2. Anforderungen an Koordinationssprachen für verteilte Applikationen

Die Schwierigkeiten bei der Entwicklung von Applikationen für heutige verteilte Systeme sind häufig eine Folge ihrer *Heterogenität* und *Dynamik*. Alle Subsysteme eines verteilten Systems, wie Prozessoren, Netzwerke, Betriebssysteme, Programmiersprachen und auch die eingesetzte Middleware selber können beliebig unterschiedlich sein. Im allgemeinen muss sich eine Applikation an die Elemente des verteilten Systems anpassen und kann diese nicht durch geeignetere ersetzen. Da sich ein solches verteiltes System potenziell auch über mehrere Verwaltungsbereiche erstrecken kann, können zusätzlich auch Politiken, z.B. bzgl. des Zugriffs auf die Rechenressourcen, in den einzelnen Subsystemen unterschiedlich sein.

Alle Systemkomponenten können prinzipiell von beliebigen Nutzern in nichtdeterministischer Reihenfolge verwendet werden, so dass die Leistungsgüten der zur Verfügung stehenden Resourcen starken Schwankungen unterworfen sind. Dies erschwert eine gute Lastverteilung, die für eine hohe Gesamtleistung einer verteilten Applikation notwendig ist. Hinzukommend ist die Fehlerwahrscheinlichkeit in verteilten Systemen im Vergleich zu lokalen Systemen hoch, so dass mit dem Ausfall von Komponenten gerechnet werden muss. Desweiteren können Ressourcen ohne vorherige Ankündigung zur Wartung aus dem System entfernt werden. Daraus resultiert die hohe Dynamik bzgl. Leistungswerte und auch des Vorhandenseins der einzelnen Systemkomponenten.

Der Einsatz von *Middleware* erzeugt eine Abstraktionsschicht für ein verteiltes System, indem es höhere Dienstgüten, wie z.B. eine zuverlässige Nachrichtenübertragung in einem unzuverlässigen Netzwerk, anbietet und den technischen Zugriff auf die System- und Applikationskomponenten vereinfacht, z.B. durch die Unterstützung entfernter Methodenaufrufe. Er erleichtert damit die Erstellung der einzelnen lokalen Komponenten einer verteilten Applikation. Um diese nun miteinander zu der Gesamtapplikation zu verbinden, lässt sich eine *Koordinationssprache* einsetzen. In dieser werden die einzelnen Interaktionen zwischen den lokalen Komponenten programmiert.

Die Sprachkonstrukte einer Koordinationssprache entscheiden darüber, wie gut sie für einen bestimmten Anwendungsbereich geeignet ist. Die Middleware NetSolve erlaubt es z.B., dass numerische Bibliotheken, die auf entfernten Rechnern vorhanden sind und auch dort ausgeführt werden, wie lokale Bibliotheken benutzbar sind [CD97]. Eine adäquate Koordinationssprache

könnte hier z.B. ein Fortran-System sein, welches Zugriff auf die Bibliotheksfunktionen erlaubt.

Möchte man dagegen verteilte Applikationen unter Einsatz von SOAP nach dem Klienten-Dienstanbieter Entwurfsmuster entwickeln, sieht eine geeignete Koordinationssprache anders aus. Eine Möglichkeit besteht darin, die verteilte Applikation als einen Graph darzustellen, dessen Knoten die Dienstanbieter und die Klienten darstellt und dessen Kanten die Interaktionen zwischen diesen beiden Knotenklassen repräsentiert. Der Entwickler muss den Graph entwerfen, der damit ein Programm in dieser Koordinationssprache darstellt. Das Entwicklungssystem kann dann anhand des Graphen die verteilte Applikation aufsetzen.

Wie schon in der Einleitung motiviert wurde, ist eine einzige Koordinationssprache für die Vielfalt der Zielplattformen und für die unterschiedlichen Anwendungsbereiche nicht ausreichend. Statt dessen wird hier der Ansatz eines modularen Rahmens für Koordinationssprachen verfolgt. Um einen derartigen Rahmen konzipieren zu können, muss trotz dieser Absage an eine allgemeine Koordinationssprache untersucht werden, ob es allgemeine Anforderungen gibt, die an Koordinationssprachen für verteilte Systeme zu stellen sind. Mit diesen Anforderungen kann ein erweiterbares Basisgerüst für die Menge aller zu unterstützenden Koordinationssprachen entworfen werden.

Dafür wird zunächst der Begriff der Koordination bestimmt. Daraufhin wird in Kapitel 2.2 eine Kategorisierung der Eigenschaften von Koordinationssprachen erstellt. Basierend auf diesen Kategorien wird eine Klasse von Koordinationssprachen spezifiziert, die sich für die Applikationsentwicklung in verteilten Systemen besonders gut eignet. Auf dieser Klasse aufbauend werden abschließend in Kapitel 2.3 Anforderungen an einen Rahmen für Koordinationssprachen gestellt.

2.1. Begriffsbestimmung von Koordination

Nach [MC94] läßt sich Koordination allgemein als die Verwaltung von Abhängigkeiten zwischen beliebigen Aktivitäten definieren. Wichtige Abhängigkeiten sind z.B:

- *Zeitliche Abhängigkeit*
  Die beiden möglichen zeitlichen Abhängigkeiten bestehen darin, dass eine Aktivität vor oder nach einer anderen Aktivität stattfinden muss. Sind zwei Aktivitäten zeitlich unabhängig voneinander, können sie nebenläufig durchgeführt werden.

- *Übertragungsabhängigkeit*
  Eine Übertragungsabhängigkeit besteht dann, wenn eine Aktivität für ihre Ausführung etwas von einer anderen Aktivität benötigt, wie z.B. Eingabedaten. Dieses Etwas muss transportiert werden und es muss die Kompatibilität der übertragenen Information mit den beteiligten Aktivitäten gewährleistet sein.

- *Verwaltungs-/Gruppierungsabhängigkeit*
  Ein Beispiel für eine derartige Gruppierungsabhängigkeit ist die Entscheidungsfindung in einer hierarchisch organisierten Organisation. Hierfür werden zuerst die Meinungen der Mitarbeiter in der untersten Hierarchiestufe von den jeweiligen Gruppenleitern eingeholt und bewertet. Die Gruppenleiter berichten wiederum ihren Vorgesetzten, bis schließlich die Firmenleitung eine Entscheidung fällt.

Diese Definition ist bewusst sehr allgemein gehalten. Sie soll als Rahmen dienen, in dem unterschiedliche Disziplinen (Informatik, Wirtschaftwissenschaft, Biologie, etc.) gemeinsam an einer Koordinationstheorie arbeiten können.

Einen theoretisch fundierten Ansatz speziell für die Informatik verfolgt [Arb98]. Hier wird eine Interaktionsmaschine definiert als eine um Ein-/Ausgabeoperationen erweiterte Turingmaschine. Koordination ist dann die dynamische Verwaltung der Topologien zwischen Interaktionsmaschinen und die Konstruktion von Protokollen zur Realisierung dieser Topologien.

Etwas pragmatischer definiert [GC92] ein Programmiermodell als bestehend aus einem Rechenmodell, welches die Spezifikation von einzelnen sequenziellen Rechenaktivitäten erlaubt, und einem Koordinationsmodell, das die einzelnen Rechenaktivitäten zu einem System verbindet. Eine Koordinationssprache enthält Operationen zur Erzeugung von Rechenaktivitäten und für die Kommunikation zwischen ihnen.

Im folgenden wird diese Definition für Koordinationsprachen verwendet, wobei es dahingehend erweitert wird, dass Rechenaktivitäten durchaus interne Parallelität aufweisen können. Dies ermöglicht es, komplexe Rechenaktivitäten zu benutzen, wie es in verteilten Applikationen üblich ist. Da eine Koordinationssprache konzeptionell nur eine eingeschränkte abstrakte Sicht auf die Rechenaktivitäten besitzt, ergeben sich durch diese Erweiterung keine Einschränkungen.

Ein Entwicklungssystem, das aus einer Koordinationssprache, Übersetzer und einem Laufzeitsystem besteht, welches das Starten von Programmen geschrieben in der Koordinationssprache erlaubt, wird im folgenden *Koordinationssystem* genannt.

2.2. Anforderung an ein Koordinationssystem

Eine Koordinationsprache für verteilte Applikationen dient der Steuerung der Interaktion der lokalen zentralisierten Applikationskomponenten, welche die Rechenaktivitäten realisieren. Sie dient nicht unbedingt der Implementierung der lokalen Applikationskomponenten. Um nun die Anforderungen zu untersuchen, die an eine Koordinationssprache für dieses Ziel zu stellen sind, werden im folgenden drei voneinander unabhängige Eigenschaften von Koordinationssprachen definiert und untersucht.

Einbettungsart

Bei Sprachen mit *endogener Einbettungsart* müssen spezielle Koordinationsoperationen in die Rechenaktivitäten eingefügt werden. Abbildung 2.1 zeigt ein

Beispiel für die Verwendung dieses Ansatzes in der Koordinationssprache Linda mit der Programmiersprache C, das [Sci00] entnommen und etwas vereinfacht wurde. Das Koordinationsmodell von Linda besteht aus einem Tupelraum, der von allen beteiligten Prozessen benutzt werden kann, um Tupel in den Tupelraum zu legen, Tupel aus diesem zu entfernen oder nichtverbrauchend zu lesen. Um durch Nebenläufigkeit verursachte Probleme zu vermeiden, ist jeder Zugriff auf den Tupelraum atomar.

Koordinatorcode:

```
void master()
{ ... // Variablendeklarationen
   out("assignment", myAssignment);// schreibe Gesamtaufgabe
   out("task", 0)              // schreibe erste Unteraufgabe
  for (i = 0; i < NWORKERS; i++) // erzeuge Arbeiter-Prozesse
    eval("worker",worker());

  worker();                     // werde selber zum Arbeiter

  for (task = 0; task <= NTASKS; task++)
    // Einsammeln der Ergebnisse
    in("result", task, ?res[task]);
  // Verarbeiten der Ergebnisse
  ...
}
```

Arbeitercode:

```
void worker()
{ ... // Variablendeklarationen
   rd("assignment", ?assignment);    // lese Gesamtaufgabe

  // bearbeite alle Aufgaben
  while (1)
  {
    in("task", ?task);    // lese verbrauchend Unteraufgabe
    out("task", task+1);  // erzeuge neue Unteraufgabe

    if (task > NTASKS)    // prüfe, ob noch was zu tun ist
      break;

    // führe die Berechnungen durch
    res = compute(assignment,task);
    out("result", task, res);      // schreibe das Ergebnis
  }
}
```

Abbildung 2.1.: Ein Beispiel endogener Einbettung mit der Koordinationssprache Linda und der Programmiersprache C

Das Beispiel besteht aus einer nebenläufigen Applikation, die nach dem Master-Worker Entwurfsmuster erstellt wurde. Ein Koordinatorprozess verteilt dabei Aufgaben an die Arbeiterprozesse. Der Code des Koordinators be-

steht aus der Prozedur `master()`. Die speziellen Koordinationsoperationen von Linda sind in der Abbildung fett gedruckt. Der Koordinator legt zuerst die Gesamtaufgabe mit der Operation `out()` als Tupel in den Tupelraum, so dass jeder Arbeiterprozess darauf zugreifen kann.

Ein Tupel kann prinzipiell beliebig viele Einträge beliebigen Typs haben. Mit auf Mustern basierenden Anfragen können interessierte Prozesse den Tupelraum durchsuchen. In diesem Beispiel sind alle Tupel sehr einfach aufgebaut und bestehen nur aus Namen-Wert-Paaren.

Zusätzlich zu der Gesamtaufgabe erzeugt der Koordinator die erste Unteraufgabe, die durch eine natürliche Zahl dargestellt wird. Auch diese legt er als Tupel in dem Tupelraum ab. Danach erzeugt er mit der `eval()`-Operation Arbeiterprozesse in der benötigten Anzahl. Diese werden sofort gestartet.

Daraufhin führt er selber den Code eines Arbeiterprozesses aus. Wenn er die Prozedur `worker()` beendet hat, ist sichergestellt, dass alle Unteraufgaben berechnet wurden. Er sammelt daher mit der `in()`-Operation alle Ergebnisse ein. Ein Ergebnis ist dabei ein zweistelliges Tupel, dessen erster Eintrag eine Zeichenkette `"result"` ist. Die Ergebnisse werden dann weiter verarbeitet.

Ein Arbeiterprozess führt nur die Prozedur `worker()` aus. Zuerst liest er die Gesamtaufgabe mit der nichtverbrauchenden Leseoperation `rd()` und speichert sie in der Variable `assignment`. Dann bearbeitet er alle Unteraufgaben, die er in dem Tupelraum findet, bis die Gesamtaufgabe gelöst ist. Hierfür liest er zunächst verbrauchend die aktuelle Unteraufgabe aus dem Tupelraum. Dann berechnet er die nächste Unteraufgabe durch Erhöhung um Eins und legt diese in den Tupelraum.

Da jede Operation auf dem Tupelraum atomar ist, ist sicher gestellt, dass sich zu jedem Zeitpunkt höchstens ein Tupel mit einer Unteraufgabe in dem Tupelraum befinden kann. Wenn andere Arbeiterprozesse nach einer Unteraufgabe suchen, blockieren sie in der `in()`-Operation so lange, bis sie Zugriff auf diesen einen Tupel erlangen.

Ist diese Unteraufgabe gültig, wird sie berechnet und das Ergebnis im Tupelraum abgelegt. Anderenfalls terminiert die Prozedur und damit der Arbeiter-Prozess.

Es ist damit offensichtlich, dass eine endogene Einbettung eine invasive Anpassung des Codes für die Rechenaktivitäten erforderlich macht. Unter der Voraussetzung, dass bestehende heterogene Softwarekomponenten mit minimalem Änderungsaufwand als Rechenaktivitäten einsetzbar sind, sind endogene Koordinationssprachen für die hier betrachtete Applikationsklasse nicht geeignet.

Bei *exogener Einbettung* stehen Koordinationsoperationen zur Verfügung, um Rechenaktivitäten, die nicht in der Koordinationssprache selber programmiert wurden, über feste Schnittstellen von außen zu kontrollieren. Abbildung 2.2 zeigt eine ebenfalls auf dem Master-Worker Entwurfsmuster basierende Applikation, die für das Paralex-System entworfen wurde[1], das einen

[1] Da das Paralex-System nicht konkret zur Verfügung stand, wurde dieses Beispiel anhand der veröffentlichten Beschreibungen erstellt. Die grafische Darstellung ist eng an die grafische Darstellung des Paralex-Editors angelehnt.

exogenen Einbettungsansatz verfolgt [DGB+96].

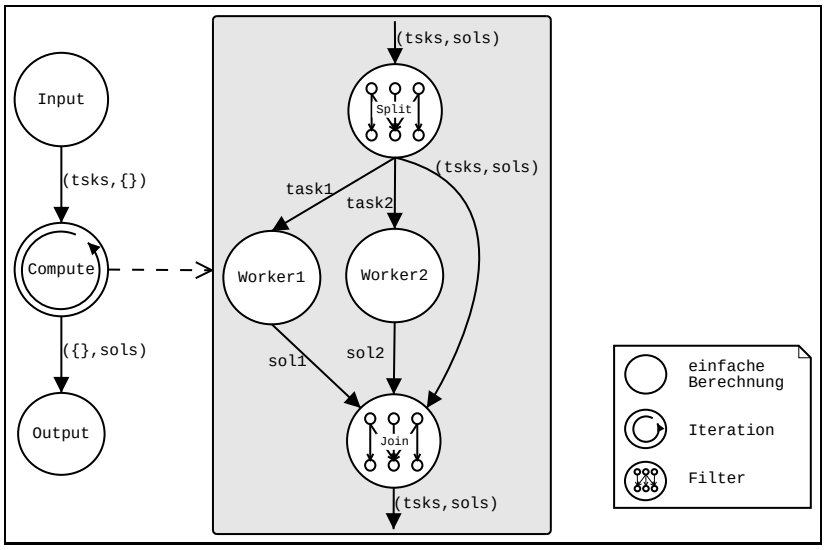

Abbildung 2.2.: Ein Beispiel exogener Einbettung mit Paralex in freier grafischer Notation

Eine Paralex-Applikation wird durch eine statische Datenflussarchitektur beschrieben. Die Knoten stellen Berechnungen auf Daten dar, während die Kanten Daten übertragen. Ein Knoten wird aktiviert, wenn an all seinen Eingabekanten Daten vorliegen. Das Ergebnis seiner Berechnungen wird über Ausgabekanten an die nächsten Knoten weitergegeben. Es gibt vier unterschiedliche Knotentypen:

- *Berechnung*
  Berechnungsknoten steuern die externen Rechenaktivitäten. Ein Berechnungsknoten wird über das Setzen von Eigenschaften parametrisiert. Mit der Eigenschaft `File` wird eine Datei spezifiziert, die entweder Quellcode oder ausführbaren Code für die Rechenaktivität enthält. Wird ein Berechnungsknoten aktiviert, wird mit dieser Datei ein Rechenprozess erzeugt. Dieser wird mit den Eingabedaten versorgt, die Rechenaktivität wird gestartet, die Ergebnisse werden abgegriffen und schließlich an die nachfolgenden Knoten geschickt. Da ein Entwurfsziel von Paralex die Ausführung rechenintensiver Applikationen auf Workstationcluster war, lässt sich mit der Eigenschaft `Obligatory` eine Klasse von Workstations spezifizieren, die für die Ausführung der Rechenaktivität geeignet ist. Es werden auch noch weitere Eigenschaften unterstützt, auf die hier im folgenden aber nicht weiter eingegangen wird.

- *Filter*
  Filterknoten ermöglichen es, sowohl eingehende strukturierte Daten zu zerlegen und über verschiedene Ausgabekanten zu schicken, als auch über verschiedene Eingabekanten eingehende Daten miteinander zu kombinieren.

- *Subgraph*
  Ein Subgraph-Knoten enthält eine eigene statische Datenflussarchitektur. Bei der Aktivierung wird konzeptionell der Subgraph-Knoten durch die interne Architektur ersetzt. Dies ermöglicht eine hierarchische Strukturierung einer Applikation.

- *Iteration*
  Alle obigen drei Knotentypen kann man zusätzlich als Iterationsknoten deklarieren. Sie werden dann solange hintereinander aktiviert, bis eine gegebene Bedingung erfüllt ist. Dabei werden bei einer Iteration die Ergebnisdaten der vorherigen Iteration als Eingabedaten benutzt.

In dem Beispiel aus der Abbildung lädt der Knoten `Input` zuerst die Eingabedaten. Diese schickt er als Paar an den Subgraph-Knoten `Compute`. Der Einsatz eines Paares als Datenstruktur erlaubt die Iteration des `Compute`-Knotens. Das erste Element des Paares sind die noch zu berechnenden Aufgaben, während das zweite Element die schon berechneten Lösungen enthält. Die Iteration terminiert, wenn keine Aufgaben mehr übrig sind. Die Ergebnisse werden dann von dem `Output`-Knoten ausgegeben.

Die Berechnung in dem `Compute`-Knoten besteht zunächst aus einem Filterknoten `Split`, der aus den noch offenen Aufgaben zwei wählt und diese zu zwei Berechnungsknoten schickt. Ist nur noch eine Aufgabe übrig, wird diese an beide Berechnungsknoten geschickt. Die übrig gebliebenen Aufgaben zusammen mit den schon berechneten Lösungen schickt er an den Filterknoten `Join`.

Die beiden Berechnungsknoten `Worker1` und `Worker2` führen nun die eigentliche Berechnung durch und schicken das Ergebnis ebenfalls an den `Join`-Knoten. Dieser fügt dann die Ergebnisse mit den bisherigen und den noch offenen Aufgaben zu dem nächsten Iterationszustand zusammen.

Wie an diesem Beispiel zu erkennen ist, ermöglicht der exogene Einbettungsansatz von Paralex den Einsatz unterschiedlicher Programmiersprachen für die Implementierung der Rechenaktivitäten und für die Koordination. Für die letztere wird in Paralex eine intuitiv verständliche grafische Notation benutzt, während für die Implementierung der aufwendigen Rechenaktivitäten die Programmiersprache C verwendet wird.

Allgemein betrachtet, erlaubt die exogene Einbettung dem Entwickler verteilter Applikationen sich für die Implementierung der einzelnen Applikationskomponenten die jeweils geeignete Programmiersprache zu wählen. Vorhandene Altsoftware kann im Gegensatz zur endogenen Einbettung mit Umhüllungstechniken in eine Applikation integriert werden. Daher sollte ein allgemeines Koordinationssystem auf einer exogenen Einbettung basieren.

Steuerungsparadigma

Eine Koordinationssprache ist häufig für ein bestimmtes Steuerungsparadigma, wie z.B. das Datenflussparadigma oder das Paradigma der aktiven, kommunizierenden Objekte, entworfen worden. Für die Erstellung verteilter Applikationen existieren erfolgreiche Systeme, die auf unterschiedlichsten Paradigmen basieren, so dass sich keine Aussage treffen läßt, welche Paradigmen

sich besonders eignen bzw. ungeeignet sind. Für eine allgemeine Koordinationssprache ist daher zu fordern, dass sie bzgl. des Steuerungsparadigmen nicht festgelegt ist. Es sollten alle Paradigmen in ihr benutzt werden können.

Aus dieser Forderung folgt, dass Koordinationssprachen, die auf vollwertigen Programmiersprachen beruhen, i.a. nicht geeignet sind. Es gibt drei Klassen von Programmiersprachen: imperative, funktionale und regelorientierte Programmiersprachen. Jede Sprache ist i.a. einer Klasse klar zugeordnet und damit bzgl. ihrer Steuerungsparadigmen festgelegt. Auch wenn man die typischen Sprachkonstrukte der anderen Sprachklassen integrieren kann, so wirken diese meist aufgesetzt und bieten häufig nicht die aus ihrer Klasse gewohnte Mächtigkeit. So lässt sich die Übergabe von Funktionen in objektorientierten imperativen Sprachen über Schnittstellen simulieren. Man erhält aber nicht die Vielfalt an Möglichkeiten, die man in funktionalen Sprachen besitzt.

Eine Alternative zu Programmiersprachen stellen Architekturbeschreibungssprachen dar. Diese beschränken sich auf eine grobe Spezifikation der Komponenten einer Applikation und deren Interaktionen in Form von sogenannten Konnektoren. Eine umfassendere Einführung in Architekturbeschreibungssprachen wird in Kapitel 3.2 gegeben.

Abbildung 2.3-I zeigt als Beispiel eine einfache Applikation, in der zwei Programme als Komponenten über eine Unix-Pipe als Konnektor miteinander verbunden werden. Der Graph mit seinen Annotationen enthält für ein Koordinationssystem genügend Informationen, um das System aufzusetzen.

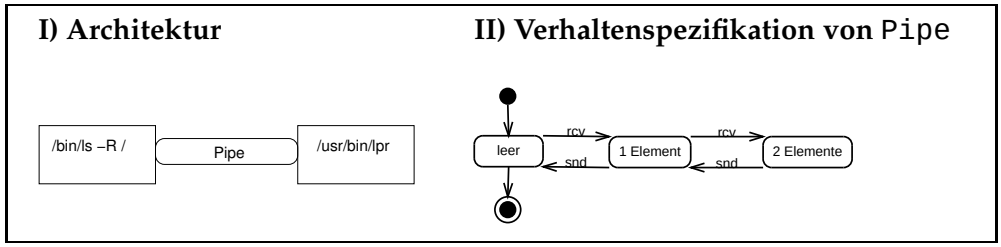

Abbildung 2.3.: Einfaches Beispiel des Einsatz einer Architekturbeschreibungssprache als Koordinationssprache

Einige Architekturbeschreibungssprachen fordern keine Spezifikation der Semantik von Komponenten und Konnektoren. Andere fordern nur eine Teilspezifikation der Semantik, die dann Verhalten genannt wird. In Abbildung 2.3-II ist als Beispiel das Verhalten des Unix-Pipe Konnektors als Zustandsautomat gezeigt, unter der Annahme, dass die Pipe einen internen Puffer der Größe zwei hat. Obwohl das Sende- und Empfangsverhalten der Pipe genau spezifiziert ist, wird über den Inhalt der Daten nichts gesagt.

Auch wenn die Architektursprache keine genaue Spezifikation der Semantik fordert, muss eine konkrete Koordinationssprache, die auf einer Architektursprache aufsetzt, die dem Entwickler zur Verfügung gestellten Komponenten und Konnektoren genau spezifizieren. Für jede Zielplattform und für jeden Anwendungsbereich lassen sich aber spezifische Komponenten- und Konnektorentypen entwickeln.

Erzeugen von Rechenaktivitäten

Eine Programm in einer Koordinationssprache verbindet Rechenaktivitäten zu einer Applikation. Die Rechenaktivitäten selber werden nicht in der Koordinationssprache definiert, sondern liegen z.B. als Code in einer Datei vor oder sind über entfernte Dienstanbieter verfügbar. In einem Programm wird jede Rechenaktivität durch eine Rechenkomponente vertreten, welche alle nötigen Referenzen und Parameter enthält, um die Rechenaktivität zu erzeugen.

Die Bindung einer Rechenkomponente an eine konkrete Rechenaktivität geschieht in zwei Schritten. Zuerst muss für die Rechenkomponente geeigneter ausführbarer Code lokalisiert oder ggf. erzeugt werden. Dann müssen verfügbare Rechenressourcen gefunden werden, mit denen der Code dann ausgeführt wird. Dieser zweite Schritt muss in einer dynamischen Umgebung, in der weder die Existenz von Rechenressourcen noch deren Auslastung zur Übersetzungszeit bekannt ist, zur Laufzeit geschehen.

Der erste Schritt kann zur Übersetzungszeit stattfinden, wenn erstens der Code verfügbar ist und zweitens auch die Architektur der einzusetzenden Rechenressource bekannt ist. Die zweite Bedingung ist in einem allgemeinen heterogenen Umfeld nicht immer gegeben. Es ist sogar möglich, dass neue Rechnerarchitekturen zur Laufzeit vorhanden sind, die zur Übersetzungszeit gar nicht bekannt waren.

Gegen eine dynamische Bindung von Rechenkomponenten spricht der hohe Laufzeitaufwand. Eine verteilte Infrastruktur muss die vorhandenen Rechenresourcen mit ihren aktuellen Leistungswerten beobachten und verwalten. In einem heterogenen System muss Code für die einzelnen Rechnerarchitekturen dann erzeugt und gebunden werden, wenn dieser benötigt wird.

Nicht für jedes verteilte System ist dieser Aufwand notwendig. Betrachtet man z.B. eine Middleware bestehend aus EJB-Servern[2], dann wird das Problem der Heterogenität durch den Einsatz von Java gelöst. Die einzelnen Rechenkomponenten werden statisch auf die EJB-Server verteilt, welche dann evtl. eine automatische dynamische Lastverteilung vornehmen. In diesem Fall muss das Koordinationssystem beim Starten die einzelnen Rechenkomponenten auf die EJB-Server verteilen und kann dann terminieren. Es ist kein weiterer Aufwand zur Laufzeit nötig.

Zusammengefasst ist zu sagen, dass ein allgemeines Koordinationssystem eine dynamische Bindung einer Rechenkomponente an Code unterstützen sollte, auch wenn dies nicht für alle Anwendungsbereiche notwendig ist. Die Zuordnung von Ressourcen, wie z.B. die Verteilung auf die Applikationsserver im obigen Beispiel, sollte allerdings stets dynamisch sein.

Zusammenfassung der Anforderungen

Fasst man diese Überlegungen zusammen, erhält man eine Liste von Anforderungen an Koordinationssprachen für Anwendungen in verteilten dynamischen heterogenen Systemen:

1. Die Koordinationssprache muss exogene Einbettung unterstützen.

[2]Enterprise Java Beans [Sun01]

2. Sie darf sich nicht auf ein bestimmtes Steuerungsparadigma beschränken. Der Einsatz einer Architekturbeschreibungssprache ist sinnvoll und empfehlenswert.

3. Sie kann eine dynamische Bindung einer Rechenkomponente an Code unterstützen. Sie muss aber eine dynamische Bindung von Code an Rechenressourcen gewährleisten.

2.3. Anforderungen an einen Rahmen für Koordinationssprachen

Die in dem vorangehenden Kapitel gestellten Anforderungen gelten für eine große Klasse von Koordinationssystemen. Ein geeigneter Rahmen muss die Erstellung eines bestimmten Koordinationssystems, das sich für eine bestimmte Middleware und für einen bestimmten Anwendungsbereich eignet, möglichst einfach gestalten. Die Erstellung eines solchen Systems geschieht, indem der Rahmen mit geeigneten Erweiterungen versehen wird. Die Kombination aus Rahmen und Erweiterungen ergibt dann das Koordinationssystem. Diese Erweiterungen lassen sich in drei Klassen unterscheiden:

- *Integration von Rechenkomponenten*
  Rechenkomponenten dienen zur Spezifikation von Rechenaktivitäten eines Programmes. Da es prinzipiell keine Beschränkung bzgl. der Art der Rechenaktivitäten gibt, muss der Rahmen leicht um verschiedene Arten erweitert werden können.

- *Integration von Steuerungsparadigmen*
  Da kein spezielles Steuerungsparadigma sich für alle Anwendungsbereiche eignet, muss der Rahmen mit den benötigten Paradigmen erweitert werden können. Sofern notwendig müssen die einzelnen Paradigmen auch miteinander interagieren können, so dass in einer Applikation verschiedene Paradigmen kombiniert einsetzbar sind.

- *Anbindung von Zielsystemen*
  Die Unterstützung eines Zielsystemes und damit die Anbindung an eine bestimmte Middleware, ist ebenfalls eine Erweiterungsklasse. Auch hier sollte es möglich sein, dass eine Applikation unterschiedliche Zielsysteme gleichzeitig benutzen kann.

Wie im vorherigen Kapitel diskutiert, eignen sich Architekturbeschreibungssprachen als Grundlage einer Koordinationssprache. In diesem Fall bietet es sich an, Komponententypen als Repräsentation der einzelnen Rechenkomponenten und Konnektortypen als Repräsentation der Steuerungsparadigmen einzusetzen. Zusätzlich ermöglichen es Architekturbeschreibungssprachen, große Systeme hierarchisch zu strukturieren, indem Subsysteme aus Komponenten und Konnektoren zu einzelnen Komponenten oder Konnektoren zusammengefasst werden.

Mit der obigen Klassifikation lassen sich die Aktoren, die mit einem Rahmen für Koordinationssysteme interagieren, in vier Klassen einteilen, die nicht disjunkt sein müssen:

1. *Komponentenentwickler* entwickeln Rechenkomponenten, die dann mit der Koordinationssprache zu einer verteilten Applikation zusammengefasst werden. Sie implementieren die Rechenaktivität, die von der Rechenkomponente dargestellt wird.

2. *Spracherweiterer* verfeinern bestehende Komponenten- und Konnektorentypen und erweitern die Koordinationssprache um neue Steuerungsparadigmen durch die Erstellung neuer Typen. Sie erweitern damit das System um die Unterstützung neuer Applikationsbereiche oder neuer Zielplattformen.

3. *Applikationsentwickler* setzen ihre Applikation aus vorbereiteten Rechenkomponenten und Sprachkonstrukten zusammen. Stellt das System nicht alle zur Implementierung der Applikation notwendigen Elemente zur Verfügung, kann ein Applikationsentwickler die Rolle eines Komponentenentwicklers oder eines Spracherweiterers annehmen, um diese Defizite zu beheben.

4. *Anwender* führen Applikationen aus, um ihre jeweiligen Probleme zu lösen.

Der Rahmen muss alle Benutzerklassen unterstützen. Das bedeutet insbesondere, dass er für jede Aufgabe geeignete Schnittstellen und Werkzeuge bereitstellen muss.