Kapitel 1

Einleitung

1.1 Motivation

In den letzten Jahren hat sich die Leistungsfähigkeit der Rechentechnik enorm erhöht, während parallel dazu die Kosten für die Komponenten, die für den Aufbau von Rechenanlagen benötigt werden, drastisch gesunken sind. Damit sind Anwendungen hoher Komplexität, wie sie noch vor wenigen Jahren undenkbar waren, zumindest von der Verfügbarkeit der Rechenleistung her realisierbar geworden. Zum anderen ist es nun möglich, informationstechnische Systeme auch in Geräte und Anlagen einzubauen, die noch vor kurzer Zeit vollkommen ohne sie auskommen mußten – sogenannte *eingebettete Systeme*.
Ein unvermeidbarer Nebeneffekt dieser Entwicklung ist die drastische Steigerung der zu behandelnden Komplexität, die durch jede neue Anwendung oder jede neue Funktionalität weiter steigt. Auf welche Weise dies geschieht, ist in vielen Fällen im Voraus nur schlecht oder nicht vorhersagbar.
Wird von der Domäne der Informationstechnik abstrahiert, so finden sich schon in der Frühzeit der Menschheit Lösungen für den Umgang mit wachsender Komplexität – „divide et impera“ ist über 2000 Jahre alt. Auch in der noch jungen Informatik existieren solche Ansätze schon lange. Strukturierte Programmierung, Modularisierung oder Objektorientierung sind Schlagworte dazu. Ihnen gemeinsam ist es, große und komplexe Probleme geeignet in kleinere Stücke zu zerlegen, die dann, sofern möglich, nach der gleichen Methode weiter zerlegt werden, bis ein beherrschbares Komplexitätsniveau erreicht ist. Dieses Zerlegen einer Aufgabe in kleine Teile wird als *Dekomposition* bezeichnet und das für das wirkliche System wichtige Zusammenfügen der Einzelteile als *Komposition*. Ein wichtiger Aspekt einer solchen Komposition ist die Interoperabilität der zusammenzufügenden Teile, also die funktionale Kompatibilität der jeweiligen Schnittstellen.
Der oben bereits erwähnte Umstand, daß informationstechnische Systeme zunehmend in immer mehr Bereichen des menschlichen Lebens und der künstlichen Umwelt auftauchen, bringt jedoch einen weiteren Aspekt mit sich, der einfach und naheliegend erscheinende Kompositionen unmöglich machen kann: Es sind nämlich nicht immer nur die *funktionalen Eigenschaften*, die relevant sind, sondern gerade bei Computersystemen, die eingebettet in anderen technischen Systemen ihre Arbeit verrichten, auch Eigenschaften, die mit der gewünschten Funktionalität nichts oder nur wenig zu tun haben, sogenannte *nichtfunktionale Eigenschaften*.
Diese nichtfunktionalen Eigenschaften sind dennoch bei derartigen Anwendungen bedeutsam, da sie indirekte Auswirkungen auf die Funktionalität des Systems haben. Beispielsweise ist die Zeit, die ein Computerprogramm für seine Abarbeitung benötigt, eine nichtfunktionale Ei-

genschaft, die gerade im Kontext eines in eine technische Anlage eingebetteten Systems von hoher Wichtigkeit ist: Wenn eine Berechnung erst zu einem Zeitpunkt abgeschlossen ist, zu dem die dadurch ausgelöste Reaktion des Systems für die Anwendung zu spät erfolgt, so kann das System seiner eigentlichen Funktion nicht mehr oder nur noch eingeschränkt nachkommen. Solche Systeme werden als Echtzeitsysteme bezeichnet und sind in der Praxis in vielen Anlagen zu finden, die zum Teil sehr wichtige Aufgaben wahrnehmen, bei denen eine Fehlfunktion Menschen zu Schaden kommen läßt.
Gebräuchliche Komponentensysteme berücksichtigen beim Komponieren ausschließlich funktionale Eigenschaften, die in Form (funktionaler) Schnittstellenbeschreibungen leicht zu spezifizieren und auf ihre Kompatibilität zu testen sind. Die Spezifikation nichtfunktionaler Eigenschaften ist dagegen weit komplizierter, da diese Eigenschaften nicht notwendigerweise nur von der Komponente und deren funktionaler Spezifikation (beispielsweise Quellcode) abhängen, sondern darüber hinaus auch von der Umgebung der Komponente (beispielsweise dem ausführenden Rechner, der seinerseits jedoch auch wieder eine Umgebung in Form von anderen Rechnern und einer physikalischen Umwelt besitzt) und möglichen Interaktionen mit anderen Komponenten. Letztere sind dabei unter Umständen vollständig unabhängig von den funktionalen Zusammenhängen zwischen den Komponenten.
Eine Komposition erfordert folglich, daß neben den funktionalen Eigenschaften der Komponenten auch ihre nichtfunktionalen Eigenschaften auf geeignete Weise Eingang in die benutzte Kompositionstechnik finden müssen, um ein vorhersagbares, sinnvolles Ergebnis der Komposition sicherzustellen.

1.2 Problemstellung

Aus den motivierenden Beschreibungen ergibt sich folgendes Problem:
„Wie kann unter Berücksichtigung nichtfunktionaler Eigenschaften komponiert werden?“
Ein weiteres Problem ist, daß es auf diese Frage sowohl sehr generische als auch sehr spezielle Antworten geben kann, wobei es zwischen diesen nicht notwendigerweise eine Abbildung geben muß. Generische Antworten können versagen, wenn es um die Anwendung auf ein konkretes Problem und eine konkrete Eigenschaft geht, während spezielle Antworten außerhalb ihres durch die Anwendung und die Zieleigenschaft gegebenen Kontextes keinen Wert haben, wenn sie sich nicht verallgemeinern lassen.
Ziel der vorliegenden Arbeit ist es darum, ein generisches Konzept für Komponierbarkeit in bezug auf Eigenschaften zu entwickeln, wobei auf Konzeptebene nicht zwischen funktionalen und nichtfunktionalen Eigenschaften unterschieden wird. Die Übertragbarkeit dieses Konzeptes auf konkrete Problemstellungen wird durch die Entwicklung einer Lösung für die Komponierbarkeit eingebetteter Echtzeitsysteme gezeigt. Die dabei betrachtete Eigenschaft ist das zeitliche Verhalten als nichtfunktionale Eigenschaft, wobei untersucht wird, inwieweit sich Erkenntnisse aus der Entwicklung der speziellen Lösung für die Anwendung des Komponierbarkeitskonzeptes auf andere Probleme verallgemeinern lassen.

1.3 Lösungsweg

In diesem Abschnitt wird der Weg zur Lösung des gestellten Problems skizziert und damit zugleich die Struktur der Arbeit vorgestellt. Abbildung 1.1, in der die im folgenden beschriebenen

Zusammenhänge der einzelnen Kapitel dargestellt sind, dient dabei als Illustration. Die Abbildung zeigt, daß der Lösungsweg entsprechend der Aufgabenstellung eingeteilt ist in allgemeine Betrachtungen zur Komponierbarkeit, die Entwicklung einer komponierbaren Architektur für eingebettete Echtzeitsysteme und schließlich verallgemeinernde und zusammenfassende Betrachtungen.

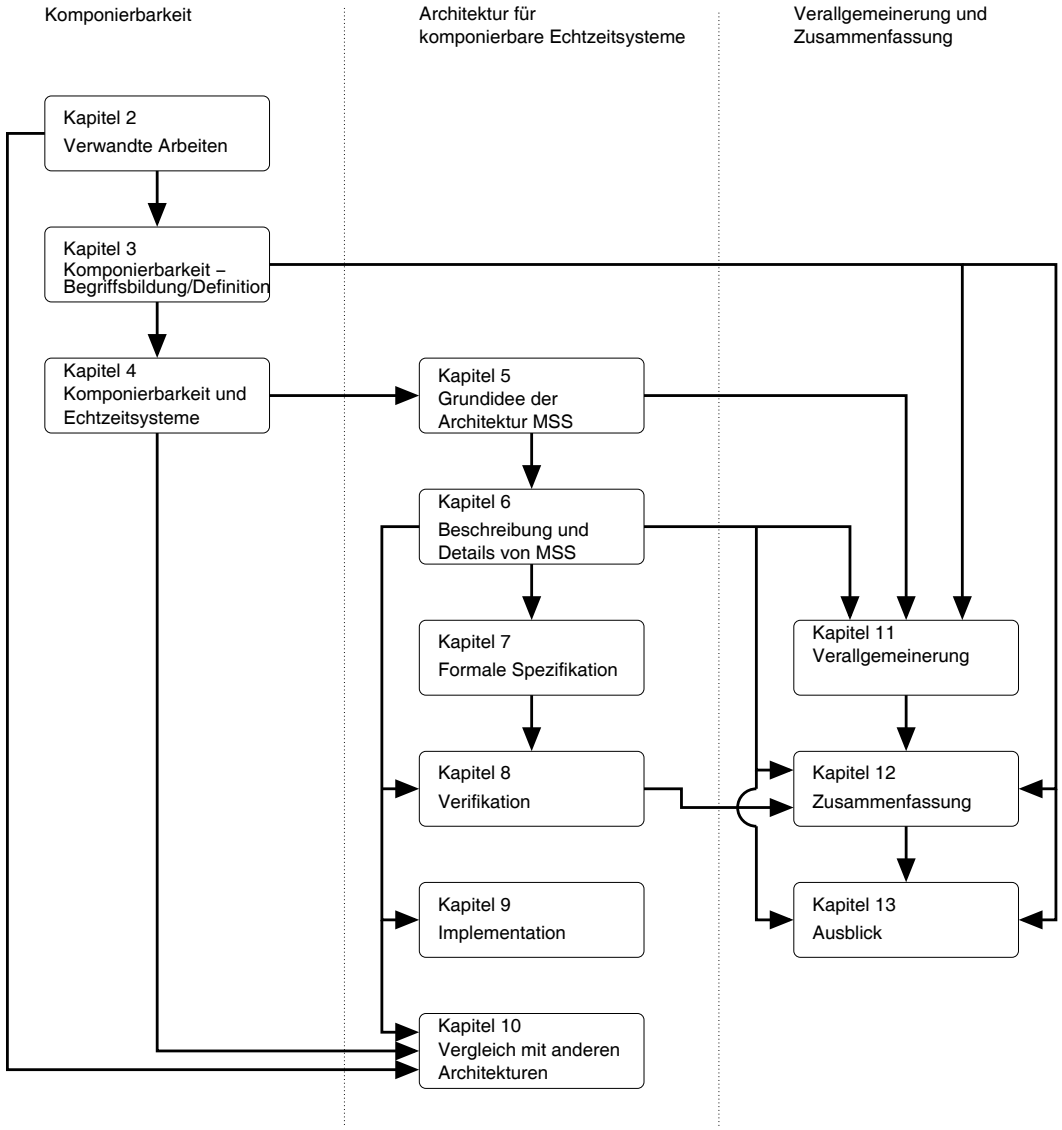

Abbildung 1.1: Zusammenhänge der Kapitel der Arbeit

Erstes Ziel der Arbeit ist es, ein generisches Konzept für Komponierbarkeit in bezug auf nichtfunktionale Eigenschaften zu entwickeln. Ausgangspunkt ist dabei die Analyse verwandter Arbeiten zum Thema „Komponierbarkeit“ im Kapitel 2. Entsprechend der Wortverwandtschaft zu „Komposition“ und „Komponente“ wird diese Analyse mit der Betrachtung des durch diese Begriffe gegebenen Umfeldes begonnen, was konkret die Vorstellung ausgewählter Komponentenarchitekturen und einer Methode zur Klassifikation von Komponentensystemen umfaßt. Im Anschluß daran werden verwandte Arbeiten zum Begriff der „Komponierbarkeit“ betrachtet, wobei eine Schwierigkeit darin liegt, daß viele Arbeiten eine Definition vermeiden und im wesentlichen vom intuitiven Verständnis ausgehen. Auf dieses Verständnis wird ebenso eingegangen wie auf Definitionen, die im Kontext bestimmter Architekturen und Systeme stehen.

Im Rahmen dieser Diskussion werden Anforderungen an das in dieser Arbeit zu entwickelnde Konzept der Komponierbarkeit aufgestellt, auf deren Grundlage schließlich im Kapitel 3 (Komponierbarkeit – Begriffsbildung und Definition) ein solches Konzept entwickelt wird, das „Komponierbarkeit in bezug auf eine Eigenschaft“ als eine Eigenschaft einer Systemarchitektur betrachtet. Dieses Konzept ist generisch sowohl hinsichtlich der Anwendungsdomäne als auch hinsichtlich der Zieleigenschaften, so daß funktionale und nichtfunktionale Eigenschaften betrachtet werden können.
Der zweite Teil der Problemstellung betrifft die Anwendung dieses Komponierbarkeitskonzeptes auf die nichtfunktionale Eigenschaft „zeitliches Verhalten“ im Kontext von eingebetteten Echtzeitsystemen. Auf Grundlage einer Einführung in das Gebiet der Echtzeitsysteme wird „Komponierbarkeit in bezug auf das zeitliche Verhalten“ im Kapitel 4 (Komponierbarkeit und Echtzeitsysteme) diskutiert und es werden Anforderungen an eine entsprechende Architektur aufgestellt.
Damit sind die Grundlagen geschaffen, eine hinsichtlich des zeitlichen Verhaltens komponierbare Architektur zu entwickeln. Diese Architektur trägt den Namen *Message Scheduled System* (MSS) und wird im Detail vorgestellt.
Begonnen wird diese Vorstellung im Kapitel 5 (Grundidee der Architektur MSS) mit einer Analyse des Gebietes der eingebetteten Echtzeitsysteme, aus der Anforderungen an die Architektur resultieren. Aus diesen Anforderungen und einer Diskussion über ereignis- und zeitgesteuerte Systeme wird die Grundidee von MSS entwickelt und zusammen mit Betrachtungen zu Kompositionen innerhalb von MSS vorgestellt.
Basierend auf dieser Grundidee wird MSS im Kapitel 6 (Beschreibung und Details von MSS) informal beschrieben. Dabei werden neben den technischen Details der Architektur Verfahren zur Komposition sowie garantierte Eigenschaften von MSS betrachtet.
Die damit gegebene Beschreibung ist jedoch nicht ausreichend, um die Einhaltung dieser Garantien nachzuweisen, so daß die Architektur MSS mit Hilfe von zeitbehafteten Petrinetzen formal spezifiziert wird. Die Spezifikation wird dabei selbst zum Gegenstand von Kompositionstechniken, indem Spezifikationen komplexer MSS-Systeme aus den Spezifikationen kleinerer Bausteine zusammengefügt (komponiert) werden.
Diese komponierbare Spezifikation wird im Kapitel 8 (Verifikation) benutzt, um die Einhaltung der Garantien von MSS formal nachzuweisen.
Abgeschlossen wird die Betrachtung der komponierbaren Architektur MSS mit zwei praktischen Aspekten, nämlich der Diskussion von Ansätzen zur Implementation im Kapitel 9 (Implementation) und Vergleichen mit anderen Architekturen im Kapitel 10 (Vergleich mit anderen Architekturen).
Der dritte Teil der Problemstellung, der sich mit der Fragestellung beschäftigt, inwiefern Techniken und Ansätze, die bei der Entwicklung der speziellen Architektur MSS benutzt werden, für die Anwendung des Komponierbarkeitskonzeptes auf andere Eigenschaften und Architekturen verwendet werden können, wird im Kapitel 11 (Diskussion) bearbeitet. Darüber hinaus wird diskutiert, inwiefern Architekturen entwickelt werden können, die komponierbar in bezug auf mehrere verschiedene nichtfunktionale Eigenschaften sind. Von besonderem Interesse sind dabei Eigenschaften, die miteinander in Zusammenhang stehen, so daß Vorkehrungen zur Sicherstellung einer Eigenschaft Auswirkungen auf eine andere Eigenschaft haben.
Am Ende der Arbeit stehen im Kapitel 12 (Zusammenfassung) zusammenfassende Bemerkungen und in Kapitel 13 (Ausblick) Betrachtungen, inwiefern die Arbeit fortgesetzt und erweitert werden kann, wobei sowohl das Konzept der Komponierbarkeit als auch die Architektur MSS im Mittelpunkt stehen.

Kapitel 2

Verwandte Arbeiten – Komponentensysteme und Komponierbarkeit

Ziel dieses Kapitels ist es, verwandte Arbeiten zum Thema „Komponierbarkeit“ vorzustellen und zu diskutieren. Da die Idee der Komponierbarkeit in der Literatur oft in Zusammenhang mit Komponentensystemen und den dazugehörigen Begriffen „Komposition“ und „Komponente“ zu finden ist, wird dieser Überblick über die verwandten Arbeiten mit Betrachtungen zu Komponentensystemen im Abschnitt 2.1 begonnen und mit einer Klassifikation von Komponentenmodellen im Abschnitt 2.2 fortgesetzt, ehe im Abschnitt 2.3 verwandte Arbeiten zum Begriff der „Komponierbarkeit“ diskutiert werden.

2.1 Komponentensysteme

2.1.1 Der Weg zu Komponentensystemen

Mit der voranschreitenden Entwicklung der Hardware-Technologien sind die technischen Voraussetzungen immer besser geeignet, um äußerst komplexe Software-Systeme realisieren zu können. Mit wachsender Komplexität steigen jedoch nicht nur die Anforderungen an die benutzte Hardware, sondern auch der Aufwand bei der Entwicklung, dem Test und der Verifikation dieser Software. Das Problem dabei ist, daß nicht wie im Hardware-Sektor davon ausgegangen werden kann, daß die erreichbare „Performance“ stetig ansteigt. Es müssen andere Lösungen betrachtet werden. Ein oft verfolgter Ansatz ist die Aufteilung der zu bewältigenden Aufgabe in kleinere Probleme, die unter Umständen weiter zerlegt werden, bis eine Vielzahl kleiner „Teile“ existiert, deren Funktionalität klar spezifiziert werden kann und deren korrekte Implementation anhand dieser Spezifikation der Funktionalität leicht verifizierbar ist. Auf diese Weise sind Probleme, die in der Komplexität eines umfangreichen Systems untergehen würden, leichter zu erkennen und zu behandeln.
Dijkstra schreibt dazu: „Die Technik, mit der Komplexität beherrscht werden kann, ist schon seit alter Zeit bekannt: *divide et impera* (teile und herrsche)“ [25].
In der Softwaretechnik existiert diese Art des Umgangs mit der Komplexität ebenfalls schon sehr lange – die Entwicklung prozeduraler und funktionaler Sprachen und das Aufkommen der Objektorientierung sind Belege dafür. Diesen Ansätzen ist gemein, daß die benutzten „Teile“ im

wesentlichen zur Benutzung innerhalb des jeweils betrachteten Systems bestimmt sind, die Teile sind also keine weitgehend eigenständigen *Komponenten*, die einfach zusammengefügt werden können. Das Zusammenfügen findet in den meisten Fällen auf der Ebene des Quellcodes oder des Linkens statt – also entweder das Zusammenkopieren verschiedener Quellcode-Abschnitte oder das Linken mehrerer Module zu einem ausführbaren Programm, beziehungsweise die Benutzung von Bibliotheken, die die betreffende Funktionalität enthalten. In Abhängigkeit davon, auf welche Weise das Zusammenfügen stattfindet, ist die Wiederverwendung solcher „Teile“ mehr oder weniger einfach möglich, wobei die Spanne von Quelltextteilen, die per „Kopieren und Einfügen“ aus einem Programmteil in einen anderen genommen werden bis hin zu Standard-Bibliotheken reicht, die innerhalb einer komplexen Softwareumgebung für die verschiedensten Zwecke eingesetzt werden und standardisierte Schnittstellen benutzen.
Als nächster Schritt in der Entwicklung sind Komponentensysteme auf Basis von beispielsweise DCE [138, 139, 164, 171, 191], CORBA [118–120, 122–129, 131–133, 174], COM+, DCOM, .NET [39, 96, 154] oder Java [38] populär geworden, die im folgenden Abschnitt 2.1.2 kurz vorgestellt werden. Diese Systeme (mit Ausnahme von Java auch als Standard-Middleware bezeichnet) erlauben die Spezifikation von systemunabhängigen[1] Schnittstellen zwischen verschiedenen Komponenten und damit das Komponieren von Komponenten, ohne Details der jeweiligen Implementation und Installation kennen zu müssen. Gegenüber beispielsweise der Benutzung von Standardbibliotheken wird so unter anderem Verteilungstransparenz, Plattformunabhängigkeit und Sicherheit erreicht, jedoch liegt der Preis in einem erhöhten Aufwand zur Laufzeit, da entsprechende Algorithmen benutzt werden müssen, die für die nötigen Abbildungen beispielsweise zwischen den plattformabhängigen Repräsentationen von Daten sorgen und systemspezifische Techniken für die Netzwerkkommunikation kapseln.

2.1.2 Beispiele für Komponentensysteme

In diesem Abschnitt sollen einige Architekturen für Komponentensysteme kurz vorgestellt werden, um auf dieser Basis im Abschnitt 2.1.3 allgemeine Eigenschaften solcher Systeme zu diskutieren.

2.1.2.1 Distributed Computing Environment – DCE

Zur Lösung des Problems der einheitlichen Verwaltung von Nutzern und Daten sowie des sicheren Zugriffs auf Daten und Hardware-Ressourcen in großen heterogenen Rechnersystemen hat die *Open Software Foundation* (später *Open Group*) Anfang der 90er Jahre das *Distributed Computing Environment* (DCE) [138, 139, 164, 171, 191] als Standard etabliert. DCE setzt dabei auf den Betriebssystemen der einzelnen Systeme auf und bietet auf Anwendungsebene Schnittstellen, die die heterogene Natur der zugrunde liegenden Systeme verbergen und Standard-Dienste bereitstellen.
Auf diesen Aspekt von DCE soll hier nicht im Detail eingegangen werden, sondern entsprechend dem auf Komponentensysteme gesetzten Fokus Eigenschaften und Dienste von DCE betrachtet werden, die auf komponentenorientierte Softwareentwicklung zielen. Das Ziel von

[1] „Systemunabhängig“ ist im wesentlichen im Sinne von „betriebssystemunabhängig“ und „hardwareplattformunabhängig“ zu verstehen. Systemunabhängig im Sinne von „nicht an einen Rechner gebunden“ sind auch Bibliotheken, die auf allen binärkompatiblen Rechnern, die ein kompatibles Betriebssystem benutzen, verwendet werden können.

DCE selbst wird als die Bereitstellung einer *Common infrastructure for distributed processing* [191] gesehen.
DCE adressiert in Zusammenhang mit der Bereitstellung einer solchen Infrastruktur die folgenden Aspekte:

- Definition von Client/Server als Mittel des entfernten Aufrufs
- Lokation von Ressourcen mittels geeigneter Dienste (Namens- und Verzeichnisdienst), um Skalierbarkeit, Replikation und Administration zu erleichtern
- Sicherheit (im Sinne von *security*) mit Unterstützung für Authentisierung, Authorisierung, Integrität und Schutz der Daten
- Skalierbarkeit durch die Einführung von DCE-Zellen als Einheit der Administration und der Möglichkeit sicherer Zugriffe innerhalb der Zelle und auf andere Zellen als ein weit verteiltes Dateisystem

Um die damit verbundenen Ziele zu erreichen, gibt es in der DCE-Architektur eine Reihe von Diensten, darunter:

- *Thread Service*
  DCE bietet einen Dienst an, der von unterschiedlichen Threadmodellen verschiedener Betriebssysteme abstrahiert und damit eine einheitliche Schnittstelle bietet.
- *Directory Service*
  Der verteilte Verzeichnisdienst von DCE bietet ein Namensmodell an, das innerhalb der gesamten verteilten Umgebung benutzt wird, um jede Art von Ressourcen über einen Namen anzusprechen und auf sie zugreifen zu können, ohne um die Lokation innerhalb des Netzes zu wissen.
- *Distributed Time Service*
  Verteilte Anwendungen hängen in vielen Fällen von einer einheitlichen und konsistenten Sicht auf die Zeit ab. Aus diesem Grund besitzt DCE einen verteilten Zeitdienst, der eine präzise, fehlertolerante Uhrensynchronisation für die beteiligten Systeme anbietet.
- *Security Service*
  Weit verteilte Systeme kommunizieren über Netzwerke, die potentiell gefährdet sind gegenüber Beobachtung oder Verfälschung. Das Problem nimmt mit zunehmender Globalisierung zu, zumal in vielen Fällen (z.B. Verbindung über das Internet) nicht einmal bekannt ist, welcher Weg genommen wird und welche Systeme beteiligt sind. Aus diesem Grund verfügt die DCE-Architektur über einen Sicherheitsdienst, der Authentisierung, Authorisierung, Integrität und Schutz der Daten anbietet und dabei auf bekannte und bewährte Verfahren wie Kerberos [59, 60, 102] aufsetzt. Innerhalb einer DCE-Umgebung fungiert der *Security Service* als vertrauenswürdige dritte Partei den Anwendungen gegenüber und erlaubt es auf diese Weise, die Gültigkeit beispielsweise von Schlüsseln zu überprüfen, ohne diese innerhalb der Anwendung speichern zu müssen.
- *Distributed File System*
  Der Philosophie von DCE folgend, ist das verteilte Dateisystem eine Client/Server-Anwendung, die den gemeinsamen und entfernten Zugriff auf Dateien in der gleichen Weise

erlaubt wie den Zugriff auf lokale Daten. Lokationstransparenz wird ebenso wie ein einheitliches Namensschema mit Hilfe des verteilten Verzeichnisdienstes erreicht. Der Sicherheitsdienst wird benutzt, um Zugriffsrechte zu verwalten und durchzusetzen. Transparente Replikation sorgt für hohe Verfügbarkeit auch im Falle des Ausfalls einzelner Systeme, während ein tokenbasiertes Protokoll für die Konsistenz sowohl zwischen mehreren Kopien als auch bei parallelen Zugriffen verantwortlich ist.

Diese Dienste basieren auf dem grundlegenden Konzept des entfernten Prozeduraufrufes (*remote procedure call* – RPC). RPCs in DCE erweitern das lokale Aufrufmodell so, daß direkte Aufrufe von Prozeduren auf entfernten Systemen unterstützt werden, wobei unterschiedliche Repräsentationen von Daten ebenso wie Unterschiede in den Details der Vernetzung durch den RPC-Dienst maskiert werden. Die Schnittstellen für RPC werden mit Hilfe einer Schnittstellen-Beschreibungssprache (*interface definition language* – IDL) beschrieben, während sämtliche relevanten Entitäten (Nutzer, Ressourcen, Dienste) in DCE mit *universal unique identifier* (UUID) eindeutig bezeichnet werden.

2.1.2.2 Common Object Request Broker Architecture – CORBA

Im Rahmen des Bestrebens, die Entwicklung einheitlicher, plattformübergreifender, objektorientierter und verteilter Anwendungen zu fördern, wurde 1989 von führenden IT-Unternehmen (u.a. 3Com, IBM, HP, Sun) die *Object Management Group* (OMG) [135] gegründet. Aufgabe der OMG ist es, Spezifikationen zu entwickeln und bereitzustellen, die dann von den Mitgliedern implementiert werden können, wobei der Schwerpunkt die Entwicklung einer Architektur für verteilte Anwendungen unter Benutzung von Techniken der Objektorientierung ist.
Diese Architektur ist die *Object Management Architecture* (OMA) [121] der OMG, die sowohl ein Objektmodell für verteilte objektorientierte Architekturen als auch ein Referenzmodell für die verschiedenen Komponenten eines solchen Systems definiert.
Kernidee dieser Architektur ist es, einen *Object Request Broker* (ORB) als grundlegende Infrastruktur für die Kommunikation zwischen Objekten zu benutzen, wobei die Kommunikation unter Abstraktion von Lokation, Plattform und Programmiersprache über Nachrichten erfolgt.
Neben der Kommunikation zwischen anwenderspezifischen Objekten kann über den ORB auch mit Diensten („CORBAservices“) kommuniziert werden, die der Verwaltung eines CORBA-Systems dienen und von der OMG spezifiziert werden – beispielsweise Namensdienste, Sicherheitsdienste, Ereignisdienste oder Transaktionsdienste.
Solche Dienste sind ebenfalls auf Anwendungsebene möglich („CORBAfacilities“), wobei in verschiedenen Domänen Standard-Dienste für domänenspezifische Zwecke definiert werden.
Common Object Request Broker Architecture (CORBA) spezifiziert die Architektur eines ORBs und damit unter anderem Sprachen, Protokolle und Schnittstellen für die Entwicklung von ORBs und darauf basierenden Anwendungen. Neben dem ORB selbst ist dabei die Sprache *Interface Definition Language* (IDL) als Spezifikationssprache für die Schnittstellen von OMA-Objekten hervorzuheben. IDL definiert den Typ eines Objektes durch die Spezifikation der Schnittstelle, die aus einem Satz benannter Operationen und deren Parametern besteht.
IDL ist von der Syntax her ähnlich der Programmiersprache C, ist aber als Schnittstellen-Beschreibungssprache unabhängig von der tatsächlich benutzten Programmiersprache, zu der die Abbildung über sogenannte Sprach-Bindungen geschaffen wird, die unter anderem für die Kompatibilität von Typen, Konzepten, Aufrufsemantiken usw. sorgen.