7. Ein Überblick über Amica

Trotz der stetig wachsenden Prozessorleistung sind Anwender immer noch auf den Einsatz entfernter Hochleistungsrechner angewiesen, um eine akzeptable Antwortzeit für ihre Applikationen zu erhalten. Dies gilt insbesondere für multidisziplinäre Anwendungen, deren Berechnung mehrere Hochleistungsrechner auslasten kann, wie z.B. Klimasimulationen oder Unfallsimulationen von Kraftfahrzeugen. Da einer Organisation im Allgemeinen nicht ausreichend viele derart leistungsfähige Maschinen in ihrem lokalen Netz zur Verfügung stehen, setzt man einen domänenübergreifenden heterogenen Rechnerverbund ein. Diese Verbunde nennt man *Metacomputer* [SC92] oder, wenn man den offenen Charakter unterstreichen möchte, auch *Gridcomputer* [FK98, Ski02].

Um hier einen einheitlichen Namen einzuführen, werden solche verteilten Systeme, die eine hohe Heterogenität und Dynamik aufweisen, im folgenden *weit verteilte Systeme* genannt. Diese stehen im Gegensatz zu *lokal verteilten Systemen*, wie Workstation-Cluster und Parallelrechner. Die Ansätze zur Applikationsentwicklung für weit verteilte Systeme lassen sich grob wie folgt klassifizieren:

1. *Transparente Applikationsentwicklung*
   Ansätze, die sich für Parallelrechner und lokal verteilte Systeme bewährt haben, werden auf weit verteilte Systeme erweitert. Bei dem Einsatz paralleler Programmiersprachen, die eine verteilungstransparente Erstellung nebenläufiger Applikationen ermöglichen, muss nur das Subsystem für die Codeerzeugung ersetzt und das Subsystem für die Codeoptimierung an die unterschiedlichen Latenzzeiten in weit verteilten Systemen angepasst werden. Dann können Applikationen ohne händische Anpassung auf beliebigen weit verteilten Systemen ausgeführt werden. In dem System Legion lassen sich z.B. Applikationen in der Sprache MPL verteilungstransparent entwickeln [GW96].

   Wird für die Erstellung einer nebenläufigen Applikation auf eine Bibliothek zurückgegriffen, die Primitive für Kommunikations- und Nebenläufigkeitskontrolle anbietet, so muss diese auf das weit verteilte System portiert werden. Vorhandene Applikationen sind dann prinzipiell nur neu zu binden, um eingesetzt werden zu können. Um eine gute Laufzeiteffizienz zu erzielen, ist es aber häufig notwendig, die Applikation händisch an das konkrete System anzupassen. Ein Beispiel für diesen Ansatz bietet die Umsetzung der MPI-Bibliothek für das Globus-System [FK97] und für das Avaki-System[ava02].

2. *Explizite Programmierung*
   Hier stehen dem Anwendungsentwickler Systemschnittstellen zur Verfügung, um direkt auf das weit verteilte System zuzugreifen. Er entwickelt seine Applikation also speziell für das konkrete weit verteilte System und kann so alle angebotenen Dienste nutzen. Dieser Ansatz birgt das Potenzial, effiziente Applikationen mit großer Funktionalität entwickeln zu können, wenn die Systemschnittstellen entsprechend reichhaltig und qualitativ hochwertig sind. Er beschränkt andererseits die Portierbarkeit erheblich. Jedes dem Autor bekannte weit verteilte System bietet solche Schnittstellen an. So bietet z.B. das Globus-System reichhaltige Bibliotheken an, die einer Applikation expliziten Zugriff auf Rechenressourcen, Netzwerke und das Sicherheitssystem ermöglichen [FK97, glo02].

3. *Grobkörnige Komposition von Subapplikationen*
   Bei diesem Ansatz werden Applikationen für weit verteilte Systeme aus Subapplikationen komponiert, die für lokal verteilte Systeme entwickelt sind. Ein Beispiel für diesen Ansatz ist das verbreitete und häufig eingesetzte System Condor [con02]. Bei diesem besteht das weit verteilte System aus verteilten Rechenressourcen, die Aufträge in Form ausführbarer Dateien entgegennehmen, in eine Wartschlange einfügen und schließlich ausführen. Die Hauptapplikation besteht i.a. aus einem Skript, das lokal auf einem Rechner ausgeführt wird. Es erzeugt die einzelnen Aufträge, übergibt sie dem Condor-System und verarbeitet die erzeugten Ergebnisse. Dieser Ansatz stellt einen Kompromiss zwischen den obigen Ansätzen dar.

Da jedem dieser drei Ansätze unterschiedliche Vor- und Nachteile eigen sind, enthält jedes konkrete Programmiersystem i.a. Eigenschaften, die zu unterschiedlichen Klassen gehören. Trotzdem lassen sich häufig Schwerpunkte identifizieren, die es erlauben, ein Programmiersystem einer Klasse zuzuordnen. Um nun diese Klassen hinsichtlich ihrer Eignung als Grundlage für eine Koordinationssprache für weit verteilte Applikationen zu bewerten, müssen zuerst die Eigenschaften der Applikationen analysiert werden, die für derartige Systeme entwickelt werden.

Multidisziplinäre Anwendungen (MDA) sind i.A. so rechenintensiv, dass sie parallel von unterschiedlichen Hochleistungsrechnern bearbeitet werden müssen, um eine akzeptable Antwortzeit zu erzielen. Sie bilden eine der wichtigsten Applikationsunterklassen für die betrachteten Systeme und dienen daher hier als Entscheidungsgrundlage für eine geeignete Koordinationssprache. In [MRZ98] wird von den folgenden Eigenschaften von MDA berichtet:

- MDA setzen häufig auf bestehenden Softwarekomponenten auf. Das können fremdentwickelte Bibliotheken oder spezielle Eigenentwicklungen sein, die evtl. an spezielle Hardware, wie z.B. bestimmte Hochleistungsrechner, an Software, wie z.B. Laufzeitumgebungen, oder an Domänen, z.B. durch Lizenzen, gebunden sind. Diese Komponenten sind häufig inhärent parallel.

- Für die meisten MDA ist es ausreichend, eine Koordinationssprache mit *einfachen Sprachkonstrukten* einzusetzen, wie z.B. die Unterstützung von Master/Worker- oder Producer/Consumer-Ansätzen. Wichtig ist allerdings ein *dynamischer Parallelitätsgrad*, der sich an die Eingabedaten anpassen kann.

Aus diesen Eigenschaften lässt sich direkt ableiten, dass der Ansatz der transparenten Applikationsentwicklung für MDA nicht geeignet ist. Altsoftware ist meist nur mit großem Aufwand auf ein anderes Programmiersystem zu portieren. Dies gilt vor allem für den Fall, dass der Quellcode nicht zur Verfügung steht. Zusätzlich können Lizensierungsrechte oder Abhängigkeiten von Spezialhardware es prinzipiell unmöglich machen, Altsoftware auf anderen Zielplattformen einzusetzen.

Auch wenn man annimmt, dass alle benötigten Softwarekomponenten in einem brauchbaren Format vorliegen, welches von der Zielplattform unterstützt wird, so ist es dennoch schwierig, eine gute Effizienz der verteilten Applikation zu gewährleisten. Als Beispiel zur Verdeutlichung soll hier das .NET System dienen [TL02]. Es unterstützt prinzipiell die Komposition verteilter SW-Komponenten, die in beliebigen Sprachen vorliegen[1]. Um nun eine gute Effizienz zu erreichen, müssen die Softwarekomponenten an die Rechenressourcen angepasst werden, auf denen sie laufen sollen. Wenn solche Ressourcen Parallelrechner mit speziellen Kommunikationsnetzwerken sind, ist eine automatische Anpassung sehr schwierig. Dies gilt insbesondere für Programmiersprachen wie z.B. C++, die nur schwer zu analysieren sind. Die Lösung, die man in der Praxis häufig antrifft, besteht aus dem Einsatz von Bibliotheken, die speziell für die bestimmte Rechnerarchitektur optimierten Code enthält.

Der Ansatz der expliziten Programmierung ist natürlich möglich und bietet Potenzial für eine gute Effizienz. Wie allerdings schon erwähnt, ist er sehr aufwendig. Da sich MDA häufig mit einfachen Sprachkonstrukten beschreiben lassen, scheint dieser Ansatz hier übertrieben. Er bietet eine ungenügende Programmiereffizienz, in dem Sinne, dass der Aufwand für die Erstellung einer Applikation und für ihre Anpassung an unterschiedliche Systeme nicht gerechtfertigt ist.

Der dritte Ansatz der grobkörnigen Komposition stellt sich als sehr geeignet für MDA dar. Altsoftwarekomponenten lassen sich mit einfachen Sprachkonstrukten zu einer verteilten Applikation komponieren. Müssen die einzelnen Rechenkomponenten neu entwickelt werden, kann man für lokal verteilte Systeme auf Werkzeuge und Bibliotheken zurückgreifen. Diese können durch Profiling oder mit statischer Analyse bei typischen numerischen Algorithmen effiziente Implementierungen für vorgegebene Rechnerarchitekturen erzeu-

[1] Hierbei ist anzumerken, dass .NET zwar eine breite Sprachheterogenität unterstützt und dies auch durch Implementierungen von Programmiersprachen belegt, die unterschiedlichsten Programmiermodellen folgen. Betrachtet man diese Programmiersprachen allerdings genauer stellt man fest, dass sie alle etwas verändert wurden, um sich gut in das .NET System zu integrieren. Daher lässt sich vermuten, dass die Integration von Altsoftware auch dann eine händische Nachbearbeitung erfordert, wenn die eingesetzte Programmiersprache unterstützt wird.

gen[2].

Um solche Komponenten einzubinden, bietet es sich an, diese uniform zu umhüllen. Sie stellen sich dann der Applikation als allgemeine Rechendienste dar. Neben den Rechenkomponenten müssen auch noch interaktive Komponenten vorhanden sein, wobei Interaktion hier sowohl die Steuerung von Algorithmen (Steering) als auch allgemeine Ein- und Ausgabeoperationen umfasst. Ähnliche Ansätze zur Einbindung von Rechendiensten verfolgen die Projekte DISCWorld [KHCF98], das eine proprietäre Infrastruktur für Rechendienste bereit stellt, und Bayanihan [SCE+02], welches auf .NET Webservices aufsetzt.

Im folgenden wird das auf ECL aufsetzende System *Amica (Abstract Metacomputing Infrastructure for coarse-grained Applications)* vorgestellt, welches ebenfalls den dritten Ansatz der grobkörnigen Komposition zur Implementierung von MDA verfolgt. Hierfür stellt es verschiedene Erweiterungsmodule und eine verteilte Infrastruktur bereit. Amica wurde so generisch entworfen, dass es sich einfach auf konkrete Systeme, wie. z.B. Globus portieren lässt. An einem Beispiel werden zunächst die einzelnen Entwurfsentscheidungen für das System motiviert. Dann wird seine Strukturierung vorgestellt. Die dann folgenden Kapitel 8.1, 8.2, 9, 9.2, 9.3 und 9.4 beschreiben die einzelnen Erweiterungsmodule detailliert. In Kapitel 10 werden abschließend einige Applikationen präsentiert, die in Amica realisiert wurden.

7.1. Motivierendes Beispiel

Abbildung 7.1 stellt schematisch ein typisches Beispiel einer rechenintensiven verteilten Applikation dar. Verteilte Sensoren liefern einen Strom von Messdaten, wie. z.B. Klimadaten. Diese werden von einer Rechenkomponente vorverarbeitet und auf eine weiterverarbeitbare Größe reduziert. Hierfür kann schon eine enorme Rechenleistung notwendig sein. So liefern die Sensoren des neuen Partikelbeschleunigers LHC des CERN voraussichtlich eine Datenmenge von einigen Petabytes/s, die auf einige Petabytes/y, also ungefähr um den Faktor 10^7 reduziert werden müssen [Hof02].

Das Ergebnis der Vorverarbeitung ist eine Menge von Daten, die unabhängig von dem Experiment ausgewertet werden können. In diesem Beispiel besteht diese Auswertung aus einer Simulation, die auf den Eingabedaten durchgeführt wird. Bestehen diese aus Klimadaten, so könnte die Simulation z.B. eine Wettervorhersage berechnen. Bei sehr langlaufenden Simula-

[2] Ein Ansatz hierfür besteht darin, numerische Algorithmen von regulärer Struktur, wie z.B. die Multiplikation von Matrizen, als affine Rekurrenzgleichungen zu beschreiben. Diese lassen sich dann statisch analysieren und z.B. auf einen SIMD-Rechner abbilden, dessen Kommunikationsnetzwerk i.a. reguläre Kommunikationsstrukturen, wie sie in den Algorithmen eingesetzt werden, gut unterstützt [PGFE98]. Als einfaches Beispiel sei eine verschobene Addition zweier Matrizen der Größe $n \times n$ gegeben durch $C[i,j] := A[i,j] + B[(i+1)\%n, (j+1)\%n]$. Sei die Zielarchitektur nun ein SIMD-Rechner mit $n \times n$ Prozessoren, in dem jeder Prozessor mit seinen vier Nachbarn kommunizieren kann. Mit statischer Analyse kann ein Übersetzer den Kommunikationsaufwand dadurch minimieren, indem er $A[i,j]$ auf den Prozessor $P_{i,j}$ und $B[i,j]$ auf $P_{(i-1)\%n,(j-1)\%n}$ abbildet.

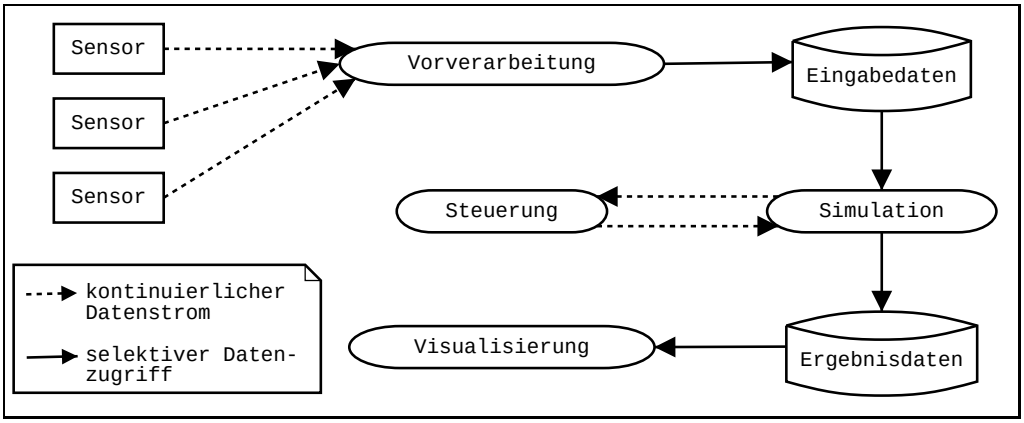

Abbildung 7.1.: Ein typisches Beispiel einer rechenintensiven verteilten Anwendung

tionen, ist es üblich, diese im laufenden Betrieb zu steuern. Dies ermöglicht u.a. eine rechtzeitige Anpassung von Simulationsparametern. Hierfür werden Informationen über den aktuellen Berechnungszustand an eine Steuerungskomponente geschickt. Diese präsentiert die Daten einem Benutzer, der dann geeignet reagieren kann. Diese Reaktion wird als Steuerungssignal zurück an die Simulation geschickt. Das Ergebnis der Simulation ist wieder eine Menge von Daten, die dann visualisiert und ausgewertet wird.

Aus diesem Beispiel lassen sich Anforderungen ableiten, welche Architekturelemente einem Entwickler derartiger Applikationen zur Verfügung gestellt werden sollten. Das zentrale Element ist eine Rechenkomponente, welche die Ausführung rechenintensiver Subapplikationen, wie die Vorverarbeitung und die Simulation, unter Einsatz entfernter leistungsfähiger Rechenresourcen übernimmt. Es ist davon auszugehen, dass solche Subapplikationen in sehr unterschiedlichen Formen vorhanden sind. Daher muss die Rechenkomponente einen starken generischen Charakter haben.

Des weiteren sind aufgrund des hohen Datenaufkommens spezielle Komponenten zur Datenspeicherung notwendig. Als Kontrollparadigmen müssen sowohl Arbeitsablaufpläne als auch Datenflusselemente unterstützt werden, die beide kombiniert in dem Beispiel eingesetzt werden. Bei Arbeitsablaufplänen werden die Daten in Behältern gespeichert, auf die selektiv zugegriffen werden kann. In der Abbildung ist dieser selektiver Zugriff durch Pfeile mit durchgezogenen Linien dargestellt. Bei dem Einsatz des Datenflussparadigmas dagegen, fließen die Daten direkt von den Erzeugern zu den Verbrauchern. Dies wird in der Abbildung durch gestrichelte Pfeile visualisiert.

7.2. Strukturierung des Systems

Abbildung 7.2 stellt die einzelnen Erweiterungsmodule und ihre Abhängigkeiten als UML-Komponentendiagramm dar. Das Erweiterungsmodul `Workflow` ist das in Kapitel 6 vorgestellte Erweiterungsmodul für arbeits-

ablaufbasierte Applikationen. Das Erweiterungsmodul `DataTypes` stellt ein Typsystem für Daten zur Verfügung, die in Amica-Applikationen benutzt werden können.

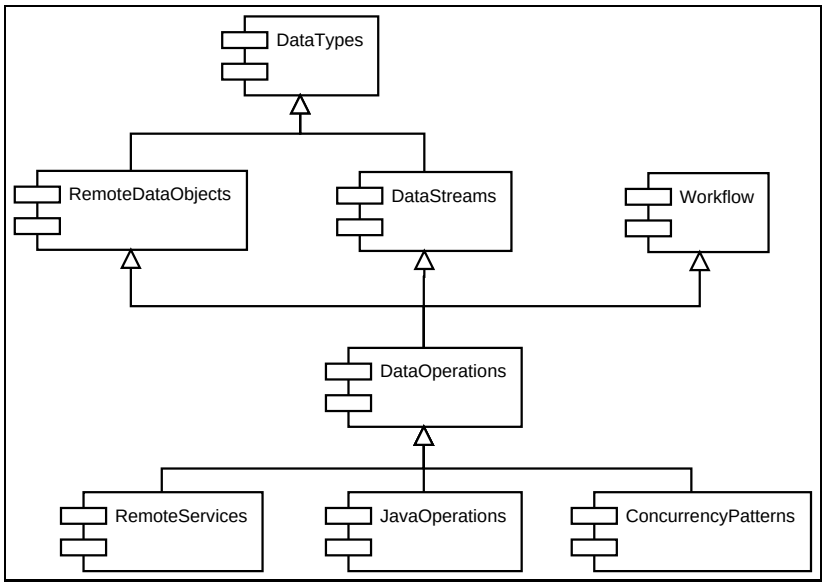

Abbildung 7.2.: Erweiterungsmodule für das Amica System

Dieses Typsystem wird von zwei Modulen benutzt, welche konkrete Architekturelemente für die Manipulation von Daten in Applikationen bereitstellen. Das Modul `RemoteDataObject` enthält entfernte typisierte Datenobjekte, die entsprechend ihres Typs Schnittstellen für einen selektiven Zugriff zur Verfügung stellen. Das Modul `DataStreams` dagegen unterstützt entfernte Ströme von typisierten Daten.

Um nun Operationen auf diesen Daten zur Verfügung zu stellen, enthält das Modul `DataOperations` abstrakte Architekturelemente, die auf den Operationselementen des `Workflow` Moduls aufsetzen. Diese Operationen benutzen zur Interaktion die obigen Datenobjekte und Datenströme und ermöglichen so die entfernte Bearbeitung großer Datenmengen.

Es gibt zwei Module, die konkrete Operationen bereitstellen. Das Modul `RemoteServices` enthält Komponenten, die den Zugriff auf entfernte Rechendienste ermöglichen, während das Modul `JavaOperations` lokale in Java implementierte Operationen ausführt. Letztere dienen zur Interaktion mit dem Benutzer und zur Überbrückung inkompatibler Rechendienste. Schließlich enthält das Modul `ConcurrencyPatterns` Konnektoren, um typische Kontrollkonstrukte in nebenläufigen Applikationen zu beschreiben. Die einzelnen Module werden in den folgenden Kapiteln detailliert beschrieben.

7.3. Verwandte Arbeiten

Auf dem Gebiet der rechenintensiven Applikationen in weit verteilten Systemen gibt es ein Vielzahl von Werkzeugen und Infrastrukturen, welche die Er-

stellung und Abarbeitung derartiger Applikationen unterstützen. Daher beschränkt sich die Beschreibung verwandter Arbeiten im folgenden auf Systeme, die, wie Amica, es ermöglichen, eine Applikation durch Komposition von rechenintensiven Komponenten eventuell mit einer grafischen Entwicklungsumgebung zu erstellen. Systeme, die nur eine grafische Schnittstelle bieten, um einzelne rechenintensive Komponenten zu verteilen und abzuarbeiten, wie z.B. WebSubmit [MKD99] und ReGTime [DHUZ97], werden nicht betrachtet. Ebenso werden Systeme vernachlässigt, die für die Applikationsentwicklung nur eine Komponentenumgebung ohne eine darauf aufsetzende Koordinationssprache bereit stellen, wie z.B. CAT [VGS+99]. Schließlich werden auch die Ansätze nicht vorgestellt, welche die eigentlichen aufwendigen Berechnungen in Java durchführen, wie z.B. Charlotte [BKKK98] oder JavaParty [PZ97].

UNICORE wurde mit dem Ziel entworfen, Applikationen, die aus einzelnen rechenintensiven Komponenten bestehen, unter Einsatz von Rechenressourcen unterschiedlicher Rechenzentren auszuführen [ES01, AS98, uni02]. Applikationen werden dabei als gerichtete azyklische Kontrollflussgraphen spezifiziert. Die Knoten beschreiben entweder Rechenoperationen, die aus der entfernten Ausführung von Programmen des Applikationsentwicklers bestehen, oder Datenübertragungen. Die Koordinationssprache von UNICORE ist damit relativ eingeschränkt. Der Schwerpunkt von UNICORE liegt eher auf der Entwicklung eines Werkzeugs, welches dem Benutzer den administrativen Aufwand bei der domänenübergreifenden Nutzung von Rechenressourcen abnimmt.

In dem System *VDCE* (Virtual Distributed Computing Environment) werden Applikationen als Graphen erstellt, wobei die Knoten entweder Dateien oder bestimmte Rechenoperationen darstellen, die in Bibliotheken in Form von ausführbarem Code zur Verfügung stehen [HTF+98]. Wie bei UNICORE sind die Graphen ebenfalls azyklisch und es stehen keine weiteren Elemente zur Steuerung des Kontrollfluss zur Verfügung. Der Schwerpunkt von VDCE liegt mehr bei der effizienten Verteilung einer Applikation auf die vorhandenen Ressourcen.

Symphony setzt, zumindestens konzeptionell, ebenfalls azyklische Kontrollflussgraphen ein [LK02]. Die einzelnen Knoten einer Applikation bestehen aus JavaBeans, die beliebige Funktionalität bereitstellen können. Der Anwender kann sich einen beliebigen Knoten auswählen und diesen starten. Der gestartete Knoten startet dann die Knoten, die vor ihm bearbeitet werden müssen, und führt seine Operation aus. Die nachfolgenden Knoten werden nicht automatisch benachrichtigt. Symphony ist damit im Gegensatz zu dem Amica-System als interaktives Werkzeug konzipiert. Der Applikationsentwickler kann interaktiv Teile der Applikation starten und weiter bearbeiten.

Gecko setzt auf Globus auf und benutzt Taskgraphen, um lose gekoppelte Applikationen zu beschreiben [LF99]. Die Graphknoten sind dabei Jobs, die in Globus ausgeführt werden.

Ein alternativer Ansatz für die Koordinationssprache besteht in dem Einsatz des Datenflussparadigmas. Hier beschreibt jeder Knoten eine Funktion, die auf die Daten der eingehenden Kanten angewendet wird, und deren Er-

gebnisse über die ausgehenden Kanten weitergeschickt werden. Ein System, welches das Datenflussparadigma einsetzt, ist das Geographical Information System *(GIS)*. Da aber auch GIS keine weiteren Elemente zur Steuerung des Kontrollflusses anbietet, sind auch hier die Applikationen auf azyklische Graphen beschränkt. GIS basiert auf dem System *DISCWorld*, das wie Amica alle Rechenressourcen als Rechendienste modelliert [KHCF98]. Allerdings unterstützt es keine typisierten Datenobjekte.

Das System *WebFlow* benutzt als Koordinationsparadigma die Steuerung über Ereignisse [HAF99]. Der Applikationsentwickler sucht sich aus einer Bibliothek seine Komponenten aus. Jede Komponente besitzt eine Menge von Anschlüssen, die sie selber verwaltet. Es gibt also keine allgemeinen Anschlüsse oder Anschlusstypen, wie z.B. einen Anschluss für Aktivierungen bei Operationen in Kontrollflussgraphen oder einen Anschlusstyp für eingehende Daten bei dem Einsatz des Datenflussparadigma. Der Applikationsentwickler verbindet die Anschlüsse und startet die Applikation. Hierfür werden alle Komponenten initialisiert und gestartet. Dieser Ansatz ist sehr allgemein und damit eine gute Grundlage, um zu untersuchen, wie Komponenten entworfen sein sollten, damit sie von einem Applikationsentwickler mit wenig Einarbeitungsaufwand benutzt werden können. Diese Untersuchung ist aber nach Kenntnisstand des Autors noch nicht geschehen.

Ebenfalls ereignisbasiert ist die in [GBS+98] vorgestellte Architektur für verteilte Umgebungen zur Lösung wissenschaftlicher Probleme. Allerdings wird hier am Beispiel eines Systems zum Lösen linearer System vorgeschlagen, für jeden Problembereich eine besondere Problemlösungsumgebung mit einer bestimmten Menge von Komponenten und einer speziellen grafischen Benutzerschnittstelle bereit zu stellen, in welcher die Komponenten miteinander verknüpft werden können. Zur Unterstützung bei der Entwicklung dieser Umgebung steht ein Komponentensystem und eine verteilte Infrastruktur für die Verwaltung von Komponenten zur Verfügung. Damit ist dieser Ansatz ähnlich dem allgemeinen Ansatz von ECL, verlangt aber einen deutlich höheren Entwicklungsaufwand vom Sprach- und Komponentenentwickler.