ISA
Institut für
SoftwareArchitektur

THM
TECHNISCHE HOCHSCHULE MITTELHESSEN

Thomas Letschert

Verteilte Berechnung und Simulation
Eine kurze Übersicht

15. März 2012

Technischer Bericht Nr. 3
des Instituts für SoftwareArchitektur

1 Verteilte Berechnungen

Nebenläufigkeit und Verteilung sind das Mittel der Wahl, wenn aufwändige Algorithmen auszuführen sind. Bei diesen Algorithmen handelt es sich ganz allgemein um Berechnungen die aber oft auch auf einen ganz bestimmten Aufgabenbereich beschränkt sein können. So kann kann beispielsweise die Anwendung eine *Data–Mining*–Applikation sein, bei der eine große Menge an statischen Daten zu analysieren ist, oder eine *Data–Streaming*–Applikation, bei der auf kontinuierlich einströmende Daten in Realzeit zu regieren ist. Eine andere Klasse von Ressourcen–intensiven Anwendungen stellen Simulationsaufgaben dar.

Verteilte und nebenläufige Berechnungen werden in einer bestimmten Notation dargelegt und dann ausgeführt. Bei der Notation kann es sich um Programme eine Programmiersprache handeln. In der Regel verwendet man jedoch im Bereich der verteilten Anwendungen jedoch keine "geschlossene" Programmiersprache, sondern ein Konglomerat aus Programmen eventuell unterschiedlicher Programmiersprachen, Konfigurationsdateien und anderen Artefakten. Beispielhaft sei das JEE–Framework genannt.

Die Inbetriebnahme (*Deployment*) und die kontinuierliche Pflege einer Anwendung mit einer solchen komplexen Struktur kann durch ein passendes Laufzeitsystem (*Runtime, Container*) unterstützt werden. Ein *Framework* kann mit Standardkomponenten die Entwicklung erleichtern. *Runtime* und *Framework* sind dabei immer aufeinander abgestimmt. Die Abstimmung ist notwendig und so eng, dass *Framework* und die dazu passende *Runtime*–Implementierung oft identifiziert werden. Auch wir werden in der Regel nur noch von *Framework* reden, auch wenn beides gemeint ist.

Die Erstellung einer komplexen verteilten Anwendung sollte wenn möglich von einen *Framework* unterstützt werden. Viele Anforderungen an eine verteilte Anwendung sind komplex aber nicht spezifisch für eine bestimmte Anwendung. Die entsprechenden Lösungen sollten darum wiederverwendet werden.

Bei der Auswahl eines Frameworks muss die Anwendungsdomäne und ihre typischen Anforderungen in Bezug gesetzt werden zu den Mitteln des Frameworks. Bei den bereit gestellten Mitteln handelt es sich in erster Linie um vordefinierte Lösungen für Standardaufgaben. (Kommunikation, Transaktionen, Persistenz etc.) Bevor aber derartige "algorithmische Ressourcen" näher untersucht werden, müssen fundamentalere Eigenschaften eines Frameworks näher untersucht werden: Das – meist nur implizit formulierte – Modell der Verteiltheit, das den vom Framework unterstützten Berechnung zugrunde liegt und das – meist explizit formulierte – Modell des Lebenszyklus' einer Anwendung und eventuell ihrer Komponenten.

2 Verteilte Berechnungen: Modelle und Notationen

2.1 Modelle

Bekanntlich (siehe z.B. [1]) sind in verteilten Systemen, im Gegensatz zu sequentiellen Systemen, Berechnungsmodelle von essentieller Bedeutung. So ist es ein fundamentales Ergebnis der theoretischen Informatik, dass die unterschiedlichen Berechnungsmodelle sequentieller Systeme (Turing Maschinen, Lambda–Kalkül, rekursive Funktionen) äquivalent sind. In einem verteilten System gilt nichts Entsprechendes. Ein Algorithmus kann dort nur in Bezug auf ein Berechnungsmodell bewertet werden.

Die Berechnungsmodelle stecken in der Spezifikation der eingesetzten Sprachen, in deren Implementierung und den verwendeten algorithmischen Standardlösungen. Einige Aspekte der Modelle sind explizit dargelegt und/oder per se klar. So ist eine Kommunikation via TCP offensichtlich synchron und eine via RPC oder RMI offensichtlich asynchron. Andere Dinge wie die Fehlersemantik sind weniger offensichtlich oder, wie etwa die Topologie der Verteilung, gänzlich offen und außerhalb der Sprachen und Frameworks.

Wir wollen hier nicht auf die reichhaltige Literatur der formalen Modelle verteilter Systeme eingehen, sondern nur die wichtigsten Charakteristika der verschiedenen Varianten kommunizierender sequentieller Prozesse eingehen, die als Basis von Implementierungen verteilter Berechnungen von praktischer Bedeutung sind. Die formale Basis sind stets globale Transitionssysteme die sich als, durch lokale prozessspezifische Transitionssysteme induziert, definieren lassen.

2.2 Kommunizierende Prozesse

Basis vieler Modelle ist das Konzept der kommunizierenden Prozesse. Ein verteiltes System besteht aus einer Menge von Prozessen, die in strikt sequentieller Manier Berechnungen ausführen und sich dabei gegenseitig Nachrichten zusenden. Unterschiede gibt es dabei in folgenden Punkten:

- Statik / Dynamik der Topologie.
- Lebenszyklus der Prozess–Bekanntschaften.
- Synchronität der Kommunikation
- Natur der Nachrichten
- Reaktive, oder ereignisorientierte Prozess–Spezifikation.

Topologie Bei einer statischen Topologie sind die Bekanntschaften der Prozesse untereinander zu Beginn der Berechnung fest und können sich im Laufe der Berechnung nicht mehr ändern. Eine Berechnung mit einer dynamischen Topologie kann die Verbindung zwischen den Prozessen ändern und eventuell neue Prozesse generieren. In besonders restriktiven Modellen sind die Prozess-Graphen (Prozesse und ihre Bekanntschaften) gerichtet und erlauben keine Zyklen.

Lebenszyklus der Prozess–Bekanntschaften Auch bei einer fixen Topologie ist es unhandlich im Code eines Prozesses mit festen Bezügen zu anderen Prozessen zu arbeiten. Wiederverwendbare Prozessdefinitionen können mit einem "Port"–Konzept erreicht werden. Man definiert Prozesse relativ zu fiktiven Endpunkten von Kommunikationskanälen, den Ports, und verbindet dann bei einer "Netzdefinition" diese Ports mit "Kommunikationskanälen". Der Sinn liegt darin, dass die Definition der Topologie und die der einzelnen Prozesse entkoppelt wird. Alternativ dazu kann man die Prozessbekanntschaften komplett dynamisch handhaben. der Unterschied liegt im Zeitpunkt der "Prozessverknüpfung":

- statisch im Code der Prozesse,
- zur "Bindezeit" via Ports,
- dynamisch zur Laufzeit.

Selbstverständlich macht das Port–Konzept wenig Sinn bei dynamischen Prozess–Bekanntschaften, d.h. bei Topologien die sich zur Laufzeit ändern können. Ein spezielles Port–Konzept ist dann unnötig. (Es sei denn man nennt eine Variable mit einer Prozessbekanntschaft "Port".)

Synchronität der Kommunikation Nachrichten können synchron oder asynchron übertragen werden. In den letzten Jahren ist die synchrone Kommunikation (RPC, RMI) etwas aus der Mode gekommen, in der Praxis spielt es immer noch eine wichtige Rolle. (RMI in JEE)

Natur der Nachrichten Nachrichten sind in der Regel einfache Werte. Manche Systeme erlauben es, veränderliche Objekte und Referenzen zu versenden. In den meisten Modellen sind die Prozesse von Werten wohl unterschieden: Sie können nicht als Nachrichten versendet werden. In manchen Modellen sind Prozesse Werte, die wie alle anderen Werte versendet werden können. Natürlich implizieren Prozesse als *first class vlaues* verteilte Systeme mit dynamischen Toplolgien.

Reaktive oder kontrollorientierte Prozess–Spezifikation Prozesse können reaktiv oder kontrollorientiert spezifiziert werden. Bei einer reaktiven (ereignisorientierten) Spezifikation werden Prozesse

durch ihre Reaktionen auf externe Ereignisse definiert. Typischerweise sind die externen Ereignisse der Empfang bestimmter Nachrichten, oder Nachrichten die einem bestimmten Muster entsprechen. Bei einer kontrollorientierten Spezifikation wird der Prozess als sequentielles Programm mit eingebetteten Sende– und Empfangsoperationen definiert.

2.3 Datenfluss–Systeme

2.4 Visuelle Datenfluss–Programmierung

Unter einem Datenfluss–System versteht man meist ein Prozess–Netz mit statischer Topologie in denen nur (unveränderliche) Werte versendet werden können. Die Verbindungen sind unidirektional und stellen Puffer mit (meist) begrenzter Kapazität dar. Die Ausdrucksmittel der entsprechenden Formalismen erlauben es oft, dass Teilnetze zu Knoten aggregiert werden.

Derartige Systeme haben eine relativ einfache Semantik und können leicht graphisch dargestellt werden. Sie sind darum sehr erfolgreich als "visuelle Programmiersprachen" für komplexe Anwendungen eingesetzt worden, meist im Bereich des Data–Minings, wo Informatik–Laien die üblichen Anwender sind.

Die Knoten sind sequentielle Programme oder Prozesse die in einer klassischen Programmiersprache oder einer Skriptsprache, in der Regel vorgefertigt, vorliegen und für spezielle Anwendungen mit Hilfe einer Ablaufumgebung zu Netzen zusammen gefügt werden. Die Definition der Knoten erfolgt mit Hilfe eines auf die Ablaufumgebung abgestimmten Frameworks. Die Ablaufumgebung glänzt üblicherweise noch mit gefälligen Visualisierungen der Ergebnisse. Prominentes Beispiel für ein derartige Systeme ist Knime ([3], http://www.knime.org/).

Die Semantik dieser Systeme wird meist nicht explizit formuliert sondern ergibt sich aus der Implementierung: Prozesse werden kontrollorientiert definiert, die Topologie ist statisch und meist frei von Zyklen. Die Wiederverwendbarkeit der Knotendefinitionen und die Statik der Netze impliziert ein Port-Konzept. Nachrichten sind reine Werte. Das Versenden von Referenzen oder gar von Prozessen ist nicht möglich. Die Beschränkung der Ausdruckskraft erlaubt insgesamt gefällige visuelle Darstellungen der Programme und vermeidet komplexe Probleme wie Verklemmungen.

Eine Sprache, mit der die Datenfluss–Programme spezifiziert werden, hat typischerweise zwei getrennte Ebenen: eine zur Definition der Knoten und eine zur Definition der Netze. In visuellen Systeme wird oft gänzlich auf die präzise Formulierung einer einer "Netzsprache" verzichtet und die "Knotensprache" ist eine anderweitig definierte Programmiersprache. Die Spezifikation eines solchen Systems kann sich darum auf die exakte Beschreibung der Interaktion der Knoten beschränken.

2.5 FBP: Flow Based Programmierung

Die visuelle Programmierung hat ihre Vorteile, aber auch ihre Beschränkungen. Selbst wenn die Netze statisch sind und damit prinzipiell eine graphische Darstellung möglich ist, bevorzugen professionelle Programmierer oft eine textuelle Darstellung. Für Datenfluss–Programme mit statischer Topologie und textueller Darstellung wird die oft die Bezeichnung *Flow Based Programming* (kurz FBP) verwendet. Der Begriff wurde von J. Paul Morrison eingeführt und in seinem Buch [4] ausführlich erläutert. Morrison entwickelte ein Framework für sein FBP, das auch in einer Java–Variante als Open–Source–Projekt bis heute gepflegt wird.

Zur Formulierung der Knoten– und der Netzsprache wurden verschiedene Ansätze untersucht. In [5] werden Knoten in Java als Ableitung einer Basisklasse *Component*, aus dem FPB–Framework, definiert in denen Ports über Annotationen bekannt geben werden. Netze werden in XML definiert und die Ablaufumgebung erzeugt lauffähige Netze aus den annotierten Knotendefinitionen und den XML–Deskriptoren.

In [6] wird eine überarbeite Version einer XML–basierten "Netzsprache" vorgestellt. Als "Knotensprache" kommen Java und Groovy in Frage. Für Knoten und Netze wird ein explizites Modell des Lebenszyklus' definiert. Die Semantik der Knotenverbindungen wird ebenfalls exakt definiert. Ports werden wieder durch Annotationen im Programmtext der Knoten angegeben, sie sind aber jetzt typisiert. Damit ist eine Typ–Prüfung auf der Ebene der Netzsprache möglich. Die Praktikabilität der Netze wird durch ein "Beobachterkonzept" für Knoten erweitert. Für jeden Knoten (–Typ) können Abfragen zum aktuellen Zustand formuliert werden, die dann zur Laufzeit vom Framework / der Ablaufumgebung ausgeführt werden.

In [7] wird das FBP–Framework weiter entwickelt. Die Trennung zwischen Netz– und Knotensprache wird hier aufgehoben. Sowohl Knoten als auch Netze werden werden in Java definiert. Die Netze in Form einer internen DSL (DSL = *Domain Specific Language*). Eine interne DSL ist eine API die so konzipiert ist, dass sie sich anfühlt wie eine Sprache, die für einen speziellen Anwendungszweck konzipiert wurde (vergl. [8]). Java ist als DSL–Sprache nur begrenzt geeignet. Eine Umstellung auf das geeignetere Groovy oder Scala ist leicht möglich.

Der Gedanke, eine eigenständige Programmiersprache mit allem was dazu gehört, Scanner, Parser, Codegenierung etc. für FBP zu definieren, liegt nahe, aber üblicherweise wird der damit verbundene Aufwand gescheut. In [9] wurde erfolgreich mit einer Knoten– Netzsprache als vollwertige compilierte Sprache mit Integration in eine Entwicklungsumgebung experimentiert.

Insgesamt lässt sich konstatieren, dass sich mit einfachen datenflussorientierten Systemen ansprechende Systeme für Computer–Laien entwickeln lassen. Die Beschränkung der Ausdrucksmittel und die weitgehende Verwendung vorgefertigter Knoten machen die Systeme einfach bedienbar und reduzieren die Gefahr von Nebenläufigkeitsproblemen. Die Herausforderung liegt in der Bedienbarkeit der Systeme und der Fähigkeit Berechnungsergebnisse in ansprechenden Reports aufzubereiten. Ent-

sprechende Systeme sind verfügbar.

Die graphische Programmierung ist für professionelle Anwender weniger geeignet. Eine textuelle Programmierung erhöht die Produktivität der Entwickler erheblich. Die genaue Form der Netz– und Knotensprache scheint nicht sehr relevant zu sein. Die Erfahrung mit verschiedenen Varianten spricht dafür, als "Knotensprache" eine oder eventuell mehrere konventionelle Programmiersprachen zu verwenden und als Netzsprache eine speziell zugeschnittene Notation, also eine DSL, zu verwenden. XML ist dabei wenig empfehlenswert, die Frage, ob die DSL intern oder extern ist, ist wenig relevant. Von entscheidender Bedeutung ist die Entwicklungsumgebung und die Möglichkeit einer bequemen Aufbereitung der Ergebnisdaten.

Textuelle Datenfluss–Programmierung im Sinne des FBP ist unüblich. Nicht weil Datenfluss–Programmierung grundsätzlich nur graphisch ist, sondern weil textuelle Systeme üblicherweise eine größere Ausdruckskraft erlauben als die visuellen Varianten oder das FBP. Rekursion, dynamische Netze sind nur in visuellen Notationen ein Problem. Textuelle Varianten wie FBP sind darum unnötig beschränkt und werden kaum verwendet. Ihre beschränkte Ausdruckskraft hat natürlich praktische Vorteile.

2.6 Push– oder Pull–Netze

Die Semantik statischer Prozessnetze erscheint auf den ersten Blick einfach und offensichtlich zu sein. Bei genauerem Hinsehen zeigen sich aber noch einige interessante Spielräume bei der Festlegung eines Modells. Das erste explizit formulierte formale Modell eines Datenfluss–Systems ist das von Kahn [10]. Ein Prozess in einem Kahn–Netz ist eine Funktion die einen oder mehrere (unendliche) Datenströme als Eingabe hat und einen Datenstrom als Ausgabe. "Kahn-Netzwerke" bieten allerdings funktionale und rekursive Ausdrucksmittel, die alle Aspekte eines Algorithmus abbilden können. Ihre Ausdruckskraft ist darum wesentlich größer als die des FBP oder der visuellen Datenfluss–Systeme.

Die Prozesse (Funktionen auf Datenströme) sind dabei stets "kontinuierlich". Einfach ausgedrückt bedeutet dies, dass neu eintreffende Werte auf den Eingabeströmen keine bereits berechneten Ausgabewerte ungültig machen dürfen und dass jeder Präfix des Ausgabestroms von Präfixen endlicher Länge der Eingabeströme abhängt. Man wartet also nie unendlich lange auf die "ersten Ausgaben" eines Prozesses.

Die Prozesse sind durch unidirektionale FIFO–Kanäle von unendlicher(!) Kapazität verbunden. Eine Empfangsoperation kann darum blockieren, eine Sendeoperation blockiert nie. In [11] wird gezeigt, dass ein solches Netz mit *Nachfrage–getrieben* Multitasking, als "Pull–Netz" implementiert werden kann. Jeder Knoten holt sich die Daten die er braucht und wenn er sie braucht.

In einem Push–Netz werden Daten sozusagen ohne Rücksicht auf den Nachfolger produziert. In dem Fall kommen Kanäle mit unendlicher Kapazität natürlich nicht in Frage. Die beschränkte Kapazität wird benötigt um Produzenten– und Konsumenten–Prozesse zu synchronisieren.

Pull–Netze sind von theoretischem Gesichtspunkt attraktiv, spielen aber praktisch keine Rolle. Üblicherweise werden die Knoten als aktive Prozesse / Threads realisiert die sich über Kanäle beschränkter Kapazität synchronisieren.

2.7 Leichte, reaktive und schwere Prozesse

In massiv parallelen Berechnungen kann es durchaus vorkommen, dass die Zahl der logischen Prozesse die Zahl der real zur Verfügung stehenden Threads oder Prozesse um ein Vielfaches übersteigt. Mehrere tausend logische Prozesse sind nicht unrealistisch, können aber nicht auf tausende von System–Threads oder Prozesse abgebildet werden, ohne dass auch ein modernes System zum sofortigen Zusammenbruch käme, es sei denn eine exotische Spezialhardware steht zur Verfügung. Es ist also wünschenswert die logischen von den realen Prozessen zu entkoppeln.

Eine Entkopplung von logischen und realen Prozessen ist nur möglich, wenn die logischen Prozesse ohne Stack auskommen. Solche Prozesse werden oft *rein reaktiv* genannt. Sie sind dadurch charakterisiert, dass es keine blockierenden Operationen gibt. Der Benutzer ist leicht zu ermuntern rein reaktive Prozesse zu spezifizieren, wenn er eine *reaktive Notation* verwenden muss. In der Essenz bedeutet die Verwendung einer reaktiven Notation, dass die Prozesse jeden Blockierungspunkt in einen Zustand umwandeln müssen. Ein einfaches Beispiel in Pseudocode mag dies illustrieren:

```
Prozess Adder {
   PortIn  p1: Integer
   PortIn  p2: Integer
   PortOut p3: Integer
   while (true) {
     var v1 = p1.read()
     var v2 = p2.read()
     var v3 = v1 +v2
     p3.send(v3)
   }
}
```

Hier handelt es sich um eine kontrollorientierte Notation in der ein Prozess spezifiziert wird, der zwei blockierende Operationen enthält: der Empfang an den beiden Ports. Eine Unterbrechung an den beiden Punkten speichert den dann aktuellen Zustand als aktuellen Zustand des Stacks. Ein Thread mit Stack wird benötigt.

Formuliert man das Gleiche ereignisorientiert dann ergibt sich:

```
Prozess Adder {
   PortIn  p1: Integer
```

```
    PortIn  p2: Integer
    PortOut p3: Integer
    var state = 00
    var v1
    var v2

    Ready(p1) =>
         case (state) {
          00 : v1 = p1.receive(); state = 10
          01 : v1 = p1.receive(); p3.send (v1+v2); state = 00
         }
       }

    Ready(p2) =>
         case (state) {
          00 : v2 = p2.receive(); state = 01
          10 : v2 = p2.receive(); p3.send (v1+v2); state = 00
         }
       }
}
```

Die beiden Werte werden angenommen, dabei wird jeweils der Zustand gewechselt. Interessant ist, dass es möglich sein muss Ereignisse zu ignorieren. Ein Operand wird wird erst angenommen, wenn sein Platz frei ist. Natürlich kann das Ganz auch zustandsbetont dargestellt werden:

```
Prozess Adder {
    PortIn  p1: Integer
    PortIn  p2: Integer
    PortOut p3: Integer
    var state = 00
    var v1
    var v2

    case (state) {
       00: case Ready(p1) =>  v1 = p1.receive(); state = 10
           case Ready(p2) =>  v1 = p2.receive(); state = 01
       10: case Ready(p2) =>  v1 = p2.receive(); p3.send (v1+v2); state = 00
       01: case Ready(p1) =>  v1 = p1.receive(); p3.send (v1+v2); state = 00
    }
}
```

Die Spezifikation mag für Programmierer ungewohnt sein, zwingt aber dazu rein reaktive Prozesse zu spezifizieren. Die Last des Managements von Zuständen könnte etwas reduziert werden, wenn multiple Empfangsoperationen zugelassen werden:

```
Prozess Adder {
   PortIn  p1: Integer
   PortIn  p2: Integer
   PortOut p3: Integer
   var v1
   var v2

   Ready(p1, p2) =>
        p3.send (p1.receive()+p2.receive());
}
```

Was so zu interpretieren wäre, dass dann, wenn sowohl an Port p1, als auch an Port p2 ein Wert verfügbar ist, die Regel "feuert". Inhaltlich ergibt sich dabei nichts Neues. Die Spezifikation hier kann automatisch in eine äquivalente mit einfachen Empfangsoperationen und mehr Zuständen umgeschrieben werden.[1] Der Benutzer wird auf Kosten der Implementierung etwas entlastet.

Leichtgewichtige reaktive Prozesse zur Verfügung zu haben ist unumgänglich in Anwendungen, die mit massiver Parallelität umgehen. Offen ist die Frage wie dies umgesetzt wird: Das Modell könnte ausschließlich reaktive Prozesse anbieten, oder sowohl reaktive als auch nicht reaktive. Rein reaktive Prozesse sind durch das explizite Zustandsmanagement oft unhandlich. Rein reaktive Systeme könnten einer ereignisorientierten Notation erzwungen werden. Der Zwang dann mit automatischem–Zustandsmanagement durch multiple Empfangsoperationen etwas versüßt werden.

Grundsätzlich ist zunächst natürlich die Frage zu stellen, ob die beschränkte Ausdruckskraft die dem FBP und den visuellen Datenfluss–Systemen zugrunde liegt, nicht schon ausreichend für die Anwendung ist. Die Erfahrung vieler lehrt, dass z.B. für Data–Mining–Aufgaben ein Modell mit statischer Netztopologie und einfachen unveränderlichen Werten als Nachrichten völlig ausreicht und zudem viele Probleme vermeidet. Wenden wir uns trotzdem jetzt Modellen mit höherer Ausdruckskraft zu. – Die natürlich auch in beschränktem Umfang verwendet werden können.

[1] Das gilt nur, wenn das Modell keine "wirkliche" Gleichzeitigkeit kennt, "gleichzeitig" also als "in beliebiger Reihenfolge" interpretiert. In dieser Art von Systemen kann dies üblicherweise vorausgesetzt werden. (Nicht jedoch in Simulationssystemen, siehe unten.)

3 Aktor–Systeme

3.1 Prozess–Algebren

Mit dem Begriff Prozess–Algebra bezeichnet man eine bestimmte Klasse von Formalismen die zur Modellierung nebenläufiger und verteilter Systeme eingesetzt werden. Ihr Kennzeichen und der Ursprung der Bezeichnung als "Algebren" ist die Tatsache, dass komplexe Systeme aus einfacheren Bestandteilen mit quasi algebraischen Operationen zusammengesetzt werden. Die Basis bildet eine kleine Menge an "Basis–Prozessen" die mit einigen wenigen Kompositionsoperationen zu komplexeren Prozessen zusammengesetzt werden.

Ein "Prozess" ist hierbei mit seinem Verhalten zu identifizieren und bei der Komposition wird aus einem Einzelverhalten ein Gesamtverhalten erzeugt. Insgesamt wird also in "algebraischer Art" mit Verhalten umgegangen: Ein Prozess wird als Term definiert und dieser Term hat ein bestimmtes Verhalten als Bedeutung. Das Verhalten selbst kann dann als Transtionssystem formalisiert werden.

Prominente Vertreter von Prozess–Algebren sind CSP [12] und CSS [13]. In dem von Tony Hoare entwickelten CSP kommunizieren Prozesse mit ihrer Umwelt und mit anderen Prozessen durch Ereignisse die sie "gemeinsam ausführen". So ist beispielsweise in

$$Q = \alpha \rightarrow P$$

Q der Prozess der am Ereignis $alpha$ teilnimmt und sich dann wie P verhält.

Wird Q mit einem Prozess S mit

$$S = \alpha \rightarrow T$$

in einer parallelen Komposition zusammen gebracht

$$Q||S \ = \ (\alpha \rightarrow P)||(\alpha \rightarrow P) \ = \ \alpha \rightarrow P||Q$$

dann führen sie gemeinsam Ereignis α aus und verhalten sich dann wie die parallele Komposition von P und Q.

CSS von Milner ist in der gleichen Art konstruiert, hat aber andere elementare Prozesse und Operatoren. Der Sinn einer Prozessalgebra kommt in CSP und CSS gleichermaßen zu Ausdruck: Sie bieten eine Basis auf der das Verhalten verteilter Systeme eindeutig definiert werden kann und in dem dann rigoros über deren Eigenschaften argumentiert werden kann.

Die Ausdruckskraft von Prozessalgebren ist durch die Möglichkeit rekursiver Prozessdefinitionen größer als die des FBP. Ein FBP–System kann in einer Prozessalgebra modelliert werden, aber nicht jedes in einer Prozessalgebra definierte System kann als FBP–Programm realisiert werden. Mit

seinem π–Kalkül [14] ist Milner einen Schritt weiter gegangen: Die Topologie ist dynamisch und Prozesse können selbst als Nachrichten versendet werden.

Die genannten Beispiele von Prozessalgebren wurden in diversen Varianten weiter entwickelt und sind weiterhin Forschungethema. Prozess–Algebren sind entsprechend ihrer Zielsetzung "axiomatisch", d.h. sie bieten nur möglichst elementare Ausdrucksmittel. Das erleichtert Beweisführungen und andere formale Argumentationen, für den praktischen Einsatz ist dies natürlich ein Hindernis und man wird auf Programmiersprachen zurück greifen wollen, deren Semantik auf einem formalen Modell basiert.

3.2 Das Aktormodell

Die Kommunikation in allen Prozessalgebren ist synchron. Das Aktor-Modell der Verteiltheit basiert dagegen auf asynchroner Kommunikation. Ansonsten ist es mit dem π–Kalkül vergleichbar. (In [15] wird entsprechend argumentiert, ein formaler Beweis ist dem Autor nicht bekannt.)

Das Aktormodell geht auf C. Hewitt [16] zurück. Die formalen Grundlagen wurden in [17] erstmals untersucht. Aktoren nach [17] sind Prozesse die

- Nachrichten senden und
- Nachrichten (strikt sequentiell) empfangen können.
- Aktoren reagieren auf eintreffende Nachrichten mit ihrem aktuellen Verhalten.
- Jeder Empfang kann das zukünftige Verhalten des Aktors beeinflussen.
- Nur der Empfang einer Nachricht kann das Verhalten eines Aktor beeinflussen.
- Nachrichten können Aktoren sein.
- Nachrichten kommen stets an ihrem Ziel an. Die Reihenfolge des Empfangs ist allerdings nichtdeterministisch.
- Die Sende–Operation blockiert nie.

Ein Aktor ist stets in einem Zustand, der Zustand bestimmt sein Verhalten beim Eintreffen einer Nachricht.[2] Ein Aktor hat stets eine Menge von Bekanntschaften denen er Nachrichten senden kann. Die Menge der Bekanntschaften ist dynamisch. Sie kann durch empfangene Nachrichten verändert werden. Aktoren können zudem andere Aktoren erzeugen und als Nachrichten versenden.

[2] Begriff "Zustand" ist dabei streng genommen ein hilfreicher aber abgeleiteter Begriff. Aktoren im Formalismus von Agha haben keinen Zustand, sie empfangen Nachrichten und werden dann zu einem zu neuen oder dem gleichen Aktor.

Die Toplologie der Bekanntschaften (Kanten im Netze) ist also dynamisch und kann jederzeit um neue Aktoren (Knoten im Netz) erweitert werden.

Nachrichten können stets versendet werden. Die Sendeoperation fügt die Nachricht in den Briefkasten (*Mailbox*) des Empfängers ein. Die Briefkästen haben unbegrenzte Kapazität und darum blockieren Sendeoperationen niemals.

Das Aktormodell ist von hoher Ausdruckskraft. Jedes andere Modell der Verteiltheit kann vermutlich im Aktormodell emuliert werden. Die formale Handhabe ist dementsprechend komplexer als die anderer, einfacherer Modelle.

3.3 Aktor–Implementierung in Scala

Das Aktormodell ist als nicht nur ein Formalismus zur Modellierung verteilter Systeme und der Untersuchung der Modelle. Es ist auch die Basis diverser Implementierungen mit denen verteilte Systeme realisiert werden (siehe [18]). Eine der bekanntesten und allgemein als vorbildlich erachteten ist Erlang [19]. Nach dem Vorbild von Erlang wurden die Aktoren von Scala entwickelt [20]. Da Scala sich gut für die Realisation interner DSLs eignet, erwecken Aktoren dort den Anschein orginärer Sprachkonstrukte, sind tatsächlich aber eine Bibliothek mit geschickt gewählter API.

Aktor–Implementierungen, die in realen Anwendungen eingesetzt werden, müssen Effizienzanforderungen genügen, die nicht ganz leicht zu erreichen sind. In [21] werden die wesentlichen Probleme einer Aktor–Implementierung dargestellt:

- Implementierungen, bei denen einem Aktoren ein eigener Prozess oder Thread zugewiesen wird, leiden unter einer aufwändigen Erzeugungsoperationen. Dazu benötigen sie viel Speicherplatz, da jedem Thread/Prozess ein eigener Stack zugewiesen werden muss.
- Ist die Zahl der Prozessorkerne kleiner als die Zahl der Threads kommen aufwändige Kontextwechsel hinzu.
- Die Aktorsemantik erlaubt keine Weitergabe von Referenzen. Bei Nachrichten, die innerhalb eines Adressraums zugestellt werden, können aber durch die Weitergabe von Referenzen erhebliche Effizienzgewinne erreicht werden.

In Scala wird das Problem der Referenz–Übergabe ignoriert. Oder anders ausgedrückt: es bleibt in der Verantwortung des Anwendungsentwicklers eine geeignete Wert– oder Referenzübergabe zu realisieren. Das Problem der Threads wird dadurch gelöst, dass es möglich ist zwei Arten von Aktoren zu definieren:

- Aktoren mit einem Zustand, der in blockierenden Operationen erhalten bleiben muss, und die darum mit Hilfe eines Threads implementiert werden.
- Zustandslose reaktive Aktoren die keinen eigenen Thread benötigen.

Die Wahl zwischen diesen beiden Varianten bleibt dem Anwendungsentwickler überlassen.

4 Simulationen

4.1 Atomare DEVS–Modelle

Discrete Event System Specification (DEVS) ist ein von Zeigler [22], entwickelter Formalismus zur Spezifikation und Analyse realer Systeme. DEVS basiert auf Zustandsübergängen die durch zeitliche Ereignisse ausgelöst werden. Ein DEVS–Modell ist entweder atomar oder zusammengesetzt. Ein zusammengesetztes Modell besteht aus anderen Modellen, atomar oder selbst zusammengesetzt.[3]

Ein atomares Modell hat einen aktuellen Zustand. Jeder Zustand hat eine definierte Verweilzeit, die allerdings auch unendlich sein kann. So lange die Verweilzeit nicht abgelaufen ist, verbleibt das Modell in diesem Zustand, es sei denn ein (Eingabe–) Ereignis tritt ein. Das Ereignis ist mit der aktuellen Zeit assoziiert. Das Ereignis, der aktuelle Zustand, sowie die im aktuellen Zustand verbrachte Zeit bis zum Ereignis bestimmen den neuen Zustand.

Ein Zustandsübergang, der durch eine abgelaufene Verweilzeit ausgelöst wird, wird als interne Transition bezeichnet. Einen durch ein externes Ereignis ausgelösten Zustandsübergang nennt man externe Transition. Mit einer internen Transition, nicht jedoch mit einer externen Transition kann ein Ausgabeereignis verbunden werden.

Ein atomares Modell ist formal eine Struktur $M = < S, X, Y, \delta_{int}, \delta_{ext}, \lambda, \tau >$ mit

- S ist eine Menge von *Zuständen*,
- X ist eine Menge von *Eingabe–Ereignissen*,
- Y ist eine Menge von *Ausgabe–Ereignissen*,
- $\delta_{int} : \ S \to S$ ist die *interne Transitionsfunktion*,
- $\delta_{ext} : \ Q \times X \to S$ ist die *externe Transitionsfunktion*,

[3] Achtung: Im Kontext der Simulation ist "Modell" die Bezeichnung für eine Modellierung der Wirklichkeit, also auch die Bezeichnung für das was bisher Prozess, Prozessnetz oder Aktor und Aktorsystem genannt wurde.

- $\lambda : \ S \to Y$ ist die *Ausgabefunfktion*,
- $\tau : \ S \to \mathbb{R}^+_\infty$ ist die *Zeitfortschrittsfunktion*.

Mit $\mathbb{R}^+_\infty$ werden nicht nicht–negativen reelen Zahlen inklusive "unendlich" bezeichnet. DEVS–Modelle arbeiten also mit dikreten Zuständen und Ereignissen und mit kontinuierlicher Zeit. Q ist die sogenannte totale *Zustandsmenge*. Sie besteht aus Paaren von Zuständen und der in diesem Zustand verbrachten Zeit:

$$Q = \{(s, e) \mid s \in S,\, s \in \mathbb{R}^+_\infty,\, 0 \leq e \leq \tau(s)\}$$

Dei externe Transitionsfunktion δ_{ext} bildet darum Zuständen und die in diesem Zustand verbrachte Zeit auf einen neuen Zustand ab.

Tritt keine externes Ereignis auf, dann verbleibt das Modell im Zustand $s \in S$ so lange bis die Zeit $\tau(s)$ abgelaufen ist. Dann geht es mit einer internen Transition in den Zustand $\delta_{int}(s)$ über. Falls $\tau(s) = 0$ dann erfolgt der Übergang sofort. Bevor die interne Transition abgeschlossen ist, wird as externe Ereignis $\lambda(s)$ generiert. Falls $\tau(s) = \infty$, dann wird die interne Transition niemals ausgeführt.

Wenn im Zustand s ein externes Ereignis x auftritt bevor $\tau(s)$ abgelaufen ist und nachdem die Zeit e in s vergangen ist, wird die externe Transition $\delta_{ext}((s, e), x)$ ausgeführt. Eine externe Transition erzeugt keine Ausgabe. Falls ein Ausgabeereignis erzeugt werden soll, dann muss ein Zustand mit der Verweilzeit 0 zwischengeschaltet werden.

4.2 DEVS–Modelle mit Ports: Parallelität und Gleichzeitigkeit

Atomare Modelle können mit Ports ausgestattet werden. Ein Port ist ein Punkt an dem ein Modell ein Ereignis annehmen oder aussenden kann. Ports dienen auch hier der Modularität. Ein Ereignis wird an einem Port empfangen oder an einen Port gesendet. Der tatsächliche Sender und Empfänger kann dann später festgelegt werden.

Inhaltlich sind Ports zunächst nur als Mittel der Strukturierung und Modularisierung interessant. Sie können formal rein syntaktisch als zusätzliche Markierungen von Ereignissen behandelt werden. Ein Ergebnis, das an Port p auftritt ist ein Ereignis, das die Markierung (den Index) p trägt. Verbindungen zwischen Ports können formal als systematische Umbenennungen der Ports behandelt werden.

Das Jonglieren mit den entsprechenden Formalismen ist nicht weiter interessant und unterscheidet sich de facto nicht von der Behandlung von Ports im FBP. Einen entscheidenden Unterschied gibt es aber. Bei einer Simulation ist "Zeit" ein Teil der Berechnung. Es kann darum, bei der Modellierung eines realen Systems angebracht sein, ein Konzept von "Gleichzeitigkeit" zur Verfügung zu haben. Es ist – eventuell – sinnvoll zu unterscheiden zwischen zwei Ereignissen die in beliebiger Reihenfolge auftreten und solchen die gleichzeitig auftreten.

In FBP, Prozessalgebren und Aktorsystemen gibt es keine Gleichzeitigkeit, bzw. "gleichzeitig" ist äquivalent zu "die Reihenfolge spielt keine Rolle". Ports sind darum dort rein syntaktische Ausdrucksmittel ohne tiefere semantische Relevanz. Wenn Gleichzeitigkeit eine Rolle spielen kann, dann müssen aber *"e1, dann e2"*, *"e2, dann e1"* und *"e1 und gleichzeitig e2"* unterscheidbar sein.

Da atomare DEVS–Modelle immer strikt sequenziell sind, tritt eine solche Gleichzeitigkeit nur auf, wenn mehrere atomare Modelle zusammen geschaltet werden. Alle weiter oben angesprochenen Modelle der Verteiltheit basieren auf (strikt sequenziellen) Transitionssystemem. Einzelne Prozesse definieren jeweils ein Transitionssystem und die Zusammenschaltung definiert ein "induziertes" Transitionssystem (siehe etwa [1]). Das induzierte Transitionssystem ist ebenfalls strikt sequentiell. – Gleichzeitigkeit wird bei der Zusammenschaltung "in Nichtdeterminismus aufgelöst". Bei einer Simulation kann das die falsche Vorgehensweise sein.

Mit PDEVS [23] wurden *parallele DEVS Modelle* vorgeschlagen, ein Formalismus der es erlaubt Gleichzeitigkeit auszudrücken. Der Formalismus ist leicht abgewandelt:

Ein atomares Modell ist eine Struktur M mit:

$$M = < S,\, X_M,\, Y_M,\, \delta_{int},\, \delta_{ext},\, \lambda,\, \tau >$$

wobei

- S ist eine Menge von *Zuständen*,
- $X_M = \{(p, v) | p \in IP\, v \in X_p\}$ ist eine Menge von *Eingabe–Ereignissen*,
- $Y_M = \{(p, v) | p \in OP\, v \in Y_p\}$ ist eine Menge von *Ausgabe–Ereignissen*,
- $\delta_{int} :\ S \to S$ ist die *interne Transitionsfunktion*,
- $\delta_{ext} :\ Q \times X_M^b \to S$ ist die *externe Transitionsfunktion*,
- $\delta_{con} :\ Q \times X_M^b \to S$ ist die konfluente *Transitionsfunktion*,
- $\lambda :\ S \to Y$ ist die *Ausgabefunktion*,
- $\tau :\ S \to \mathbb{R}_\infty^+$ ist die *Zeitfortschrittsfunktion*.

Die Interpretation dieses Formalismus' erweitert die nicht–parallele Variante in zwei Punkten: Eine externe Transition wird jetzt nicht mehr von einem Eingabeereignis (mit-) bestimmt, sondern von einer Menge von Eingabe–Ereignissen an Ports ($Q \times X_M^b$). Dazu kommt noch eine eine konfluente Transition die die Kollision einer internen mit einer externen Transition auflösen soll. Sie wird angewendet, wenn eine interne und eine externe Transition gleichzeitig ausgeführt werden könnten. In dem Fall wird nach Vorgabe der konfluenten Transition serialisiert – atomare PDEVs–Modelle sind immer noch strikt sequentiell.

Die konfluente Transition kann per default definiert werden entweder als

$$\delta_{con}((s,e),x) = \delta_{ext}((\delta_{in}(s),0),x)$$

oder als

$$\delta_{con}((s,e),x) = \delta_{in}(\delta_{ext}((s,e),x).$$

Entweder die interne oder die externe Transition wird zuerst ausgeführt.

PDEVS ist ein komplexer Formalismus aber wohl auch der Formalismus mit der größten Ausdruckskraft. Alle anderen Modelle können wohl in PDEVS formuliert werden.[4]

4.3 Simulation von PDEVS–Modellen

PDEVS ist ein formales Modell. Es sagt nichts darüber aus, wie die Modelle realisiert werden. Ein PDEVS–Modell kann mit einer strikt sequentiellen, einer parallelen oder einer verteilten Implementierung – einem "Simulator" – abgearbeitet werden. Es ist wichtig zwischen dem Modell–Formalismus, dem Modell und dem Simulator zu unterscheiden ([22]).

- *Formalismus* (leider auch oft und auch hier "Modell" genannt): Die Sprache (Syntax und Semantik) in der Modelle formuliert werden.
- *Modell*: Darstellung eines Sachverhalts in der Realität.
- *Simulator*: Ausführender einer Simulation.

Der Simulator eines im PDVS–Formalismus definierten Modells muss also nicht unbedingt parallel oder gar verteilt sein. Parallelität und Verteilung sind aber auch hier die üblichen Mittel um mit aufwendigen Problemstellungen umzugehen.

Ein Simulator kann grundsätzlich auf zwei Arten realisiert werden: *zeitgesteuert* oder *ereignisgesteuert*. Zeitgesteuerte Algorithmen werden von einer zentralen Uhr getrieben, die zu jeder Uhrzeit alle Ereignisse und die ihnen zugeordneten Transitionen auslöst, die zu dieser Zeit anliegen.

Ereignisgesteuerte Algorithmen verwalten eine zentrale, nach Zeiten priorisierte Warteschlange von Ereignis–Zeit–Paaren die sie abarbeiten. Eine Uhr ist unnötig. Die aktuelle Uhrzeit ist die Zeit des ersten Ereignisses in der Warteschlange. Zeitgesteuerte Algorithmen sind unüblich. Ereignissgesteuerte Simulationen sind einfacher und effizienter zu implementieren.

[4] Das gilt auch für hier nicht angesprochene Modellierungsformalismen wie Petrinetze [24].

Eine verteilte Simulation kann *zentral* oder *dezentral* sein. Ein zentraler Simulator behält die zentrale Uhr bzw. die zentrale Warteschlange bei. Ein *dezentraler* Simulator arbeitet mit lokalen Uhren oder lokalen Warteschlangen. Diese müssen dann natürlich geeignet synchronisiert werden. Diese Synchronisation muss garantieren, dass Ereignisse nicht "rückwärts in der Zeit" ausgeführt werden.

Dezentrale Lösungen sind von hohem akademischen Interesse und haben eine intensive Forschungsarbeit provoziert. Für eine Übersicht und Bewertung der verschiedenen Ansätze siehe [25].

Literatur

[1] Gerard Tel
Introduction to Distributed Algorithms
Cambridge University Press; 2nd edition, 2000

[2] Richard Stephens
A Survey Of Stream Processing
Acta Informatica Acta Informatica 34,7 (1997) (auch verfügbar über CiteSeer)

[3] Michael R. Berthold, et al.
KNIME – The Konstanz Information Miner
in: Data Analysis, Machine Learning and Applications Studies in Classification, Data Analysis, and Knowledge Organization, 2008, V, 319-326

[4] J. Paul Morrison
Flow based programming: A new approach to application development
Van Nostrand Reinhold (New York) 1994

[5] Sven Steinseifer
Evaluation and Extension of an Implementation of Flow-Based Programming
Master–Thesis, THM, 2009

[6] Philipp Hoffmann
Ausdrucksmittel für datenflussorientierte Datenanalyse
Master–Thesis, THM, 2011

[7] Niels Braden
Konzeption und Implementierung eines Laufzeitsystems zur Ausführung von Datenfluss–Netzen in Java
Master–Thesis, THM, 2011

[8] Martin Fowler
Domain-specific languages
Addison-Wesley Professional, 2010

[9] Björn Kasteleiner
Eine Domänen spezifische Sprache für datenflussorientierte Berechnungsnetzwerke realisiert mit einer Xtext-Implementierung
Master–Thesis, THM, 2011

[10] G. Kahn
The Semantics of a Simple Language for Parallel Programming
Proc. of the IFIP Congress 74, North-Holland, 1974

[11] G. Kahn, D. B. MacQueen
Coroutines and Networks of Parallel Processes
Information Processing 77, B. Gilchrist, editor, North-Holland, 1977

[12] C.A.R. Hoare
Communicating Sequential Processes
Prentice Hall, 1985 (verfügbar über http://www.usingcsp.com/)

[13] Robin Milner
Communication and Concurrency
Prentice Hall, 1989

[14] Robin Milner
Communicating and Mobile Systems: The Pi-Calculus
Cambridge University Press, 1999

[15] Anonymer Autor
Actor model and process calculi history
Wikipdia: http://en.wikipedia.org/wiki/Actor_model_and_process_calculi_history

[16] Carl Hewitt, Peter Bishop, Richard Steiger
A universal modular ACTOR formalism for artificial intelligence
IJCAI'73 Proceedings of the 3rd international joint conference on Artificial intelligence, 1973
Verfügbar über: http://dl.acm.org/citation.cfm?id=1624775.1624804

[17] Gul Agha
Actors: A Model of Concurrent Computation in Distributed Systems
MIT Press, 1986

[18] Rajesh K. Karmani, Gul Agha
Actors
Bericht, Open Systems Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign

[19] Joe Armstrong
Programming Erlang: Software for a Concurrent World
Pragmatic Bookshelf, 2007

[20] Phillip Haller, Frank Sommers
Actors in Scala
Artima, 2011

[21] Rajesh K. Karmani, Amin Shali, and Gul Agha
Actor frameworks for the JVM platform: a comparative analysis.
Proc. ACM Conference on Principles and Practice of Programming in Java, pp. 11–20, 2009.

[22] Bernard P. Zeigler, Tag G. Kim, Herbert Prähofer
Theory of Modeling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic System
Academic Press; 2nd revised edition, 2000

[23] A. C. Chow, B. Zeigler
Parallel DEVS: A parallel, hierarchical, modular modeling formalism
Proceedings of the Winter Computer Simulation Conference, Orlando, FL. 1994.

[24] Hans L.M. Vangheluwe
DEVS as a Common Denominator for Multi-formalism Hybrid Systems Modelling
Proc. IEEE International Symposium on Computer-Aided Control System Design, pages 129-134. IEEE Computer Society Press, 2000

[25] Jafer Shafagh
Parallel Simulation Techniques for Large-Scale Discrete Event Models
Doctoral Thesis, Carleton University, Ottawa, August 2011
zugreifbar über http://www.mcs.uvawise.edu/sj2fd/