3. Stand der Forschung

In diesem Kapitel wird versucht, einen Überblick über die Koordinationssprachen zu geben, die sich zur Erstellung von weit verteilten Applikationen eignen. Für einen allgemeinen Überblick über Koordinationssprachen wird auf [PA98] verwiesen. In Kapitel 2.2 wurde anhand einer Kategorisierung eine Klasse von Koordinationssprachen definiert, welche die Anforderungen für die Entwicklung verteilter Applikationen erfüllt. Da es in dieser Klasse viele konkrete Systeme gibt, wird im folgenden eine weitere orthogonale Klassifikation benutzt, um diesen Überblick überschaubar zu gestalten. Diese Klassifikation orientiert sich an der syntaktischen Struktur der Sprache.

3.1. Sprachbasierte Koordinationssprachen

In sprachbasierten Koordinationssprachen wird eine vollwertige Programmiersprache zur Erstellung des Programms eingesetzt, die auf externe Komponenten zugreifen kann. Eine derartige Programmiersprache wird dann häufig als *Skriptsprache* oder auch als *Konfigurationssprache* bezeichnet.

Der externe Zugriff läßt sich mit zwei Ansätzen realisieren. Eine Möglichkeit besteht in der Nutzung von Middleware, wie z.B. CORBA [BVD01] oder DCOM [EE98]. So lässt sich die Sprache Python [pyt02] sowohl mit CORBA [fno02] als auch mit COM [act02] einsetzen. Ähnliches gilt für Perl [per02].

Eine leichte Variation besteht in der Neuentwicklung einer Sprache basierend auf bewährten Sprachkonstrukten mit direkter Unterstützung einer bestimmten Middleware. So kombiniert beispielsweise die Sprache Mondrian Haskell mit DCOM [mon02] und die Sprache CorbaScript Python mit CORBA [cor02b] während in [Adl95] TCL um Client/Server Fähigkeiten erweitert wurde.

Opus [LHMZ00, CHM+97] erweitert High-Performance-Fortran um Module und task-parallele Konstrukte. So lässt sich eine verteilte Applikation durch Komposition von vorhandenen Fortran-Komponenten erstellen. Dieser Ansatz konzentriert sich damit stark auf das Gebiet der rechenintensiven numerischen Applikationen.

Die Programmiersprache Lua wurde als erweiterbare imperative Konfigurationssprache konzipiert [IFC96]. Sie bietet einen kleinen Kern, der sich mit externen Bibliotheken und über sogenannte Fallback-Funktionen erweitern lässt. Diese sind in Lua definierte Funktionen, die von dem Laufzeitsystem in spezifizierbaren Situationen, wie. z.B. bei Arithmetik- oder Programmfehler, aufgerufen werden. Ein Beispiel für die Anwendung von Lua im Kontext verteilter Systeme ist LuaSpace [BR00], welches die Erstellung verteilter Applikation

durch Konfiguration von CORBA-Objekten ermöglicht.

Der zweite Ansatz zum Zugriff auf externe Komponenten besteht darin, eine Sprache einzusetzen, die es sowohl ermöglicht, verteilte Applikationen zu entwickeln, als auch lokale heterogene Komponenten zu umhüllen. Ein Beispiel hier ist Java, das mit Java-RMI und JNI die geforderten technologischen Eigenschaften besitzt. Beide Ansätze lassen sich auch kombinieren.

Ein Nachteil beim Einsatz einer allgemeinen Programmiersprache als Koordinationssprache besteht in dem im Allgemeinen recht hohen Einarbeitungsaufwand. Der Entwickler muss sich sowohl in die Programmiersprache als auch in die Einbettung der unterstützten Middleware einarbeiten. Setzt der Entwickler die Programmiersprache ebenfalls für die Implementierung der Rechenaktivitäten ein, reduziert sich der Aufwand deutlich. Allerdings besteht dann die Gefahr, dass die Trennung zwischen Rechenaktivitäten und Koordinierung aufgehoben wird.

Zusätzlich besteht bei Einsatz einer Programmiersprache die Gefahr, dass sich die Struktur der Applikation nicht in der Struktur ihres Programmcodes wiederspiegelt. Es ist zwar auch in Programmiersprachen möglich, Strukturierungskonstrukte, wie Module, Funktionen etc. einzusetzen; dies erfordert aber Disziplin seitens des Entwicklers. Der Einsatz einer Architekturbeschreibungssprache, die sich auf die Darstellung der Struktur konzentriert, bietet sich hier als bessere Alternative an.

Desweiteren sind prozedurale Sprachen häufig seiteneffektbehaftet und damit schwierig zu analysieren. Dies führt zu einer erhöhten Fehlerwahrscheinlichkeit und erschwert die Wiederverwendung aufgrund fehlender Werkzeuge für automatische oder halbautomatische Korrektheitsanalyse von Kompositionen. Der Einsatz von funktionalen Sprachen bietet sich hier an, allerdings leiden diese noch immer unter geringer Akzeptanz.

3.2. Architekturbasierte Koordinationssprachen

Diese Klasse umfasst Koordinationssprachen, in denen eine Applikation in Form eines Graphs erstellt wird. Die Koordinationssprache selber kann dabei sowohl textbasiert als auch grafisch sein. Eine scharfe Differenzierung zwischen sprachbasierten und architekturbasierten Koordinationssprachen ist allerdings prinzipiell nicht möglich, da man jede Sprache auch grafisch darstellen kann. So kann man beispielsweise Nassi-Shneidermanndiagramme [DIN11] benutzen, um prozedurale Algorithmen zu spezifizieren.

In diese Kategorie der architekturbasierten Koordinationssprachen fallen viele Sprachen zur Implementierung verteilter Applikationen, da es naheliegend ist, verteilte Komponenten einer lose gekoppelten Applikation als aktive Objekte zu modellieren, die über Kanäle Informationen austauschen. Die Komponenten bilden dann Knoten in dem Graph der Applikation, während die Kanäle die Verbindungen zwischen den Knoten darstellen. Da solche Verbindungen i.a. parametrisierbar und von unterschiedlichem Typ sein können, werden sie im Folgenden als Konnektoren bezeichnet, im Gegensatz zu einfachen Kanten eines Graphs.

Der Graph der Applikation wird im folgenden als die Architektur der Applikation bezeichnet. Üblicherweise werden die Komponenten benutzt, um die Rechenaktivitäten einer Applikation zu beschreiben. Neben einer Beschreibung der Rechenaktivität an sich, enthält eine Komponente i.a. auch die Beschreibung einer Schnittstelle, mit der die Rechenaktivität zu nutzen oder zu steuern ist. Die Konnektoren dienen dagegen der Koordination der Komponenten. Im allgemeinen orientieren sich Konnektoren an einfachen Koordinationsparadigmen, wie z.B. das Senden und Empfangen von Ereignissen oder einem synchronen Methodenaufruf.

Um eine Diskussion dieser Sprachen zu ermöglichen, werden hier weitere grundlegende Begriffe aus dem Bereich der Softwarearchitekturforschung eingeführt [SG96]:

- Eine *Software-Architektur* beschreibt die einzelnen Elemente eines Software-Systems und wie sie interagieren. Hierbei wird das System i.a. so grobkörnig unterteilt, dass dessen Elemente noch weiterer Spezifikation bedürfen, um das Gesamtsystem genau zu verstehen. Zur Spezifikation der Elemente werden häufig andere Ansätze als Architektursprachen eingesetzt.

- Ein *Architekturstil* beschreibt eine Menge von Architekturen mit gemeinsamen Merkmalen. Beispiele sind hier Client-Server oder Datenflussarchitekturen. Architekturen eines Architekturstiles lassen sich häufig aus den gleichen, i.a. parametrisierbaren Grundelementen zusammensetzen. Diese Bausteine bilden das *Vokabular* des Architekturstils.

- Eine *Architekturbeschreibungssprache* erlaubt es, Architekturen formal zu spezifizieren. Häufig bieten Architekturbeschreibungssprachen (auf Englisch: Architecture Description Language, ADL) auch die Möglichkeit zur Spezifikation von Vokabularen. Moderne ADLs bieten hierfür auch eine objektorientierte Unterstützung wie z.B. Vererbung oder den Einsatz logischer Prädikate zur Spezifikation gültiger Architekturstrukturen an.

Häufig wird zwischen dem *abstrakten Verhalten* und der Semantik einer Applikation unterschieden (z.B. in [BSP99, Luc96]). Dabei ist mit dem abstrakten Verhalten die Abstraktion der Semantik auf die grobe Interaktion zwischen den einzelnen Elementen unter Vernachlässigung der konkreten Ein- und Ausgabedaten gemeint. Ein verbreiteter Ansatz besteht z.B. in der Verhaltensmodellierung der einzelnen Architekturelemente mit Zustandsautomaten, die dann über Ereigniskanäle miteinander verbunden werden [AG94, LV95] oder direkt miteinander komponiert werden [GSB02]. Die Ereignisse tragen dabei keine Daten, die bei einem konkreten Programmablauf zwischen den Elementen ausgetauscht werden. Dieser formale Ansatz ermöglicht es, eine Applikation bzgl. formal definierbarer Eigenschaften wie z.B. Konsistenz zu überprüfen [AHT02].

Im folgenden werden einige ausgewählte Arbeiten vorgestellt, die eine architekturbasierte Entwicklung verteilter Applikationen ermöglichen. Für

einen allgemeinen Überblick und eine Klassifikation von Architekturbeschreibungssprachen siehe [MT00].

Eingeschränktes Vokabular

Einige architekturbasierte Sprachen unterstützen nur ein bestimmtes Interaktionsparadigma, also einen festen Satz kaum parametrisierbarer Konnektortypen. Der Data Explorer, der zwar zur Visualisierung wissenschaftlicher Datensätze entworfen wurde, dessen Programmiersprache inzwischen aber auch als allgemeine Koordinationssprache eingesetzt wird, basiert z.B. vollständig auf dem Datenflussparadigma [Res].

Triana [tri02] basiert auf Grid OCL [TS98] und unterstützt Komponenten, die Java Klassen kapseln, und einfache Konnektoren, die auf Datenfluss basieren. Es bietet kein Typsystem bzgl. der Architekturelemente, ermöglicht es allerdings, entfernte Komponenten zu einer verteilten Applikation zu kombinieren.

Das System OCoN (Object Coordination Net) ermöglicht die visuelle Spezifikation, Implementierung und Simulation des abstrakten Verhaltens von verteilten Systemen [GW01]. Das verteilte System wird mit UML-Elementen als hierarchisches statisches Objektdiagramm beschrieben. Das abstrakte Verhalten des Systems beschreibt nun mit Elementen aus dem Bereich der Petrinetze, welche Methoden unter welchen Umständen aufgerufen werden.

PCL (Proteus Configuration Language) unterstützt Konnektoren, die Nachrichten synchron oder asynchron verschicken [pro02]. Komponenten kapseln Berechnungen und es ist spezifizierbar, ob sie einen eigenen Kontrollfaden besitzen und damit aktiv sind, oder ob sie als passive Komponenten nur aufgerufen werden können. Neue Komponenten können zwar durch Vererbung erzeugt werden, allerdings ist es nicht möglich, neue Paradigmen zu integrieren oder die vorhandenen Parametrisierungsmöglichkeiten zu erweitern.

Manifold basiert auf dem Ereignis-Paradigma [Arb98, PA00]. Komponenten kapseln Prozesse, die über Ereignisse gesteuert werden. Als Konnektortyp stehen Ereignisströme zur Verfügung, die Komponenten miteinander verbinden. Eine Komponente besitzt jeweils einen Koordinationsprozess und einen Arbeitsprozess. Der Koordinationsprozess wird über einen Zustandsautomaten spezifiziert, der eingehende Ereignisse verarbeitet, Ereignisse produziert und sendet und Kommandos an den assoziierten Arbeitsprozess absetzt. Dies ermöglicht eine formale und analysierbare Beschreibung des abstrakten Verhaltens einer verteilten Applikation bei gleichzeitiger Möglichkeit beliebige Softwarekomponenten als Arbeiterprozesse zu integrieren. Eine Erweiterung des Systems bzgl. der Konnektortypen ist aber nicht vorgesehen.

Sieht man Objekte von Java-Klassen als Komponenten und die Verknüpfung von Objekten über das Observer-Muster als Konnektor an, dann lässt sich auch JavaBeans [Mic97] als architekturbasierte Koordinationssprache mit beschränktem Vokabular ansehen. Symphony ist ein System, das auf JavaBeans aufsetzt [LK02]. Es besteht hauptsächlich aus einer Sammlung spezieller JavaBeans, die es ermöglichen, auf das Globus-System zuzugreifen, einer Infrastruktur für verteiltes Rechnen [FK97].

Die Idee, ein architekturbasiertes Entwicklungssystem auf der Grundlage ei-

nes Komponentenmodells zu realisieren, wurde in VPCE [SRW+00, WLR+00] aufgegriffen. Das System VPCE (Visual Programming Composition Environment) stellt ein Komponentensystem und eine Infrastruktur zur Verfügung, um sogenannte Problemlösungsapplikationen zu entwickeln. In diesen kann der Anwender Komponenten auswählen und miteinander verknüpfen.

Erweiterbares Vokabular

Etliche Architekturbeschreibungssprachen unterstützen die Definition neuer Architekturstile. In Acme [GMW97] sind Architekturstile Mengen von Architekturelementtypen, die erzeugt und zu konkreten Architekturen verknüpft werden können. Neue Typen lassen sich durch Spezialisierung leicht erstellen. Armani [Mon01] erweitert Acme um die Möglichkeit, Invarianten für gültige Architekturstrukturen mittels logischer Prädikate zu erstellen. Beide Sprachen beschränken sich allerdings auf die Struktur also die Syntax von Architekturen und bieten keine Möglichkeit, um Aussagen bzgl. der Semantik oder auch nur des abstrakten Verhaltens zu treffen. Sie sind als Komponente eines größeren Modellierungs- oder Entwicklungssystems angelegt, welches dann die Erstellung oder Analyse einer Applikation ermöglicht.

UniCon unterstützt eine feste Menge an Komponenten- und Konnektortypen, wobei neue Konnektoren vom Benutzer auch hierarchisch zusammengesetzt werden können [SDK+95]. Applikationen können als Softwarearchitekturen spezifiziert und zu einem ausführbaren Programm übersetzt werden. Dabei können allerdings unterschiedliche Architekturstile nicht gemischt werden.

In der Sprache Darwin wird das abstrakte Verhalten von Komponenten mit dem π-Kalkül spezifiziert, das für die Modellierung nebenläufiger verteilter Systeme entwickelt wurde [MDE95]. Im π-Kalkül besteht ein System aus unabhängigen Prozessen, die über Nachrichtenkanäle miteinander kommunizieren [MPW92]. Darwin enthält zusätzlich eine Makrosprache für eine parametrisierbare Konstruktion von Architekturen und die Möglichkeit, Abhängigkeiten zwischen den Komponenten zu beschreiben.

In [MTK97] wird ein einfacher Architekturstil vorgestellt, der es ermöglicht, verteilte Applikationen als hierarchische Architektur zu beschreiben. Aus dieser Architekturbeschreibung lassen sich dann die benötigten Schnittstellen in CORBA-IDL generieren.

Regis setzt auf Darwin auf [MDK94, KM85] und beschränkt sich auf Konnektoren, die typisierte Ereignisse übertragen. Neue Komponenten können durch Komposition erzeugt werden. Implementierungen der einzelnen Komponenten müssen als C++ Klassen vorliegen und Schnittstellen bereitstellen, die bestimmten Namenskonventionen unterliegen. Regis unterstützt sowohl verteilte Applikationen als auch die dynamische Konfiguration von Applikationen.

Mit ZCL lassen sich Konfigurationen und Rekonfigurationen von CORBA Objekten durch eine Architektur beschreiben [PB02]. Diese werden nach Lua übersetzt, einer imperativen Konfigurationssprache [IFC96]. Das Verhalten der einzelnen Komponenten wird in Z spezifiziert und ermöglicht so, Aussagen über die gesamte Architektur formal zu verifizieren. Eine Erweiterung ist nur

durch rekursive Konstruktion von Komponenten möglich.

Olan verfolgt einen komponentenorientierten Ansatz mit dem Schwerpunkt, bestehende Software zu integrieren [BBB+98]. Diese Softwarekomponenten werden mit einer Schnittstellenbeschreibungssprache spezifiziert, die neben der Typsignatur der einzelnen Methoden auch eine Verhaltensbeschreibung und weitere Metadaten enthält, wie z.B. den Ort des Codes. Aus den Schnittstellenbeschreibungen lassen sich dann automatisch für verschiedene Zielsprachen Stubs in Python generieren, die evtl. noch manuell angepasst werden müssen. Olan bietet eine feste Menge von Konnektoren an, die auf Sende- bzw. Empfängerobjekte in Python abgebildet werden, welche unterschiedliche Middleware-Systeme einsetzen. Neue Komponenten und Konnektoren lassen sich mittels rekursiver Konstruktion erstellen. Eine in Olan spezifizierte Applikation läßt sich unter Einsatz der Metainformationen in den Komponenten automatisch verteilt installieren, konfigurieren und starten.

Die Koordinationsprache Aster [IBRZ00, IBS98] konzentriert sich auf die Spezifikation und Konkretisierung von Konnektoren. Eine formale Verhaltensspezifikation von abstrakten Konnektoren als Applikationselemente und von konkreten Konnektoren, die von einer Middleware zur Verfügung gestellt werden, erlaubt es, einerseits zu prüfen, ob sich eine Middleware überhaupt zur Implementierung der Applikation eignet, andererseits automatisch abstrakte Applikationskonnektoren auf konkrete Konnektoren der Middleware abzubilden. Zur Verhaltensspezifikation werden Prozessalgebren eingesetzt.

Die deklarative Sprache Piccola [AN01] wurde entworfen, um Applikationen durch Komposition von Softwarekomponenten zu erstellen. Konnektoren sind Operatoren, die Komponenten verknüpfen. Die zulässigen Kompositionen werden mittels Algebren definiert. Alle Applikationselemente werden als Strukturen mit benannten Elementen dargestellt. Zur Ausführung werden Applikationen in das $\pi\mathcal{L}$ Kalkül übersetzt und entsprechend berechnet. Piccola unterstützt aktuell keine Verteilung.

Bei der Betrachtung der hier vorgestellten erweiterbaren architekturbasierten Koordinationssprachen wird deutlich, dass jedes System entweder für ein bestimmtes Anwendungsgebiet und damit nur für eine eingeschränkte Klasse von Applikationen oder primär zur Modellierung und nicht zur Entwicklung konzipiert ist. Im allgemeinen setzen die Entwicklungssysteme auf bestimmten Koordinationsparadigmen auf, wie z.B. auf der Kommunikation über Ereignisse oder auf der deklarativen Spezifikation von Abhängigkeiten. Dies erschwert es, neue Koordinationsparadigmen hinzuzufügen, die mit den Basisparadigmen nicht kompatibel sind. Um eine allgemeine Erweiterbarkeit zu ermöglichen, ist es demzufolge notwendig, kein bestimmtes Basiskoordinationsparadigma einzusetzen.