z] erzeugt tatsächlich die zwei Fraglets
  - Aber nicht robust!
    - Ist das Fraglet weg, wird nichts mehr verdoppelt.

Erster Ansatz

- Fraglet verdoppeln
  - [**matchP** : x : **split** : z : * : z]
    [**matchP** : x : **split** : z : * : z]
  - Wird eine Regel verloren, so wird die Ersatzregel doppelt so häufig aufgerufen
  - Aber kein Hinweis auf einen Problemfall → keine Korrekturaktionen
  - schlimmstenfalls geht das zweite Fraglet auch noch verloren

Neuer Ansatz

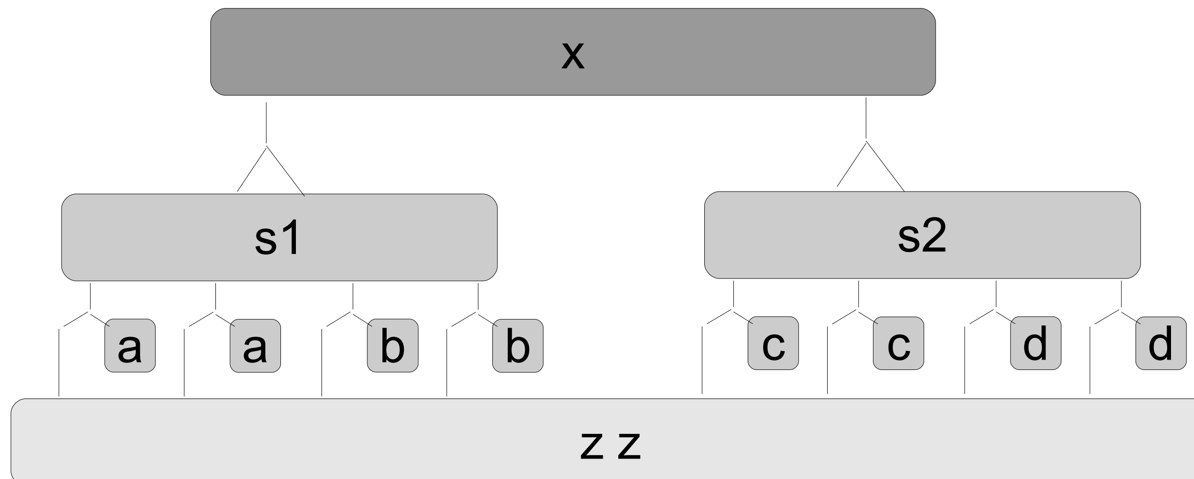
[**matchP** : x : **split** : s1 : * : s1]
[**matchP** : x : **split** : s2 : * : s2]
[**matchP** : s1 : **split** : z : * : a]
[**matchP** : s1 : **split** : z : * : b]
[**matchP** : s2 : **split** : z : * : c]
[**matchP** : s2 : **split** : z : * : d]

} Wahrscheinlichkeit 50% für die Auswahl einer Regel

- Die Hälfte aller x-Fraglets wird zu s1-Fraglets, die andere Hälfte zu s2-Fraglets
  - diese jeweils verdoppelt, in Summa also doppelt so viele z wie x
  - Es gibt zu jedem z ein korrespondierendes a, b, c oder d
- Test: Was passiert, wenn eine Regel stirbt?

Robustes Protokoll

- Hintergrund: „Suppenaspekt“ der Verarbeitung
  - Konkurrierende Regeln (Agenten)
  - Fällt eine Regel aus, übernimmt die konkurrierende Regel die Arbeit
  - ähnlich wie eine chemische Reaktion



Selbstheilung

- Hinzufügen von zwei weiteren Fraglets
  - [matchP : a : match : c]
    [matchP : b : match : d]
- Man beobachtet nun die relativen Häufigkeiten der Fraglets
  - wie ein Lackmustest: Säuregrad der Lösung bestimmen
  - daraus schlussfolgern, welche Regel kaputt ist
  - eigentliche Funktion (x→zz) bleibt unberührt
  - Automatische Ergänzung von [matchP]-Regeln erwies sich als sehr kompliziert*

*Man kann durch externe Mechanismen bewirken, dass die fehlenden Regeln ergänzt werden;
eine automatische Ergänzung durch andere Fraglet-Regeln scheint jedoch schwierig [2].

Verteilung des Protokolls

${}_p$[**matchP** : a : **match** : c]

${}_p$[**matchP** : b : **match** : d]

${}_o$[**matchP** : s2 : **split** : **send** : p : z : * : **send** : p : c]

${}_o$[**matchP** : s2 : **split** : **send** : p : z : * : **send** : p : d]

${}_m$[**matchP** : s1 : **split** : **send** : p : z : * : **send** : p : a]

${}_m$[**matchP** : s1 : **split** : **send** : p : z : * : **send** : p : b]

${}_n$[**matchP** : x : **send** : m : **split** : s1 : * : s1]

${}_n$[**matchP** : x : **send** : o : **split** : s2 : * : s2]

Auch nützlich für die Erkennung des Absturzes eines Knotens

Überzählige Regeln

- Im Szenario eines vollständig selbstverwaltenden Systems müssen auch überzählige Regeln erkannt und entfernt werden

  → Keine [**matchP**]-Regeln mehr, sondern ausschließlich [**match**]
  - vollständig dynamisches Programm, das sich permanent selbst umschreibt
  - dann aber Steuerung erforderlich, die das Programm in seinen Bahnen hält
  - vergleichbar mit Biologie: Transskriptionssignale erforderlich, welche die Genexpression steuern

Ein neuer Weg?

- Chemische Verarbeitung liefert neues Prinzip
  - Kein Programm mehr, das Daten verarbeitet
  - sondern Daten, die miteinander reagieren
- Flexible Selbstjustierung des Systems zwischen Redundanz und Effizienz
  - Selbstheilung möglich
  - Evolution möglich
- Bisher aber wieder nur simple Protokolle

Literatur

[1] Tschudin: Fraglets – A Metabolistic Execution Model for Communication Protocols. Proc. 2nd Annual Symposium on Autonomous Intelligent Networks and Systems AINS, 2003

[2] Tschudin, Yamamoto: A Metabolistic Approach to Protocol Resilience. Technischer Report CS-2004-003, Universität Basel, 2004

[3] Perrig, Song: A First Step towards the Automatic Generation of Security Protocols. Proc. Network and distributed systems symposium NDSS, Feb. 2000

[4] Sharples, Wakeman: Protocol construction using genetic search techniques. In Real-World Applications of Evolutionary Computing, EvoWorkshops 2000, LNCS 1803

[5] Song, Perrig, Phan: AGVI – Automatic Generation, Verification, and Implementation of Security Protocols. Proc. 13th conference on Computer-Aided Verification CAV 2001