Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Überblick

1.1 Einführung

Das an der Universität Stuttgart am IPVR[1] entwickelte APRICOT-System[2] dient zur zuverlässigen Abwicklung von langlebigen, verteilten Datenbank-Anwendungen. Mit Hilfe dieses Systems sollen ConTract-Skripte sicher und kontrolliert abgearbeitet werden.

Ein ConTract-Skript ist eine formale Darstellung einer Ablaufbeschreibung einer speziellen Anwendung. Ein Skript besteht aus Basiseinheiten, den Steps, die durch eine Kontrollflußbeschreibung sowie statischen Transaktions-Beziehungen verbunden sind. Die ConTract-Skripte werden von dem ConTract-Manager abgewickelt, einem möglichst sicheren Laufzeitsystem.

Das APRICOT-System besteht aus einer Menge von Server-Prozessen. Das sind zum einen die Resource Manager, die transaktionsgeschützte Dienstaufrufe (sogenannte Services) implementieren. Sie verwalten bestimmte, gemeinsam benutzte Objekte (Ressourcen) wie z. B. eine Datenbank (Bsp.: Verwaltung eines Warenlagers einer Fabrik) oder eine Fertigungsmaschine (Bsp.: Bohren, Fräsen oder Schneiden eines Werkstücks).

Die andere Menge von Server-Prozessen sind die Step Computation Server, die die Steps beinhalten. Steps sind Programmschritte einer Anwendung. Die Steps können auf gemeinsam benutzte Objekte, also auf Resource Manager, zurückgreifen, um die nötigen Teilaufgaben dieser Anwendung durchzuführen.

[1]Institut für Parallele und Verteilte Höchstleistungsrechner
[2]A PRototype Implementation of a COnTract System

1.2 Aufgaben und Ziele der Studienarbeit

Aufgabe dieser Studienarbeit ist es, eine interaktive Programmierumgebung zu entwerfen und zu implementieren, die dazu dient, die Definition, Erstellung und Verwaltung von großen, transaktionsgeschützten Client-Server-orientierten Anwendungen in ConTracts mit graphischen Mitteln zu unterstützen. Das fängt an beim Entwurf und der Definition von Services, geht weiter über die Definition von Resource Managern und APRICOTS-Komponenten, bis hin zur automatischen Erstellung von Resource Managern und speziell Step Computation Servern, sowie der Verwaltung und Steuerung der Komponenten in der Laufzeitumgebung.

Als Teil der Aufgabe war ursprünglich auch die Steuerung (starten, stoppen) und die Verwaltung von ConTracts vorgesehen. Diese Teilaufgabe wurde aber inzwischen zu einer eigenen Diplomarbeit ausgegliedert, da ihr Umfang zu groß für den Rahmen dieser Studienarbeit gewesen wäre.

Der praktische Teil der Studienarbeit gliederte sich wie folgt:

- Entwurf eines Informationsmodells der einzelnen Teile und des globalen Verarbeitungszustandes eines APRICOT-Systems.
- Implementierung dieses Informationsmodells mittels Ingres / Windows 4GL unter dem X-Window-System.
- Entwurf und Implementierung der darauf aufsetzenden graphischen Oberfläche, unter Berücksichtigung bekannter Kriterien zur Gestaltung von Mensch-Computer-Schnittstellen.
- Integration der Oberfläche in das APRICOT-System.

1.3 Übersicht über die einzelnen Kapitel

In *Kapitel 2* steht die Erstellung einer APRICOTS-Komponente, im speziellen eines Resource Managers bzw. eines Step Computation Servers im Vordergrund. Es werden Fragen behandelt wie z. B. „Was muß man machen, um einen Resource Manager oder einen Step Computation Server „von Hand“ zu erstellen?“ oder „Welche Unterstützung bietet die graphische Oberfläche?“.

Kapitel 3 beschreibt das ConTract Processing Repository, die Datenbasis, mit dem ihm zugrunde liegenden Informationsmodell. Einige der später verwendeten Relationen werden genauer beschrieben.

Im darauffolgenden *Kapitel 4* wird das APRICOTS User Interface, das graphische Hilfsmittel zur Unterstützung des Anwenders und des Service-Programmierers, beschrieben. Insbesondere werde ich allgemeine Entwurfskriterien vorstellen, die Oberfläche in groben Zügen beschreiben sowie die Schritte vorstellen, die notwendig sind, um eine Anwendung zu erstellen.

Kapitel 5 geht auf einige offen gebliebene Probleme ein, die während dem Entwurf und der Implementierung der graphischen Oberfläche aufgetreten sind. Es werden die Probleme beschrieben sowie Lösungsvorschläge mit ihren Vor- und Nachteilen vorgestellt.

Zum Abschluß beschreibe ich in *Kapitel 6* mögliche Erweiterungen der Oberfläche, deren Realisation den Umfang dieser Arbeit gesprengt hätte, und mache Vorschläge, wie diese Erweiterungen aussehen könnten.

Kapitel 2

Funktionen der graphischen Oberfläche

Die Hauptaufgabe der graphischen Oberfläche ist die Definition, Erstellung und Verwaltung von großen transaktionsgeschützten Client-Server-orientierten Anwendungen, den ConTracts. In diesem Kapitel geht es in erster Linie um die Frage, was man machen muß, um eine solche Client-Server-orientierte Anwendung zu erstellen. Das ist einmal die Definition und Programmierung von Services, Resource Managern und Step Computation Servern. Zum andern muß eine APRICOTS-Komponente definiert und aus dieser Definition eine ablauffähige Komponente erstellt werden. Der ganze Ablauf ist in Abbildung 2.1 skizziert.

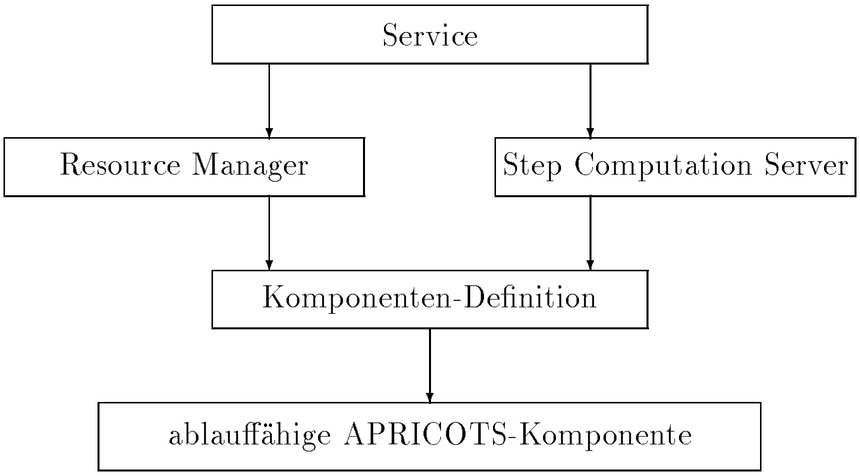

Abbildung 2.1: Erstellen einer APRICOTS-Komponente

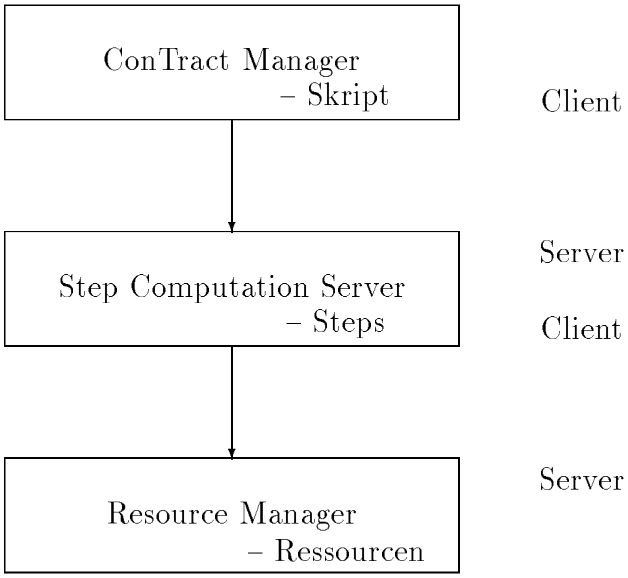

Abbildung 2.2: Client-Server Architektur

Die dem APRICOT-System zugrunde liegende Client-Server Architektur ist in Abbildung 2.2 dargestellt.

Ein Service ist ein Dienst, der gemeinsam benutzte Objekte (Ressourcen) verwaltet. In der Regel ist das eine Datenbank oder auch z. B. eine Fertigungsmaschine. Resource Manager (Objektverwalter) beinhalten diese Dienste. Diese werden von den Steps aufgerufen, die in Step Computation Servern eingebettet sind. Steps sind Teilschritte einer speziellen Anwendung. Sie werden in einem Skript mit Hilfe von Kontrollflußkonstrukten zu einer Ablaufbeschreibung dieser speziellen Anwendung zusammengefaßt. Dieses Skript wird vom ConTract-Manager abgearbeitet.

In Abbildung 2.3 ist das APRICOTS-Prozeßmodell veranschaulicht. Auf der rechten Seite sind die statischen Definitionen abgebildet. Das sind die Definitionen der Dienste, die Definitionen der Server, also der Resource Manager und Step Computation Server sowie die Prozeß-Definition. Auf der linken Seite sind die zugehörigen Prozesse zu sehen, zum einen die ablauffähige APRICOTS-Komponente, die Prozeß-Instanz, zum andern die Resource Manager-Instanz. Die Abhängkeiten zwischen den Definitionen und den Prozessen sind ebenfalls in der Abbildung dargestellt.

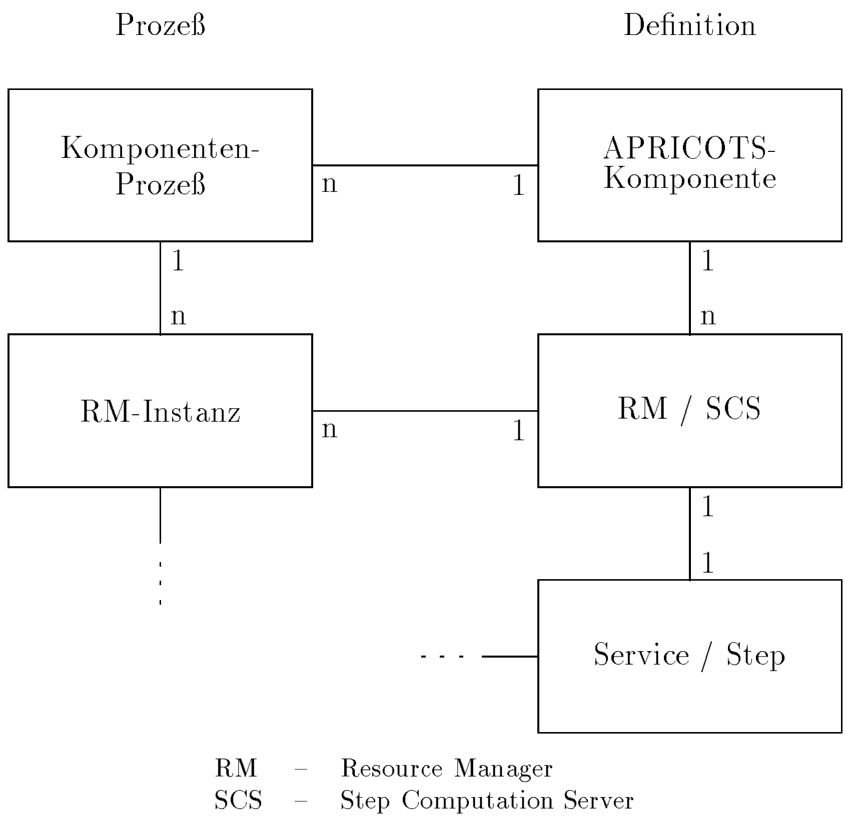

Abbildung 2.3: APRICOTS-Prozeßmodell

2.1 Notwendige Schritte bis zur fertigen Komponente

In diesem Abschnitt stelle ich kurz die notwendigen Schritte vor, um eine ablauffähige APRICOTS-Komponente zu erstellen. Dazu muß man zunächst Services (bzw. Steps) programmieren. In dieser Ausarbeitung unterscheide ich nicht streng zwischen Services und Steps. Ich verwende im folgenden häufig das Wort »Service«, auch wenn an einigen Stellen teilweise »Step« stehen müßte. Das ist zum einen möglich, da sich in dieser Studienarbeit meist alles auf beides bezieht, d. h. auf Services und Steps. Zum anderen werden Services und Steps immer austauschbar gestaltet, d. h. ein Service kann als Step und umgekehrt verwendet werden.

Neben dem eigentlichen Service, dessen Code im File `<service_name>.c` steht, gehören dazu noch weitere Funktionen:

- `<service_name>_get()`
  `<service_name>_put()`

Diese beiden C-Funktionen werden für einen Step Computation Server benötigt. Steps benötigen sogenannte Kontext-Elemente[1]. Diese werden mit den beiden Funktionen aus der Kontext-Datenbank geholt bzw. dort abgelegt.

- `<service_name>_decode()`
  `<service_name>_encode()`
  Bei der Erstellung eines Resource Managers werden diese beiden C-Funktionen verwendet. Sie dienen dazu, die von einem Service benötigten Daten aus dem dem Resource Manager übergebenen Parameter zu extrahieren bzw. die vom Service ermittelten Daten in den Rückgabeparameter zu schreiben.

- `<service_name>_decl.h`
  Ein C-Include-File, in dem alle vom Service benötigten Variablen deklariert sind. Dieses File wird sowohl bei der Erstellung eines Step Computation Servers als auch bei der Erstellung eines Resource Managers benötigt.

Auf die weitere Verwendung dieser Funktionen gehe ich in den nächsten beiden Abschnitten genauer ein.

Nachdem die Services programmiert sind, können die Resource Manager und die Step Computation Server definiert werden. Dies geschieht hauptsächlich durch die Angabe, welcher Service (bzw. Step) darin enthalten sein soll. In späteren Versionen soll es auch möglich sein, mehrere Services in einem Resource Manager zu verwenden. Nach der Definition können die Resource Manager und Step Computation Server erstellt (programmiert) werden. Darauf werde ich im nächsten Abschnitt genauer eingehen.

Nachdem einige Resource Manager und Step Computation Server fertiggestellt sind, kann man eine APRICOTS-Komponente definieren. Dazu muß man alle Resource Manager und Step Computation Server angeben, die in der Komponente enthalten sein sollen. Eine ablauffähige APRICOTS-Komponente erhält man, indem das Programm *make_apricot_component*[2] aufgerufen wird, nachdem man die für dieses Programm notwendige Dateien erstellt hat. Auch dies werde ich später ausführlich erläutern.

2.2 Programmierung eines Step Computation Servers

Ein Step Computation Server beinhaltet immer genau einen Step. Für die Erstellung eines Step Computation Servers werden folgende vom Service-Programmierer erstellten Funktionen (vrgl. letzter Abschnitt) benötigt:

[1]Spezielle Daten, die in einer Datenbank abgelegt sind. Genaueres siehe [Bilger92].
[2]Autor: Dieter Laube

- <service_name>()
- <service_name>_get()
- <service_name>_put()
- <service_name>_decl.h

Jede Funktion steht in einem gleichnamigen File mit der Endung ».c« z. B. heißt das zu der Funktion <service_name>_get() gehörende File <service_name>_get.c.

Zusätzlich wird eine Funktion benötigt, die die Step-Instanz-Daten aus dem übergebenen Parameter des Kommunikationssystems extrahiert:

```
Extract_Step_Instance(INBUFFER, *stepinst).
```

In Abbildung 2.4 ist ein grober Aufbau eines Step Computation Servers zu sehen. Als Beispiel enthält er einen Step mit Namen *Speditionsauftrag*. Der Step hat als Eingangsparameter die Variable *auftragsnummer*, Ausgangsparameter ist *speditionsname*.

Zu dem Aufbau des Step Computation Servers noch einige Anmerkungen:

- In Zeile 1 steht der Funktionsaufruf des Step Computation Servers. Der Aufrufname wird erzeugt, indem man vor den eigentlichen Step-Namen das Wort »Step_« hängt. Allgemein sieht der Aufuf eines Step Computation Servers also folgendermaßen aus:

  ```
  Step_<service_name>( ... );
  ```

- In den Zeilen 12 bis 15 geht es um die Entry-Invariante. Diese Invariante dient zur Überprüfung einer bestimmten Bedingung, zur Sicherung eines bestimmten Zustandes. Sie wird vor dem Step ausgeführt.
  Beispiel: Bevor Teile aus einem Lager geholt werden, sollte überprüft werden, ob genügend Teile im Lager vorhanden sind. Diese Überprüfung ist in diesem Beispiel die Entry-Invariante.

- <Verlasse_Funktion> in den Zeilen 15, 18, etc. bedeutet, daß die darauffolgenden Aktionen nicht mehr ausgeführt werden und die laufende Funktion mit einer Fehlermeldung verlassen wird (es erfolgt ein Sprung in Zeile 31).

- Zeilen 26-29, Exit-Invariante: Ähnlich der Entry-Invariante, nur wird sie nach einem Step ausgeführt. Die Exit-Invariante dient dazu, einen bestimmten Zustand festzustellen.
  Beispiel: In einem Step sollen 100 Schrauben in ein Lager gelegt werden. Eine mögliche Exit-Invariante ist es, hinterher zu überprüfen, ob auch wirklich mehr als 100 Schrauben in dem Lager liegen.

```
Step_Speditionsauftrag(INBUFFER, OUTBUFFER)
{
#include <Speditionsauftrag_decl.h>
ADT_stepinst stepinst; /* Deklaration des ADTs stepinst */
/* weitere Variablendeklaration */
Extract_Step_Instanz(INBUFFER, &stepinst);
                  /* Extrahieren der Step-Instanz-Daten */
/* holen der Entry-Invariante aus dem ADT stepinst */
/* Ausfuehren der Entry-Invariante, falls vorhanden */
/* Falls Fehler --> <Verlasse_Funktion> */
Speditionsauftrag_get(stepinst,
                           &auftragsnummer, &speditionsname);
if (FEHLER) then <Verlasse_Funktion>;
Speditionsauftrag(auftragsnummer, &speditionsname);
if (FEHLER) then <Verlasse_Funktion>;
Speditionsauftrag_put(stepinst,
                             auftragsnummer, speditionsname);
if (FEHLER) then <Verlasse_Funktion>;
/* holen der Exit-Invariante aus dem ADT stepinst */
/* Ausfuehren der Exit-Invariante, falls vorhanden */
/* Falls Fehler --> <Verlasse_Funktion> */
RM_Call_Return;
}
```

Abbildung 2.4: Aufbau eines Step Computation Servers

- Die Funktion RM_Call_Return(...) dient zum Verlassen / Beenden eines Resource Manager Aufrufs. Sie wird in [Laube92] genauer beschrieben.

Um nun einen Step Computation Server zu programmieren, muß man in erster Linie die für die Entry- und Exit-Invarianten-Überprüfung benötigten Funktionen erstellen. Der Rest besteht dann daraus, die einzelnen Teile zusammenzufügen sowie den Programmteil für die Überprüfung der Rückgabewerte zu schreiben. Trotzdem erfordert es einigen Aufwand, einen Step Computation Server zu programmieren, obwohl dabei im Prinzip jedesmal dasselbe gemacht werden muß.

2.3 Programmierung eines Resource Managers

Ein Resource Manager besteht immer aus einem Dienst. Der Aufbau eines Resource Managers ist einfacher als der eines Step Computation Servers, ansonsten aber sehr ähnlich. In erster Linie fehlen die beiden Invarianten-Überprüfungen. Der in einem Resource Manager enthaltens Service bekommt seine Parameter mit dem Aufruf übergeben, während der im Step Computation Server enthaltene Step sie aus dem Kontext holt. Das hat zur Folge, daß die Parameter des Service nicht stabil sind, d. h. sie können bei einem System-Absturz verloren gehen, was bei den Parametern der Steps nicht passieren kann.

Im Unterschied zum Step Computation Server werden folgende Funktionen benötigt:

- `<service_name>()`
- `<service_name>_decode()`
- `<service_name>_encode()`
- `<service_name>_decl.h`

Der grobe Aufbau des Resource Managers ist in Abbildung 2.5 dargestellt. Als Beispiel wird in dieser Abbildung ein Service namens *Preis_Berechnen* verwendet. Seine Eingangsparameter sind *Artikel_Nummer* und *Menge*, sein Ausgangsparameter ist *Preis*. Weitere Erläuterungen zu dem Aufbau eines Resource Managers siehe bei den Erläuterungen zum Step Computation Server Aufbau im Abschnitt 2.2.

Die Programmierung eines Resource Managers ist, wie man sieht, erheblich einfacher als die Programmierung eines Step Computation Servers. Im Prinzip muß man nur die einzelnen Funktionsaufrufe zusammenfügen und den Code für die Überprüfung der Rückgabewerte hinzufügen. In der Tabelle 2.1 sind noch einmal die benötigten Funktionen und Teile eines Resource Managers und eines Step Computation Servers im Vergleich aufgeführt.

```

Service_Preis_Berechnen(INBUFFER, OUTBUFFER)
{
#include <Preis_Berechnen_decl.h>

/* Variablen-Deklaration */

Preis_Berechnen_decode(INBUFFER, &Artikel_Nummer,
                                          &Menge, &Preis);
     /* Extrahieren der benoetigten Variablen aus INBUFFER */
if (FEHLER) then <Verlasse_Funktion>;

Preis_Berechnen(Artikel_Nummer, Menge, &Preis);
     /* Ausfuehrung des Service */
if (FEHLER) then <Verlasse_Funktion>;

Preis_Berechnen_encode(&OUTBUFFER, Artikel_Nummer,
                                          Menge, Preis);
     /* Rueckgabewerte in OUTBUFFER schreiben */
if (FEHLER) then <Verlasse_Funktion>;

RM_Call_Return( ... );
}

```

Abbildung 2.5: Aufbau eines Resource Managers

Resource Manager	Step Computation Server
`<service_name>_decl.h`	`<service_name>_decl.h`
---	`Extract_Step_Instance.c`
---	`Entry_Invariant.c`
`<service_name>_decode.c`	`<service_name>_get.c`
`<service_name>.c`	`<service_name>.c`
`<service_name>_encode.c`	`<service_name>_put.c`
---	`Exit_Invariant.c`

Tabelle 2.1: Aufbau Resource Manager vs. Step Computation Server — Aus implementierungstechnischer Sicht

2.4 2-Phasen-Commit

Im APRICOT-System werden übergeordnete Transaktionen angeboten, d. h. mehrere Steps können unter einer Transaktion abgearbeitet werden (siehe Abbildung 2.6).

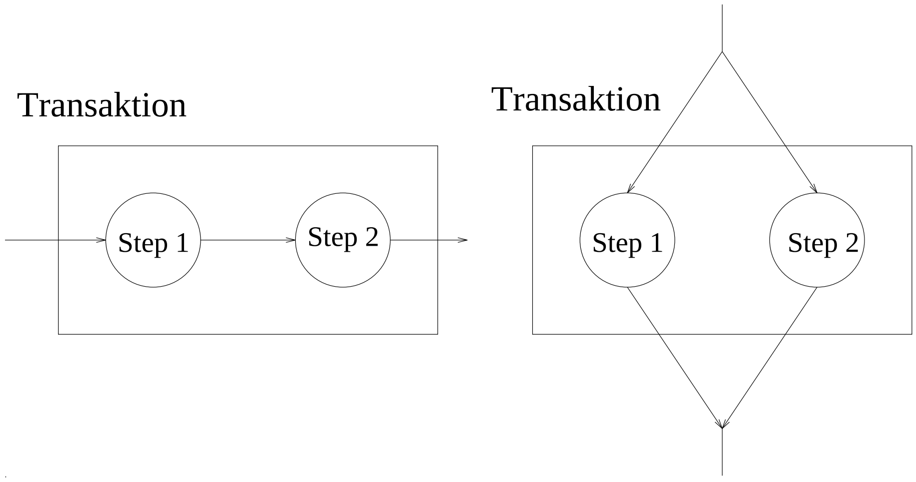

Abbildung 2.6: Beispiele für Transaktionen

Um zu gewährleisten, daß eine solche Transaktion ordnungsgemäß beendet wird, muß ein 2-Phasen-Commit angeboten werden. Aus diesem Grund muß jeder Resource Manager und Step Computation Server folgende Funktionen anbieten:

- `prepare()`
- `commit()`
- `rollback()`

Die Schnittstellen dieser Funktionen sollen der XA-Spezifikation [XOpen91] entsprechen.

2.5 Erstellung einer ablauffähigen APRICOTS-Komponente

Eine APRICOTS-Komponente (siehe Abbildung 2.7) besteht aus Resource Managern und / oder Step Computation Servern. Hinzu kommt ein Teil des Kommunikationssystem [Laube92], der für den internen Ablauf der Komponente sowie für das Empfangen und Senden von Nachrichten zuständig ist (der Teil des Kommunikationssystems ist das eigentliche Hauptprogramm). Die Resource Manager und Step Computation Server bestehen aus je genau einem Service. In einer späteren Version soll es möglich sein, daß zumindest der Resource Manager aus mehreren Services bestehen kann. Bisher ist das wegen Problemen bei der Implementierung noch nicht der Fall.

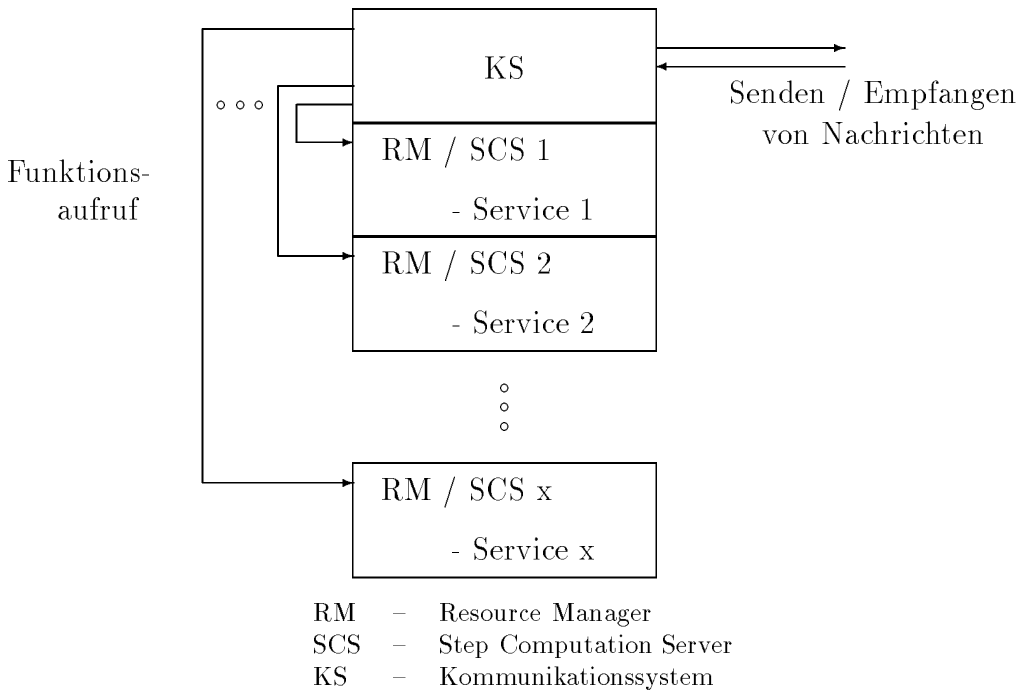

Abbildung 2.7: APRICOTS-Komponente

Um eine APRICOTS-Komponente zu erstellen, muß man sie zuerst definieren. D. h., man muß die Resource Manager und Step Computation Server zusammenstellen, die in dieser Komponente enthalten sein sollen.

Danach müssen Definitionsdateien erstellt werden, die von dem Programm *make_apricot_component* benötigt werden (vgl. [Laube92/1]). Schließlich kann dieses Programm gestartet werden, um eine ablauffähige Komponente zu generieren.

Die Definitionsdatei hat einen beliebigen Namen mit der Endung ».def«, z. B. *comp.def*. In dieser Datei stehen zuerst die Linkfiles. Das sind die Files, die beim Linken der Komponente hinzu gebunden werden müssen. Dann kommen die Definitionen der Funktionen, wobei in jeder Zeile genau eine Funktion definiert wird. Eine Zeile ist folgendermaßen aufgebaut:

```
service_name<tab>resource_manager_class<tab>c_function_name
```

Service_name ist der Name des in dem Resource Manager bzw. Step Computation Server namens *resource_manager_class* enthaltenen Service, *c_function_name* ist der Name der C-Funktion, die diesen Dienst realisiert.

Zu jeder Definitionsdatei muß es eine Datei mit dem gleichen Namen geben, allerdings mit der Endung ».h« (im Beispiel oben hieße die Datei *comp.h*). In diese Datei muß der Programm-Code aller Resource Manager und Step Computation Server geschrieben werden, die in der APRICOTS-Komponente enthalten sind und deren Aufruf in der Definitionsdatei beschrieben ist.

Der Aufruf von *make_apricot_component* sieht dann so aus:

```
make_apricot_component comp.h -o Komponente,
```

wobei der Parameter *-o Komponente* den Namen der fertigen Komponente angibt. Allgemein erfolgt der Aufruf durch

```
make_apricot_component <file1>.h <file2>.h ... -o <outputname>,
```

wobei *<filex>.h* jeweils den Code eines oder mehrerer Resource Manager und / oder Step Computation Server enthält. Zu jedem File *<filex>.h* existiert auch eine Definitionsdatei *<filex>.def* wie oben beschrieben.

Die ablauffähige APRICOTS-Komponente besteht dann aus den einzelnen Resource Managern und den Step Computation Servern sowie einem Teil des Kommunikationssystems, der für das Senden und Empfangen von Nachrichten zuständig ist und die Aufrufe der Services verwaltet (vergleiche Abbildung 2.7).

2.6 Unterstützung durch das User Interface

In den vorangegangenen Abschnitten habe ich gezeigt, welche Schritte man „von Hand“ durchführen müßte, um eine ablauffähige APRICOTS-Komponente zu erstellen. In diesem Abschnitt zeige ich, welche dieser Schritte dem Anwender bzw. Programmierer durch das APRICOTS User Interface abgenommen werden, und welche der Anwender nach wie vor selbst erledigen muß.

2.6.1 Service-Definition

Das Programmieren eines Service kann dem Service-Programmierer leider nicht abgenommen werden. Das User Interface dient in diesem Fall nur zur Verwaltung von bereits erstellten Services. Der Benutzer gibt hierzu den Namen eines Service an, den C-Funktions-Namen, den Pfad des Source-Codes[3] sowie eine Beschreibung des Service.

2.6.2 Resource Manager Erstellung

Bei der Resource Manager- oder Step Computation Server-Erstellung gibt es für den Anwender bzw. Programmierer nicht mehr viel zu tun. Nachdem er angegeben hat, ob ein Resource Manager oder ein Step Computation Server erstellt werden soll, gibt er einen Namen für diesen an. Zum Schluß muß noch der Service aus einer Liste mit allen verfügbaren Services ausgewählt werden, der in diesem Resource Manager bzw. Step Computation Server enthalten sein soll. Aus diesen Angaben, die in einer Datenbank gespeichert werden, wird später automatisch der Code für den Resource Manager bzw. Step Computation Server erzeugt. Der Anwender muß also nicht den Code „von Hand“ zusammensetzen oder irgendwelche Code-Stücke noch selbst programmieren.

2.6.3 Erzeugen der APRICOTS-Komponente

Auch bei der Erzeugung einer ablauffähigen APRICOTS-Komponente bleibt für den Anwender nicht mehr viel zu tun. Zuerst muß er einen Namen für die Komponente angeben, dann muß er aus einer Liste mit allen verfügbaren Resource Managern und Step Computation Servern diejenigen heraussuchen, die in der fertigen Komponente enthalten sein sollen. Auch diese Definition wird in einer Datenbank abgespeichert. Die ablauffähige Komponente wird dann zu einem späteren Zeitpunkt generiert. Würde man die ablauffähige Komponente sofort erstellen, müßte dies nach jeder durchgeführten Änderung erneut geschehen. Deshalb ist es günstiger wenn die Komponente erst unmittelbar vor dem Start generiert wird. Um die dazu notwendigen Definitionsdateien (siehe oben) muß sich der Anwender ebensowenig kümmern wie um den korrekten Aufruf von *make_apricot_component*.

[3] Der Service-Programmierer muß den fertigen C-Code zur Verfügung stellen. Sollte der Service-Code von einem Precompiler übersetzt werden müssen, so muß dies in der jetzigen Version der graphischen Oberfläche noch außerhalb gemacht werden (vgl. Abschnitt 6.5).

2.7 Zusammenfassung

Insgesamt läßt sich sagen, daß die graphische Oberfläche das Erstellen von APRICOTS-Komponenten sehr erleichtert. Der Anwender / Programmierer spart nicht nur Zeit, es werden auch mögliche Fehler bei der Erstellung und Programmierung der Resource Manager bzw. Step Computation Server vermieden, da dies automatisch geschieht. Allerdings kann das aufwendige Programmieren von Services – natürlich – nicht ebenfalls automatisiert werden.

Zum Abschluß dieses Kapitels folgt noch eine Übersicht über alle Schritte, die notwendig sind, um eine APRICOTS-Komponente von Hand bzw. mit dem User Interface zu erstellen.

Aktion	Schritte von Hand	Schritte mit dem APRICOTS User Interface
Service programmieren	Die in Abschnitt 2.1 vorgestellten Files (6 Stück) programmieren / erstellen	Die in Abschnitt 2.1 vorgestellten Files (6 Stück) programmieren / erstellen
Service verwalten	Programmierer muß sich selbst ein System für die Verwaltung überlegen	Eingabe der Definition in das User Interface, welches die Verwaltung übernimmt
Resource Manager definieren	Service aussuchen	Auswählen eines Service aus einer Liste aller verfügbaren Services
Resource Manager erstellen	Zusammenstellen / programmieren des Codes für den Resource Manager	automatisch
Resource Manager verwalten	Programmierer muß sich selbst ein System für die Verwaltung überlegen	Eingabe der Definition in das User Interface, welches die Verwaltung übernimmt
Step Computation Server definieren	Step aussuchen	Auswählen eines Steps aus einer Liste aller verfügbaren Steps
Step Computation Server erstellen	Zusammenstellen / programmieren des Codes für den Step Computation Server, insbesondere muß der Code für die Invarianten-Überprüfung geschrieben werden	automatisch
Step Computation Server verwalten	Programmierer muß sich selbst ein System für die Verwaltung überlegen	Eingabe der Definition in das User Interface, welches die Verwaltung übernimmt
Komponente definieren	Aussuchen der Resource Manager / Step Computation Server	Auswählen der Resource Manager / Step Computation Server aus einer Liste aller verfügbaren
Komponente erstellen	Definitionsfiles und Codefiles erstellen, Programm *make_apricot_component* mit Parametern starten	automatisch bei Start der Komponente

Tabelle 2.2: Erstellung einer APRICOTS-Komponente

Kapitel 3

ConTract Processing Repository

Bei der Arbeit mit der graphischen Oberfläche fallen sehr viele Daten an, die gesichert werden müssen. Beispielsweise müssen die Definitionen der Services, Resource Manager, Step Computation Server und der Komponenten gespeichert werden. Um die anfallenden Daten zu sichern, wurde das *ConTract Processing Repository* entworfen, eine „Sammlung“ mehrerer Relationen. Diesem Repository liegt das APRICOTS-Prozeßmodell zugrunde, in dem der Zusammenhang zwischen den statischen und dynamischen Einheiten abgebildet ist. Der für das ConTract Processing Repository wichtige Teil des APRICOTS-Prozeßmodells ist in Abbildung 2.3 zu sehen. In diesem Prozeßmodell werden die statischen Definitionen den zugehörigen Prozessen gegenübergestellt.

Im folgenden werden die wichtigsten Relationen kurz beschrieben.

3.1 Relationen für die statische Verwaltung

3.1.1 Komponenten Definition

Die bei der Definition einer Komponente anfallenden Daten werden in zwei Relationen abgespeichert. In der Relation *Component_Definitions* (siehe Tabelle 3.1) werden alle zu einer Komponente gehörenden Resource Manager und Step Computation Server gesichert. Die zweite Relation, *Component_Object_File*, enthält neben dem Namen der Komponente auch Pfadangaben für die Objekt-Code Datei der Komponente sowie für ein Icon, welches zur graphischen Repräsentation des späteren Komponenten-Prozesses verwendet wird.

Die Daten der Komponenten-Definition wurden deshalb in zwei Relationen aufgeteilt, da es möglich ist, daß eine Komponente mehrere Resource Manager und Step Computation Server enthält. Ohne diese Aufteilung hätten die beiden Pfadangaben (für die Objekt-Code Datei und das Icon) jeweils mit allen Kombinationen Komponente /

Component_Definitions		
Attribut	Typ	Erläuterung
Component_Name	char(60)	Eindeutiger Name der Komponenten-Definition
RM_Class_Name	char(60)	Klasse des Resource Managers

Tabelle 3.1: Relation *Component_Definitions*

Resource Manager gespeichert werden müssen. Das hätte unnötig viel Speicherplatz verschlungen, da jede Pfadangabe bis zu 256 Zeichen lang sein darf.

3.1.2 Resource Manager Definition

Die Daten von Resource Managern und Step Computation Servern werden in einer gemeinsamen Relation gespeichert. Zu jedem Resource Manager wird der zugehörige Service, eine Pfadangabe für ein Icon sowie ein Flag gespeichert (siehe Tabelle 3.2).

RM_Definition		
Attribut	Typ	Erläuterung
RM_Class_Name	char(60)	Name der Klasse des Reource Managers
Service_Name	char(60)	Name eines Dienstes, den diese Resource Manager Klasse implementiert
RM_Icon	char(255)	Pfadangabe für eine Bitmap zur graphischen Repräsentation dieses Resource Managers
RM_SCS	long	Flag: 0 falls RM, 1 falls SCS

Tabelle 3.2: Relation *RM_Definition*

Das Flag dient zur Unterscheidung, ob es sich um einen Resource Manager oder um einen Step Computation Server handelt. Das Icon dient – wie bei einer APRICOTS-Komponente – zur graphischen Repräsentation des Resource Managers.

Damit man zu einem späteren Zeitpunkt nicht die Relation abändern muß, ist nicht nur der Name des Resource Managers, sondern auch der Name des Service als Schlüssel implementiert. Somit ist es in späteren Versionen ohne Änderung der Relation möglich, mehrere Services einem Resource Manager zuzuordnen und dies in der Datenbank zu verwalten.

3.1.3 Service Definition

Die Daten einer Service Definition werden wieder in zwei Relationen gesichert. In der Relation *Service_Definition* werden alle Daten abgespeichert, die direkt mit dem Service zu tun haben. Neben dem Namen des Service ist das eine kurze Beschreibung des Service, eine Pfadangabe, unter der eine ausführliche Beschreibung eines Service zu finden ist, der Name der C-Funktion sowie eine Pfadangabe, wo der zugehörige Quell-Code abgespeichert ist (siehe Tabelle 3.3).

Service_Definition		
Attribut	Typ	Erläuterung
Service_Name	char(60)	Aufrufname des Dienstes
Description	char(255)	Kurze Beschreibung des Dienstes
Detailed_Description	char(255)	Pfadangabe / Aufrufkommando, um die Detailbeschreibung anzuzeigen
C_Fuction_Name	char(255)	Name der C-Funktion, wie im Code definiert
Source_Code	char(255)	Pfadangabe und Name des Quell-Codes
Service_Step	long	Flag: 0 Service, 1 Step — unbenutzt

Tabelle 3.3: Relation *Service_Definition*

In der Relation *Sub_Services* (Tabelle 3.4) sind die von einem Service benötigten Sub-Services aufgeführt. So ist es z. B. vor der Ausführung eines Service möglich, zu überprüfen, ob alle von diesem Service benötigten Sub-Services erreichbar sind. Für jedes Paar Service / Sub-Service gibt es genau einen Eintrag.

Sub_Services		
Attribut	Typ	Erläuterung
Service_Name	char(60)	Aufrufname des Dienstes
Required_Sub_Service	char(60)	Für diesen Dienst benötigter weiterer Dienst

Tabelle 3.4: Relation *Sub_Services*

3.2 Relation für die dynamische Verwaltung

In diesem Bereich gibt es nur eine Relation, die für die graphische Oberfläche von Bedeutung ist, die Relation *Component_Process* (Tabelle 3.5). In diese Relation erfolgt ein Eintrag, nachdem eine APRICOTS-Komponente gestartet wurde.

Beim Start einer Komponente bekomt diese einen logischen Namen vom Kommunikationssystem zugewiesen (siehe dazu [Laube92]). Soll diese Komponente von der graphischen Oberfläche aus angesprochen werden, ist das nur über diesen logischen Namen möglich. Deshalb muß die Verbindung zwischen dem Komponenten-Namen und dem logischen Komponenten-Namen gesichert werden.

In dieser Relation wird also zu jeder gestarteten Komponente der Komponenten Name, der zugehörige logische Name sowie der aktuelle Zustand der Komponente gespeichert. Wird eine Komponente gestoppt, wird der Eintrag aus der Relation wieder entfernt.

Component_Process		
Attribut	Typ	Erläuterung
Component_Name	char(60)	Eindeutiger Name der Komponenten-Definition
Log_Comp_Name	char(60)	Logischer Name der Komponente / Adresse
Component_State	long	Aktueller Zustand der Komponente

Tabelle 3.5: Relation *Component_Process*

3.3 Verwendung des Repositories

Zum einen dient das ConTract Processing Repository zur Speicherung der bei der Definition von Services, Resource Managern und Komponenten anfallenden Daten.

Anderseits kann das Repository aber auch dazu verwendet werden, bestimmte Zustände während des Ablaufs von APRICOTS-Komponenten zu sichern. Dazu ein Beispiel:

Ein Step benötigt oft verschiedene Sub-Services, um korrekt (bzw. überhaupt) ablaufen zu können. Beim Start einer Komponente ist es mit Hilfe des Repositories möglich, zu überprüfen, ob alle Sub-Services, die von einem Step der gestarteten Komponente benötigt werden, schon irgendwo in der Ablaufumgebung vorhanden sind. Siehe dazu auch Abschnitt 5.2.

Weiterhin ist es mit Hilfe des ConTract Processing Repositories leicht möglich, Fragen wie z. B. „welche Services sind in der Komponente x enthalten?“ oder „auf welchem Host läuft der Resource Manager *Maschine A*?“ zu beantworten.

Es gibt noch weitere wichtige Verwendungsmöglichkeiten des ConTract Processing Repositories. Da ist zum Beispiel die Nutzung des Repositories bei einem Recovery. Alle wichtigen Daten zur Herstellung des »alten« Zustands können aus dem Repository geholt werden. Ein anderes Beispiel ist das Protokollieren vieler Vorgänge im APRICOT-System. Es ist mit Hilfe des Repositories auch möglich, einen im Netz ablaufenden ConTract zu verfolgen.

Zum Abschluß dieses Kapitels folgt in der Tabelle 3.6 eine Zusammenfassung aller Relationen des ConTract Processing Repositories mit einer kurzen Beschreibung, wozu die Relationen in erster Linie verwendet werden.

Relation	Verwendung
Component_Definitions	Speicherung aller Resource Manager / Step Computation Server einer Komponente
Component_Object_File	Sonstige Daten einer Komponente
RM_Definition	Alle Daten einer Resource Manager / Step Computation Server Definition
Service_Definition	Alle wichtigen Daten eines Service
Sub_Services	Die von einem Service benötigten Sub-Services
Component_Process	Daten einer gestarteten APRICOTS-Komponente

Tabelle 3.6: Übersicht über die Relationen

Kapitel 4

Das APRICOTS User Interface

Die graphische Oberfläche dient dazu, Definition, Erstellung und Verwaltung von großen Client-Server orientierten Anwendungen zu unterstützen. Alle hierbei anfallenden Daten werden im ConTract Processing Repository gesichert. Die wichtigsten Funktionen der graphischen Oberfläche habe ich bereits in Kapitel 2 vorgestellt, das ConTract Processing Repository ist im letzten Kapitel ausführlich behandelt worden.

In diesem Kapitel stelle ich das APRICOTS User Interface etwas genauer vor. Zuerst gehe ich auf allgemeine Entwurfskriterien zur Gestaltung einer Mensch-Maschine Schnittstelle ein und zeige, welche Kriterien ich beim Entwurf der graphischen Oberfläche verwendet habe. Zum Schluß gibt es eine Übersicht über die Funktionen des User Interface sowie eine Beschreibung einzelner Fenster[1]. Da in dieser Ausarbeitung nicht alle Fenster ausführlich vorgestellt werden sollen, habe ich ein paar stellvertretend für die anderen ausgewählt. Eine genaue Beschreibung (Bedienungsanleitung) aller Fenster gibt es im Arbeitspapier [Zellmer92].

4.1 Entwurfskriterien einer graphischen Oberfläche

Für die Entwurfskriterien einer graphischen Oberfläche gibt es eine Reihe von Faktoren, die den Entwurf entscheident beeinflussen können. Da sind z. B. Randbedingungen, wie die verfügbare Rechnerumgebung oder die erwartete Funktionalität.

Ein weiterer wichtiger Einflußfaktor ist die vorhandene Hardware: welche Qualität haben die Datenendgeräte, gibt es ein Zeigegerät (eine Maus), ist eine graphische Datenausgabe möglich? Für eine graphische Darstellung ist oft, je nach Größe der Graphiken, eine hohe Rechnerleistung nötig, da sonst die Ausgabe zu langsam ist.

[1]= Frame, Window, Rahmen. Im APRICOTS User Interface gibt es für jede Funktion einen Fenster.

Außerdem muß die Übertragungskapazität zwischen Rechner und Terminal hoch genug sein, damit bei der Übertragung umfangreicher Daten keine allzugroßen Wartezeiten auftreten.

Weitere Einflußfaktoren sind z. B. die verwendbare Software oder die potentiellen Benutzergruppen der graphischen Oberfläche. All diese Dinge müssen beim Entwurf berücksichtigt werden, ist doch die Verwendung von Buttons nur dann sinnvoll, wenn auch eine Maus zum Bedienen dieser Buttons zur Verfügung steht.

Es folgt eine Auflistung einiger Design-Prinzipien einer Mensch-Computer Schnittstelle[2]:

- WYSIWYG – What you see is what you get
  Dieser Aspekt bedeutet im Zusammenhang mit einer graphischen Oberfläche, daß die externe (sichtbare) Darstellung eines Objekts immer mit der internen Darstellung übereinstimmen sollte. Durch den Zugriff auf ein Bildschirm-Objekt muß man immer auf das zugehörige interne Objekt zugreifen können.

- Darstellung der Aktionsmöglichkeiten
  Der Benutzer muß jederzeit erkennen können, welche Aktionen möglich sind, d. h. nicht verwendbare Aktionen sollten entweder gar nicht oder andersartig (z. B. schattiert) dargestellt werden.

- Flexible Bearbeitungsmöglichkeiten
  Der Benutzer sollte z. B. die Wahl haben, ob er seine Aktionen mit der Maus oder lieber mit der Tastatur durchführt. Auch sollte er die Möglichkeit haben, am Bildschirm schon dargestellte Objekte mit der Maus zu selektieren, anstatt sie erneut mit der Tastatur eingeben zu müssen.

- Verringerung der Komplexität
  Es sollten nicht immer alle möglichen Informationen auf einmal dargestellt werden, da sonst die Darstellung unübersichtlich wird. Es ist besser, wenn der Benutzer weitere ihm wichtig erscheinende Daten auf Abruf dargestellt bekommt.

- Vermeidung von Verdrängung
  Ausgaben vom System sollten schon vom Benutzer gemachte Eingaben nicht überschreiben. Für Ausgaben sollte deshalb ein spezieller Bereich bzw. ein extra Fenster verwendet werden.

- Benutzer-Aktion auf Abruf
  Das System sollte vom Benutzer gesteuert werden und nicht der Benutzer vom System.

- Hilfsfunktionen
  Dem Benutzer sollten an jeder Stelle diverse Hilfsfunktionen zur Verfügung stehen.

[2]Siehe dazu auch [CaSeCa86], [Schnei86] und [Sommer87].

- Gute Einteilung der Aktionen
  Eine schlechte Einteilung der Funktionen macht nicht nur die Anwendung unübersichtlich, es können auch leichter Fehler gemacht werden.

- Sinnvolle Fehlermeldungen
  Fehlermeldungen sollten den Benutzer nicht verwirren, sie sollten ihm Unterstützung bieten, ihm einen Hinweis darauf geben, warum der Fehler aufgetreten und wie er zu beheben ist.

Das Ziel einer graphischen Oberfläche ist es, den Benutzer zu unterstützen und ihm zu einem „angenehmen“ Arbeiten zu verhelfen.

4.2 Verwendete Kriterien bei der graphischen Oberfläche

Von den oben vorgestellten Kriterien sind die Punkte *Flexible Bearbeitungsmöglichkeiten* und *Hilfsfunktionen* noch nicht implementiert worden. Alle anderen Kriterien sind so weit wie möglich bei der Gestaltung der graphischen Oberfläche berücksichtigt worden. So ist zum Beispiel die Verwendung eines Objekts, zum Beispiel einer Komponente, nicht möglich, wenn sie nicht im ConTract Processing Repository vorhanden ist. Dabei ist auch sichergestellt, daß das dargestellte Objekt den gleichen Aufbau hat, wie das im Repository gespeicherte. (siehe Punkt *wysiwyg*, interne Darstellung ist gleich der externen). Der Punkt *Vermeidung von Verdrängung* ist z. B. so implementiert worden, daß Ausgaben der graphischen Oberfläche an den Benutzer immer in einem extra erscheinenden Fenster dargestellt werden.

Ich habe bei der Gestaltung der graphischen Oberfläche in erster Linie auf ein einheitliches Aussehen der einzelnen Fenster (Frames, Windows) sowie auf eine einheitliche Bedienung geachtet. Allerdings mußten dabei die Möglichkeiten von Ingres / Windows 4GL ([Ing90/1]) beachtet werden, da die graphische Oberfläche unter diesem System implementiert werden sollte. In Abbildung 4.1 ist ein typisches Fenster des APRICOTS User Interface abgebildet.

Im oberen Teil eines Fensters steht immer der Funktionsname. Er wird durch einen horizontalen Balken vom restlichen Teil des Fensters abgetrennt. Der linke Teil wird immer von einer Tabelle (Ausgabeliste) – soweit vorhanden – eingenommen. Im rechten Teil befinden sich dann die Buttons zur Funktionsausführung sowie eventuell nötige Eingabestrings.

Cancel-, Quit- und OK-Button befinden sich immer rechts oben, unterhalb des Abgrenzungsbalkens. Der Cancel-Button dient grundsätzlich zum Abbrechen der momentanen Funktion, der OK-Button dient zum Durchführen derselben. Der Quit-Button ist in den Fenstern vorhanden, die nur Auswahlmöglichkeiten zur Verfügung

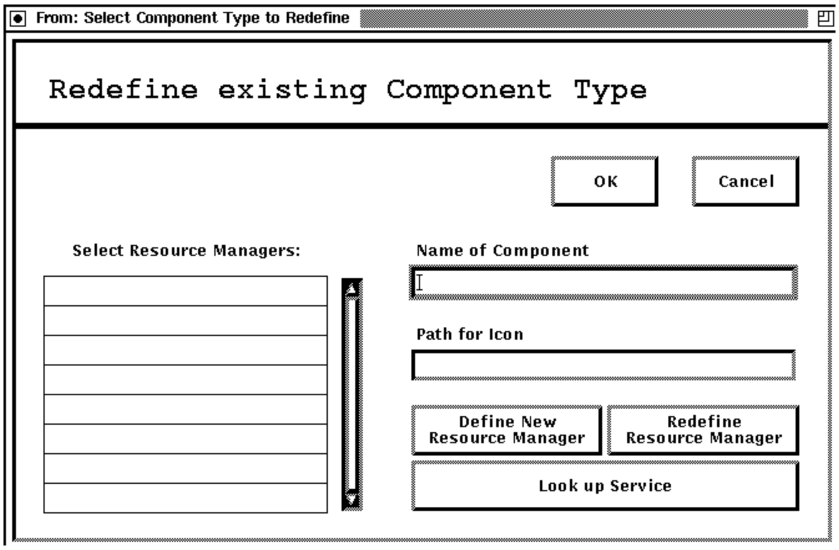

Abbildung 4.1: Typisches Fenster des APRICOTS User Interface

stellen (Fenster mit einem Auswahlmenü). Er ersetzt in diesen Fenstern den Cancel-Button und dient zum Verlassen der Fenster bzw. zum Beenden des APRICOTS User Interface.

Für jede Funktion des APRICOTS User Interface wird ein eigenes Fenster verwendet (zur Erläuterung der einzelnen Funktionen siehe Abschnitt 4.3). Zur Bedienung der Fenster möchte ich an dieser Stelle nur sagen, daß durch Anklicken eines Buttons eine Aktion durchgeführt wird. Viele Aktionen benötigen vor ihrer Durchführung ein Objekt aus der jeweiligen Tabelle, welches ebenfalls per Mausklick ausgewählt werden kann. Momentan nicht verwendbare Buttons werden schattiert dargestellt. Weitere Bedienungshinweise sind in [Zellmer92] zu finden.

4.3 Übersicht über die Funktionen

Nachdem dem Start des APRICOTS User Interface befindet man sich im Hauptmenü (Abbildung 4.2).

Vom Hauptmenü aus kann man folgende Untermenüs aufrufen:

- Static Configuration Management (Statische Konfigurationsverwaltung)
- Dynamic Configuration and Process Management (Dynamische Konfigurations- und Prozeß-Verwaltung)
- ConTract-Management (Ablaufsteuerung von ConTracts; noch nicht implementiert)

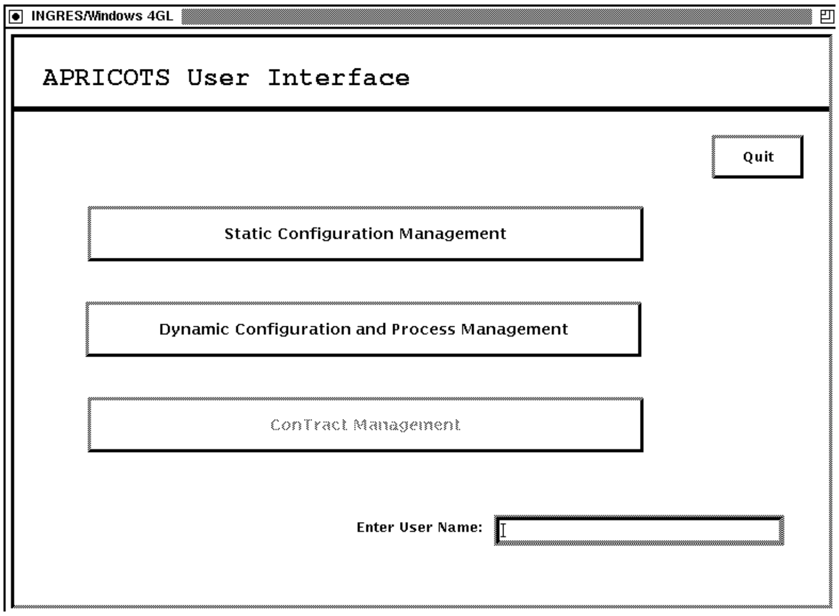

Abbildung 4.2: Hauptmenü des APRICOTS User Interface

4.3.1 Statische Konfigurationsverwaltung

Die statische Konfigurationsverwaltung umfaßt Funktionen zum Anschauen, Eintragen, Ändern und Löschen von Service-, Resource Manager- und APRICOTS-Komponenten-Definitionen. Hier ist es auch möglich, eine sogenannte Konfiguration anzugeben. Das ist eine Liste mehrerer APRICOTS-Komponenten, zu denen jeweils ein Rechnername sowie eine Anzahl von Instanzen mit angegeben wird. Beim Starten einer Konfiguration werden x Instanzen der angegebenen APRICOTS-Komponente auf dem angegebenen Rechner gestartet. Das erspart mühevolle Kleinarbeit.

Im folgenden stelle ich nun ein paar Funktionen (Fenster) der statischen Konfigurationsverwaltung etwas genauer vor.

Definieren eines neuen Resource Managers

(siehe Abbildung 4.3)

Um einen neuen Resource Manager zu definieren, muß man zuerst angeben, ob ein „normaler“ Resource Manager oder ein Step Computation Server[3] erstellt werden soll. Dann muß man den Namen des neuen Resource Manager angeben. An dieser Stelle wird überprüft, ob dieser Name bereits existiert. Sollte dies der Fall sein, wird er zurückgewiesen.

[3]Ein Step Computation Server kann als ein spezieller Resource Manager angesehen werden. Deshalb gibt es für die Definition auch nur eine einzige Funktion

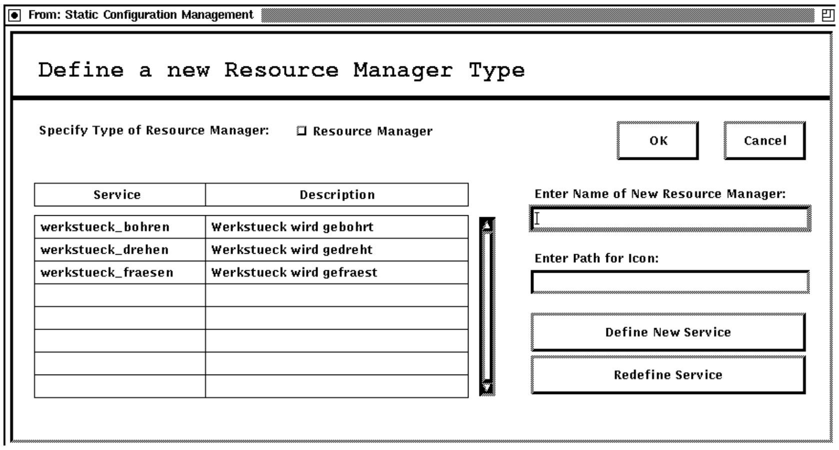

Abbildung 4.3: Definieren eines neuen Resource Managers

Wenn ein Icon für den Resource Manager vorhanden ist, kann man den Pfadnamen des Icons mit angeben. Das Icon dient im späteren Ablauf der Anwendung zur graphischen Repräsentation des Resource Managers. Bevor die Funktion durchgeführt werden kann, muß man noch einen Service angeben, der in dem Resource Manager bzw. Step Computation Server enthalten sein soll.

Von diesem Fenster aus gibt es auch die Möglichkeit, die Funktionen zum Erstellen bzw. Ändern einer Service Definition aufzurufen. Das kann mitunter recht sinnvoll sein, wenn ein Service nicht der Erwartung des Anwenders entspricht.

Ändern eines Service

(siehe Abbildung 4.4)

Beim Aufruf dieser Funktion sind alle Eingabefelder bereits vorbelegt, sofern in der Datenbank ein Eintrag dafür vorhanden ist. Es gibt folgende Eingabefelder:

- Service-Name
  An diese Stelle gehört der logische Name des Service.
- Description
  Eine kurze Beschreibung, was der Service macht.
- C-Function-Name
  Name (Aufruf) der C-Funktion, so wie er im Source Code enthalten ist.
- File with Detailed Description
  Zeiger auf ein File mit der ausführlichen Beschreibung der Services. Dieses File kann von dieser Funktion aus angeschaut und auch editiert werden.

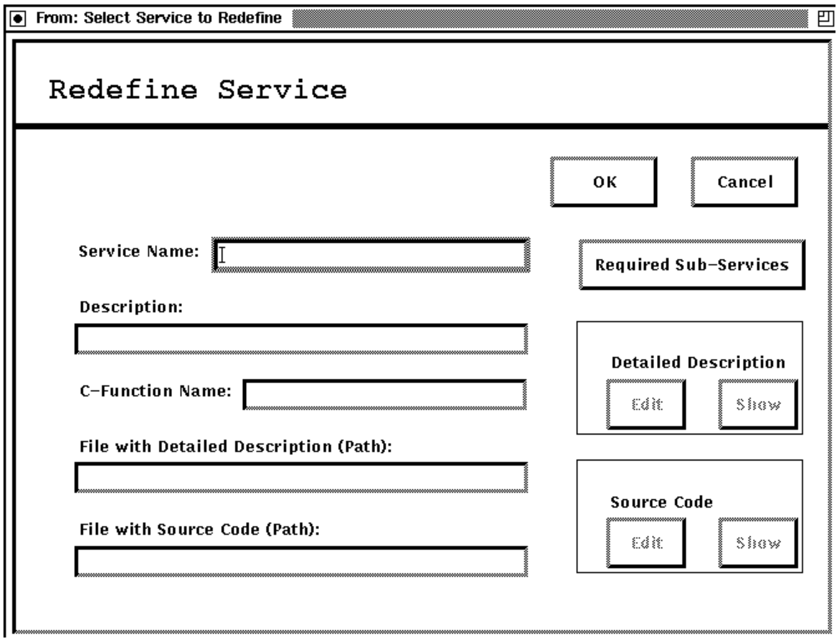

Abbildung 4.4: Ändern eines Service

- File with Source Code
  Zeiger auf den Source Code des Service. Auch dieses File kann angeschaut und editiert werden.

Weiterhin kann man über den Button *Required Sub-Service* alle von diesem Service benötigten Sub-Services angeben. Dazu wird ein weiterer Fenster dargestellt.

Für das anzugebende Quell-Code File gibt es noch eine wichtige Voraussetzung: Es muß das fertige C-File sein, d. h. es muß, sofern notwendig, außerhalb der Oberfläche von einem Precompiler übersetzt werden. Das ist zum Beispiel dann notwendig, wenn in dem Service Datenbankanfragen gemacht werden (mittels Embedded SQL). In diesem Fall muß der Service von dem zur Datenbank gehörenden Precompiler übersetzt werden, ehe ein für den C-Compiler verständlicher Code vorliegt.

Wie ich bereits in Kapitel 2 dargestellt habe, werden zu dem eigentlichen C-File des Service noch fünf weitere Files benötigt (z. B. das File `<service_name>_decode.c`). Diese Files kann man zur Zeit noch nicht vom APRICOTS User Interface aus erstellen bzw. editieren. Eine solche Möglichkeit wird aber in Kürze implementiert werden.

4.3.2 Dynamische Konfigurations- und Prozeß-Verwaltung

In diesem Teil der graphischen Oberfläche geht es hauptsächlich um die Ablaufsteuerung von APRICOTS-Komponenten. Von diesem Menü aus kann man die Funktionen zum Starten, Stoppen und Migrieren der Komponenten aufrufen. Ein weiterer

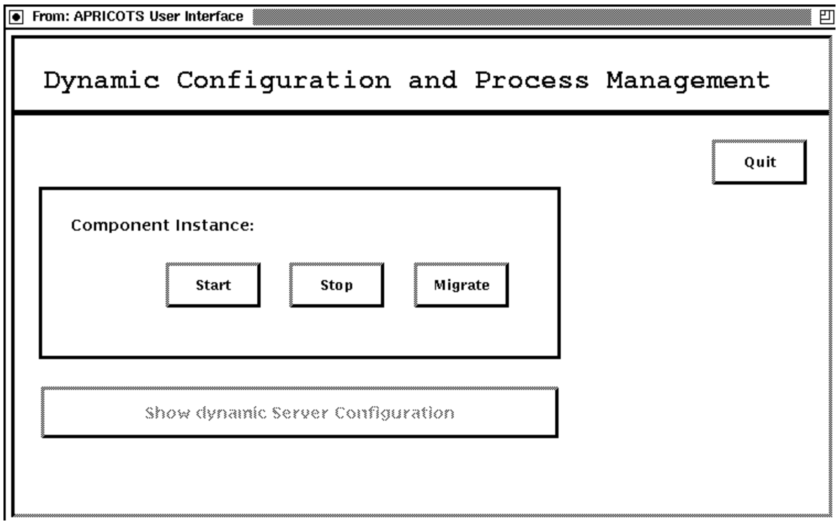

Abbildung 4.5: Menü für die dynamische Konfigurations- und Prozeß-Verwaltung

Menüpunkt führt zur Anzeige einer dynamischen Server-Konfiguration, d. h. Anzeigen aller momentan laufenden Komponenten, Resource Manager und Step Computation Server sowie deren aktueller Zustand. Vorläufig ist es nur möglich, eine fest vorgegebene Konfiguration anzuzeigen. In einer späteren Version soll es möglich sein, jede beliebige Konfiguration darzustellen.

Starten einer APRICOTS-Komponente

(siehe Abbildung 4.6)

In der im linken Teil des Fensters stehenden Liste sind alle verfügbaren APRICOTS-Komponenten aufgelistet. Hat man eine davon ausgewählt sowie einen Rechnernamen in das Eingabefeld auf der rechten Seite eingetragen, kann man mittels *OK* diese Komponente starten.

Mit dem Button *Look up Component Resource Managers* kann man alle Resource Manager und Step Computation Server anschauen, die in der ausgewählten Komponente enthalten sind (siehe dazu Abbildung 4.7). Von diesem Fenster aus ist es dann möglich, sich den in den Resource Managern bzw. Step Computation Servern enthaltenen Service genauer anzusehen.

Stoppen einer Komponente

(Abbildung 4.8)

Dieser Frame ähnelt dem Frame zum Starten einer Komponente, allerdings fehlt der Eingabe-String. In der Tabelle auf der linken Seite sind alle zum jetzigen Zeitpunkt laufenden Komponenten aufgelistet, zusätzlich dazu wird der Name des Rechners

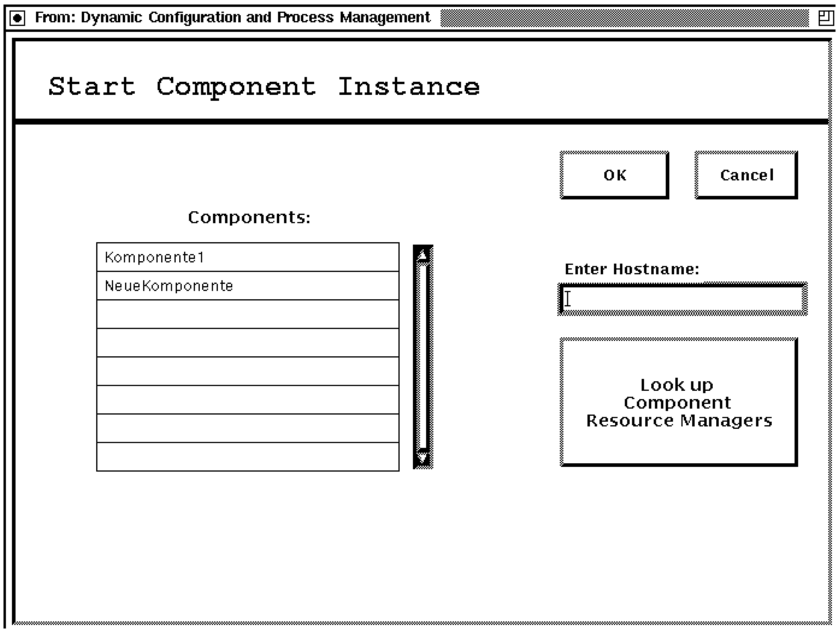

Abbildung 4.6: Starten einer Komponente

angezeigt, auf dem die Komponente läuft. Stoppen kann man eine APRICOTS-Komponente, indem man sie durch einen Mausklick anwählt und danach den *OK*-Button betätigt. Die beim Stoppen einer Komponente auftretenden Probleme werden in Abschnitt 5.3 diskutiert.

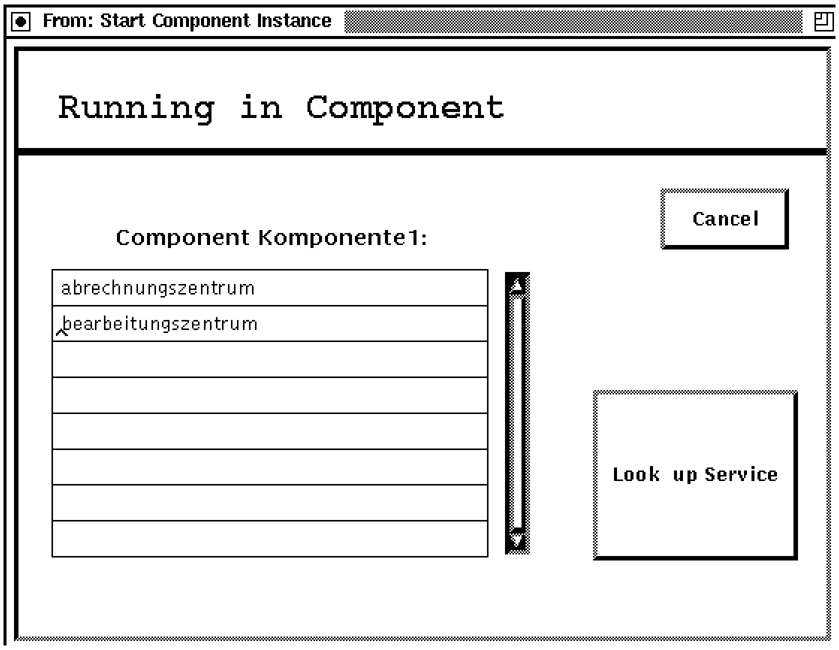

Abbildung 4.7: In einer Komponente enthaltene Resource Manager

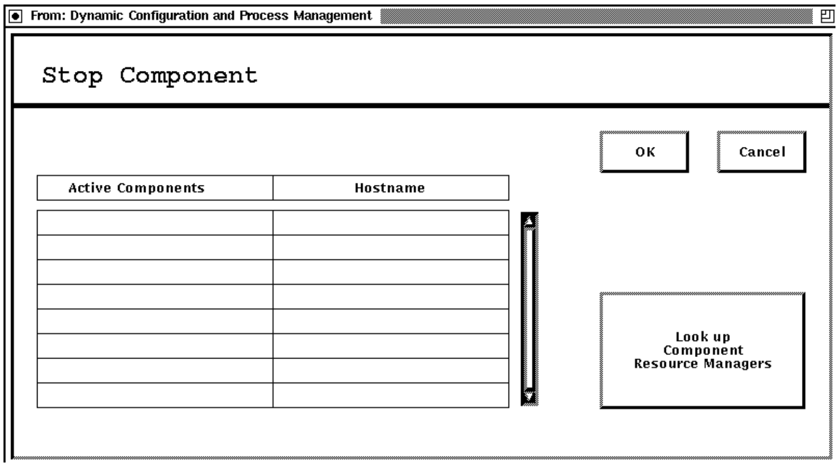

Abbildung 4.8: Stoppen einer Komponente

Kapitel 5

Diskussion

Bei der Definition und Implementierung der graphischen Oberfläche sind einige Probleme aufgetreten, für die es zum Teil verschiedene Lösungsmöglichkeiten gibt. Jeder dieser Lösungen hat Vorteile, aber zum Teil auch erhebliche Nachteile. Ich werde in diesem Kapitel die wichtigsten dieser zum Teil noch offenen Probleme vorstellen, mögliche Lösungswege aufzeigen sowie die Vor- und Nachteile dieser Lösungswege aufführen.

5.1 Änderung einer Service-Definition

Hin und wieder kommt es vor, daß der Service-Programmierer einen Service ändern will, sei es, weil er eine Verbesserung vornehmen möchte oder weil bei der Ausführung des Service ein Fehler aufgetreten ist. Allerdings darf ein Service nicht so ohne weiteres geändert werden, hierbei gibt es einiges zu beachten.

5.1.1 Änderung des Service-Code

Will der Service-Programmierer den Code eines Service (hierzu gehört auch der *c-function-name*) ändern, kann es zu Problemen kommen, wenn der Service Teil einer gestarteten Komponente ist. Die Art des Problems hängt von der Frage ab, warum der Service-Code geändert werden soll.

- **Änderung wegen einem Programmierfehler:**
  In diesem Fall sollte die Änderung auch in der laufenden Komponente vorgenommen werden. Dies ist aber zur Zeit nur möglich, indem man die Komponente stoppt (abbricht), neu zusammenbaut und dann wieder startet. Jedoch kann eine aktive Komponente nicht so ohne weiteres abgebrochen werden.

- **Änderung des logischen Ablaufs:**
  Soll ein momentan aktiver Service (bzw. eine aktive Anwendung) von einer solchen Änderung nicht betroffen werden, muß die laufende Komponente nicht geändert werden. Problem: Muß diese Komponente aus irgendeinem Grund neu gestartet werden (z. B. Rechnerabsturz → Recovery wird notwendig), kann sie nicht mehr in ihrem ursprünglichen Zustand gestartet werden, da die Definition des Service nicht mehr dieselbe ist.

Ein weiteres Problem tritt auf, wenn der Name eines Service, der *service-name* geändert werden soll.

5.1.2 Änderung des Service-Name

Ruft der ConTract-Manager einen Service bzw. ein Service einen Sub-Service auf, so wird der aufgerufene Service immer über seinen Namen angesprochen. Will nun der Service-Programmierer – warum auch immer – den Namen eines Service ändern, müßten alle Files (genauer der Skript-Code sowie der C-Code von Services), in denen dieser Service-Name vorkommt, ebenfalls geändert werden. Macht man das nicht, wird es vom Kommunikationssystem eine Fehlermeldung »unbekannter Service« geben, sobald dieser Service aufgerufen werden soll. Weiterhin müssen auch alle Tupel der Relationen des ConTract Processing Repository geändert werden, in denen der geänderte Service-Name vorkommt. Insbesondere betrifft das die Relation der Resource Manager Definition sowie die Relation, in der die Sub-Services gespeichert sind.

So gesehen erscheint es fast sinnvoll, eine Änderung einfach zu verbieten. Allerdings wäre es dann nicht mehr möglich, einen Tippfehler auszubessern, wenn die Service-Definition einmal abgeschlossen und gespeichert ist. Eine mögliche Alternative zu dem Änderungsverbot wäre es, eine entsprechende Warnmeldung anzuzeigen, wenn jemand den Namen eines Service ändern will.

5.1.3 Wann darf eine Service-Definition geändert werden?

Für die Beantwortung dieser Frage gibt es mehrere Möglichkeiten. Die einfachste Lösung wäre die, die Änderung von Service-Definitionen zu verbieten. Mit Sicherheit wäre das aber auch die schlechteste Lösung, könnte man doch keinerlei Fehler mehr verbessern. Eine andere Lösungsvariante wäre es, davon auszugehen, daß, wenn eine Änderung in einem Service-Code vorgenommen wird, diese Änderung auch auf jeden Fall in einer laufenden Komponente duchgeführt wird. Bei dieser Lösung entfällt das Problem des Recoveries, die Komponente muß aber trotzdem gestoppt werden, damit die Änderung eingebunden werden kann. Hier wäre es optimal, wenn die Änderung auch bei einer laufenden (und aktiven) Komponente durchgeführt werden könnte.

Die von mir für die graphische Oberfläche gewählte Lösung ist die, Änderung an der Service-Definition nur dann zuzulassen, wenn der zu ändernde Service in keiner laufenden Komponente enthalten ist. Dadurch entfällt das Recovery-Problem, und es muß auch keine Komponente abgebrochen werden, um die Änderung durchzuführen.

Ähnliche Probleme wie die hier für die Service-Definition geschilderten treten auf, wenn die Definition eines Resource Manager oder gar die Definition einer Komponente selbst geändert werden soll. Insbesondere bei einem Recovery können große Schwierigkeiten entstehen. Aus diesem Grund habe ich mich auch hier dazu entschieden, eine Änderung nur dann zuzulassen, wenn der zu ändernde Resource Manager in keiner laufenden Komponente enthalten ist, bzw. die zu ändernde Komponente nicht gestartet ist.

5.2 Benötigte Services in einer laufenden Umgebung

In diesem Abschnitt geht es um die Frage, was passiert, wenn ein benötigter Service nirgendwo in der laufenden Umgebung (= alle gestarteten Komponenten, ConTract Manager, etc.) vorhanden ist. Diese Frage ist schnell beantwortet. Wird ein nicht vorhandener Service aufgerufen, wird dies vom Kommunikationssystem mit der Fehlermeldung »unbekannter Service« quittiert. Daraus ergibt sich die Frage, wie man sicherstellen kann, daß so etwas nicht vorkommt.

Zum einen gibt es die Möglichkeit, schon beim Start einer Komponente darauf zu achten, daß alle Sub-Services, die von der in der Komponente enthaltenen Services[1] benötigt werden, irgendwo in der laufenden Umgebung vorhanden sind. Ob es einen Sub-Service gibt, der nicht in der laufenden Umgebung zu finden ist, kann man ohne großen Aufwand feststellen. Findet man so einen Sub-Service, gibt es verschiedene Möglichkeiten, diese Information zu verarbeiten:

1. Zuerst Ausgabe einer Liste mit den nicht vorhandenen Sub-Services, dann den Benutzer entscheiden lassen, was gemacht werden soll.
2. Automatisches Starten einer Komponente, die den benötigten Sub-Service enthält (soweit eine solche Komponente vorhanden ist).
3. Automatisches Erstellen einer Komponente (und eventuell eines dazu notwendigen Resource Manager) mit dem benötigten Sub-Service.

Inwieweit eine dieser Möglichkeiten überhaupt sinnvoll erscheint, ist eine andere Frage, z. B. besteht die Möglichkeit, daß der Benutzer sowieso eine weitere Komponente

[1] Eigentlich sind die Services in Resource Managern bzw. Step Computation Servern enthalten, aus denen sich dann die Komponente zusammensetzt.

starten wollte, in der dieser benötigte Sub-Service enthalten ist. In diesem Fall würde eine Meldung oder gar das automatische Starten eher stören.

Eine andere Möglichkeit wäre die, beim Start eines ConTracts zu überprüfen, ob alle von dem ConTract benötigten Services in der laufenden Umgebung vorhanden sind (einschließlich eventuell benötigter Sub-Services). Diese Aufgabe müßte entweder direkt von der ConTract-Verwaltung[2] oder vom ConTract-Manager übernommen werden. Letzteres würde einen regen Nachrichtenaustausch zwischen dem ConTract-Manager und der graphischen Oberfläche nach sich ziehen. Die Möglichkeiten, die es gibt, sollte ein Service gefunden werden, den es nicht in der laufenden Umgebung gibt, sind dieselben wie oben, also Ausgabe an den Benutzer oder automatisches Starten bzw. Erstellen einer Komponente (soweit das möglich ist).

Ein weiterer möglicher Zeitpunkt eine entsprechende Maßnahme einzuleiten wäre der, wenn vom Kommunikationssystem die Meldung kommen würde: »unbekannter Service«. Welcher Teil des Gesamtsystems in so einem Fall reagieren sollte, müßte sorgfältig abgewägt werden. Zum einen gäbe es da das Kommunikationssystem, daß eine Meldung an die graphische Oberfläche schicken könnte, wenn ein ihm nicht bekannter Service verlangt wird. Auf der anderen Seite steht der ConTract-Manager, der nach solch einer Fehlermeldung eine entsprechende Nachricht absenden könnte. In diesem Fall müßte aber auch ein Service, der Sub-Services aufruft, in der Lage sein, eine Nachricht abzusenden, wenn er diese Fehlermeldung (»unbekannter Service«) zurückbekommt. Bisher ist allerdings weder in der graphischen Oberfläche noch beim Kommunikationssystem oder beim ConTract Manager eine Schnittstelle vorgesehen, über die es dann möglich wäre, eine Komponente mit einem fehlenden Service zu starten.

Die bisher beschriebenen Möglichkeiten dienen alle dazu, zu verhindern, daß eine Anwendung (ein ConTract) wegen einem Service abgebrochen wird, der nicht in der laufenden Umgebung zu finden ist. Allerdings nützt keine dieser Maßnahmen etwas, wenn zum Beispiel ein benötigter Service noch gar nicht programmiert wurde. Insofern bleibt es doch zumindest teilweise in der Verantwortung des Benutzers, seine Anwendung so zu erstellen, daß alles reibungslos ablaufen kann. Deshalb sollte die graphische Oberfläche den Benutzer soweit unterstützen, daß Warnungen und Hinweise auf fehlende, noch zu erstellende Programmteile ausgegeben werden.

5.3 Stoppen einer Komponente

Soll eine APRICOTS-Komponente gestoppt werden, darf diese zum einen nicht in diesem Moment aktiv sein, zum anderen müssen alle in dieser Komponente begonnenen Transaktionen abgeschlossen sein. Dies sicherzustellen ist nicht die Aufgabe

[2] Der Teil der graphischen Oberfläche, der das Starten, Stoppen, etc. von ConTracts und deren Darstellung (Visualisierung) übernimmt. Dieser Teil gehört allerdings nicht mehr in das Aufgabengebiet dieser Studienarbeit.

der graphischen Oberfläche. Ein möglicher „Aufpasser“ wäre das Kommunikationssystem, da dieses die Verteilung der Service-Aufrufe unter sich hat sowie auch das Stoppen einer Komponente durchführt. Die graphische Oberfläche übermittelt nur die Aufforderung, eine Komponente zu stoppen.

5.4 Gestaltung der Oberfläche

Bei dem Entwurf der graphischen Oberfläche kam die Frage auf, ob es benutzerfreundlicher ist, die Oberfläche funktionsorientiert zu gestalten oder ob vielleicht eine objektorientierte Gestaltung der Oberfläche Vorteile hat. Im folgenden erkläre ich kurz, was eine funktionsorientierte bzw. eine objektorientierte Gestaltung der Oberfläche ist, dann stelle ich ihre Vor- und Nachteile gegenüber.

5.4.1 Funktionsorientierter Aufbau der Oberfläche

Funktionsorientiert heißt, kurz gesagt, man wählt zuerst die Funktion aus, die man benutzen möchte und dann erst das Objekt, auf dem diese Funktion angewendet werden soll.

Beispiel: Der Benutzer der Oberfläche will eine Komponente starten. Also klickt er zuerst den Button *Starten* an. Aus der dann in einem neuen Fenster erscheinenden Liste, in der alle Objekte des Typs *Komponente* aufgelistet sind, wählt er die zu startende Komponente aus. Siehe dazu Abbildung 5.1.

Der grundlegende Aufbau der funktionsorientierten Oberfläche ist also der: Man hat ein Fenster, in dem die verschiedenen (möglichen) Funktionen als Buttons ausgewählt werden können. Klickt man nun einen dieser Buttons an, erscheint ein neues Fenster, in dem eine Liste der Objekte angezeigt wird, auf die man die zuvor gewählte Funktion anwenden kann.

5.4.2 Objektorientierter Aufbau der Oberfläche

Objektorientiert bedeutet, daß man zuerst das zu benutzende Objekt auswählt, dann erst die darauf anzuwendende Funktion.

Beispiel: *Stoppen einer Komponente*. In dem Fenster, in dem diese Funktion ausgewählt werden kann, gibt es eine Liste, in der alle gespeicherten Komponenten enthalten sind (Vergleiche Abbildung 5.2). Weiterhin existieren Buttons z. B. für *starten*, *stoppen*, *cancel*, etc. Zuerst sind alle Buttons außer dem *Cancel*-Button schattiert dargestellt, da deren Funktion nur auf ein ausgewähltes Objekt angewendet werden kann. Erst nachdem der Benutzer ein Objekt aus der Liste ausgewählt hat, sind diejenigen Buttons verwendbar, deren Funktion auf dieses Objekt angewendet werden kann.

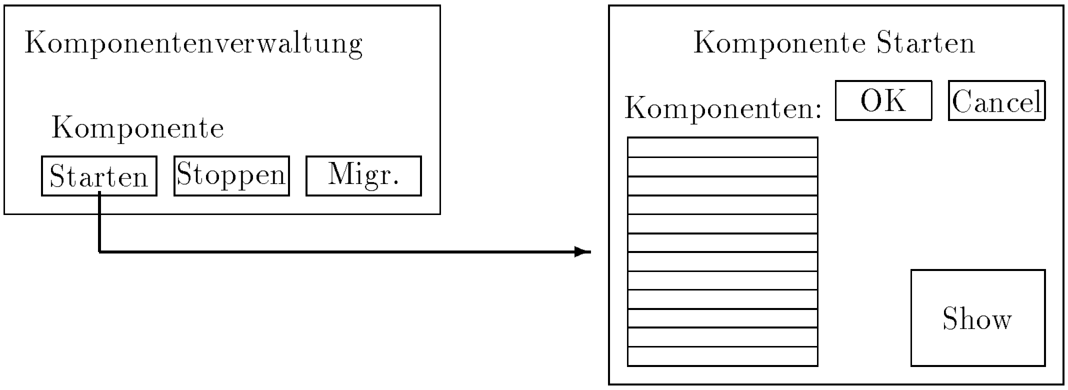

Abbildung 5.1: Funktionsorientierte Oberfläche

Um eine Komponente stoppen zu können, muß der Benutzer eine Komponente auswählen, deren aktueller Zustand »active« ist, also eine Komponente, die zuvor einmal gestartet und bis jetzt nicht mehr abgebrochen wurde. Insofern wäre es also sinnvoll, in der Objektliste auch den Zustand der Komponente[3] mit anzuzeigen.

5.4.3 Vergleich funktions- / objektorientierte Oberfläche

Ein Vorteil einer funktionsorientierten Oberfläche ist es, daß durch die Wahl der auszuführenden Funktion eine Vorauswahl der darzustellenden Objekte getroffen werden kann. Möchte der Benutzer z. B. eine Komponente stoppen, werden nur diejenigen Komponenten angezeigt, die gerade in der laufenden Umgebung vorhanden sind. Durch die zum Teil wesentlich geringere Anzahl dargestellter Objekte ist die funktionsorientierte Oberfläche in diesem Fall übersichtlicher.

Dieser Vorteil der funktionsorientierten Oberfläche ist zugleich der größte Nachteil der objektorientierten Oberfläche. Bei einer größeren Anwendung kann die Liste der Objekte mitunter recht lang werden, es sei denn, der Benutzer gibt das Objekt direkt an, das er manipulieren will. Allerdings würde das den Komfort mindern, da ein einfaches Anklicken des Objekts so nicht möglich wäre.

Weiterhin muß der Benutzer immer darauf achten, welche Funktionen er auf sein ausgewähltes Objekt anwenden darf. Selbst wenn die nicht verwendbaren Funktionen z. B. schattiert dargestellt werden, ist doch ein gewisses Maß an Aufmerksamkeit gefordert, damit sich der Anwender nicht andauernd „verklickt“, da sich die verschiedenen Darstellungen am Bildschirm (zumindest unter Ingres / Windows 4GL) oft nur schwer unterscheiden lassen.

[3] Natürlich ist das nur in diesem Beispiel so.

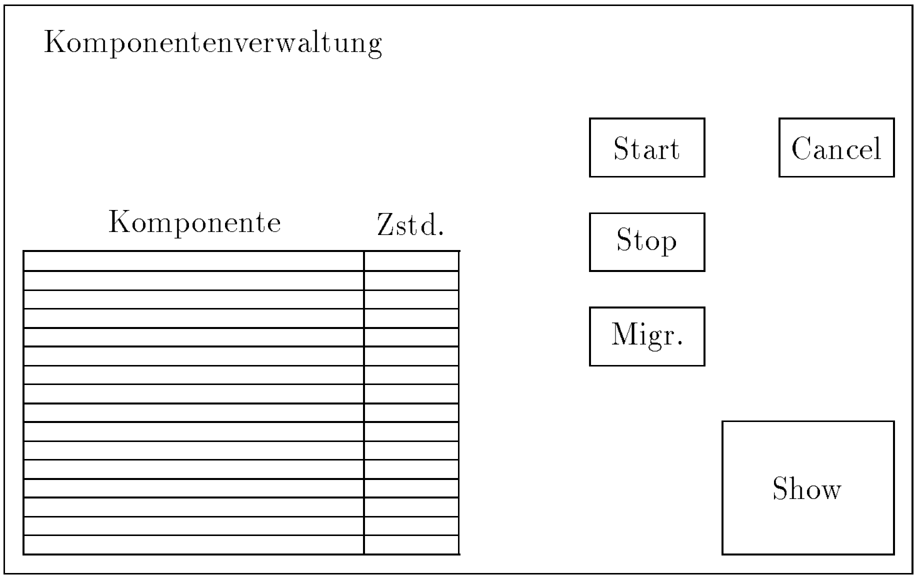

Abbildung 5.2: Objektorientierte Oberfläche

Beim objektorientierten Aufbau der Oberfläche werden weniger Fenster benötigt wie beim funktionsorientierten Aufbau. Nimmt man als Beispiel die Komponentenverwaltung (Starten / Stoppen / Migrieren einer Komponente), so wird beim objektorientierten Ansatz nur ein Fenster benötigt, während man bei der funktionsorientierten Lösung neben dem Fenster zur Funktionsauswahl noch jeweils ein Fenster für jede Funktion benötigt (in diesem Beispiel also insgesamt vier). So wird bei der funktionsorientierten Oberfläche nicht nur mehr Speicherplatz auf der Festplatte und im Hauptspeicher benötigt, sondern es ist auch mehr Verwaltungsaufwand[4] nötig, der die Geschwindigkeit der Anwendung herabsetzt.

Möchte der Benutzer oft zwischen den möglichen Funktionen wechseln, muß er bei der funktionsorientierten Oberfläche immer zwischen den einzelnen Fenstern hin- und her schalten, was doch ziemlich viel Zeit kosten kann und eher unpraktisch ist.

In der Tabelle 5.1 sind nochmals alle Vor- und Nachteile kurz zusammengefaßt.

Abschließend kann man sagen, daß beide Ansätze Vor- und Nachteile haben. Ich habe mich bei meiner Arbeit für den funktionsorientierten Ansatz entschieden. Für mich war der Vorteil ausschlaggebend, daß immer nur diejenigen Objekte angezeigt werden, die von der ausgesuchten Funktion verwendet werden können. Auch ein theoretisches Durchspielen beider Ansätze ergab, daß die wenigen Vorteile des funktionsorientierten Ansatzes überwiegen.

[4] Z. B. ist ein Fensteraufruf mehr zu tätigen.

	Funktionsorientierte Oberfläche	Objektorientierte Oberfläche
Vorteile	Vorauswahl der dargestellten Objekte weniger Objekte dargestellt Übersichtlicher	Man sieht auf einen Blick, welche Funktion für ein bestimmtes Objekt möglich ist Schnelleres und flüssigeres arbeiten, wenn oft zwischen den Funktionen gewechselt wird
Nachteile	Benutzer muß zuerst die Funktion wählen, dann das Objekt Mehr Bedienungsaufwand Mehr Fenster notwendig, deshalb mehr Speicherplatz und mehr Verwaltungsaufwand nötig Bei jedem Wechsel der Funktion umschalten auf anderes Fenster notwendig	Nicht für alle angezeigten Objekte kann jede angebotene Funktion verwendet werden Unübersichtlich Viele Objekte auf einmal dargestellt

Tabelle 5.1: Vor- und Nachteile von funktionsorientierten und objektorientierten Oberflächen

5.4.4 Eine weitere Lösungsmöglichkeit für die Oberflächengestaltung

Eine alternative Lösung zu den beiden oben beschriebenen ist die Mischung der funktionsorientierten mit der objektorientierten Oberfläche. Diese Lösungsmöglichkeit soll im folgenden kurz beschrieben werden. Sie verbindet die Vorteile der beiden anderen Ansätze.

- Eine Vorauswahl der dargestellten Objekte ist möglich.
- Es gibt eine übersichtliche Darstellung.
- Es ist nur ein Fenster notwendig.
- Die Wahl ob der Benutzer lieber objekt- oder funktionsorientiert Arbeiten will, bleibt ihm überlassen.

Eine solche „gemischte“ Oberfläche hätte Ähnlichkeit mit der objektorientierten Oberfläche (siehe Abbildung 5.2 im Abschnitt 5.4.2). Im Unterschied wären bei der

„gemischten“ Oberfläche alle Buttons sofort anwählbar. Der Benutzer hätte nun zwei Möglichkeiten:

1. Objektorientiert: Der Benutzer wählt aus der Objektliste ein Objekt aus. Die Buttons, deren Funktion auf das gewählte Objekt angewendet werden kann, werden hervorgehoben. Der Benutzer kann nun einen dieser Buttons anklicken um die Funktion auszuführen.

2. Funktionsorientiert: Der Benutzer drückt einen Button. Dann wird die Liste der angezeigten Objekte auf diejenigen reduziert, auf die die gewählte Funktion angewendet werden kann. Von diesen Objekten kann der Benutzer nun dasjenige auswählen, das verwendet werden soll.

5.5 Zusammenfassung

Dies war eine kleine Auswahl zum Teil noch offen gebliebener Probleme, die mir wichtig genug erschienen, etwas ausführlicher dargestellt zu werden. Ich wollte versuchen, für diese Auswahl ein paar Lösungsvorschläge zu erarbeiten, die möglicherweise realisiert werden können und die Sicherheit der Anwendung sowie den Komfort für den Anwender erhöhen. Auf mögliche Erweiterungen der graphischen Oberfläche gehe ich im nächsten Kapitel ein.

Kapitel 6

Erweiterung der Oberfläche, Ausblicke

In diesem Kapitel möchte ich einige mögliche Erweiterungen der graphischen Oberfläche, des APRICOTS User Interface, mit möglichen Lösungswegen vorstellen. Diese Erweiterungen würden zum einen den Anwender unterstützen, ihm die Arbeit noch weiter erleichtern, zum andern die Sicherheit der Anwendung erhöhen.

6.1 Help-Funktion

Eine sehr wichtige und äußerst nützliche Funktion für den Anwender ist eine Help-Funktion. Durch die Möglichkeit, Hilfstexte abrufen zu können müßte der Anwender nicht bei jedem kleinen Problem das Handbuch zu Rate ziehen. Wie eine mögliche Implementierung einer Hilfsfunktion aussehen könnte, hängt unter anderem von den Möglichkeiten der Ingres / Windows 4GL-Programmierumgebung ab.

Unter Berücksichtigung dieser Möglichkeiten habe ich mir mehrere realisierbare Implementierungsmöglichkeiten überlegt:

- Ein jeweils größerer Bereich der Fenster wird für die Hilfstextdarstellung reserviert. Dort wird ein Hilfstext ständig für alle Funktionen des jeweiligen Fensters gleichzeitig dargestellt. Diese Lösung gefällt mir weniger gut, da die gesamte Darstellung dadurch unübersichtlich werden würde. Außerdem würden die Frames unnötig groß werden.

- Klickt man mit der rechten Maustaste auf ein Feld (z. B. Button, Anzeigefeld, o. ä.) erscheint ein kleines Fenster, in dem ein Hilfstext für dieses Feld ausgegeben wird. Diese Lösung hat den Nachteil, daß der Benutzer auf jedenfall erst eine Aktion ausführen muß, ehe er den Hilfstext zu Gesicht bekommt.

- Nun die Lösung, die mir am sinnvollsten erscheint. Ein kleiner Bereich von jedem Frame (z. B. eine Zeile unmittelbar unterhalb des Trennbalkens) wird für einen standigen Hilfstext verwendet. In diesem Bereich wird immer eine kurze Erklärung für das Feld des Frames angezeigt, das sich gerade unter dem Mauszeiger befindet. Eine solche kurze Erläuterung würde dem Benutzer in den meisten Fällen schon weiterhelfen, eine ausführlichere Erklärung könnte dann in einem extra erscheinenden Fenster dargestellt werden, wenn der Benutzer die rechte Maustaste über einem Feld drückt.

6.2 Sicherheitsaspekte

6.2.1 Berechtigte Benutzer

Eine mögliche Erweiterung der Oberfläche ist eine Paßwortabfrage. Ein Feld zum Eingeben des Benutzernamen ist im Hauptmenü bereits vorhanden (siehe Abbildung 4.2). Man müßte es nur so erweitern, daß nach Eingabe des Benutzernamen noch ein Paßwort zu diesem verlangt wird. Mit den Mitteln von Ingres / Windows 4GL wäre diese Erweiterung problemlos zu realisieren.

Eine Paßwortabfrage hätte den Vorteil, daß zum einen nur berechtigte Benutzer das APRICOTS User Interface benutzen könnten, zum anderen könnte jeder Benutzer eindeutig identifiziert werden. Die Einführung mehrerer »Berechtigungsklassen« wäre auch ein sinnvoller Aspekt. So könnte man einem unerfahrenen Benutzer verbieten, Änderungen oder Löschungen an den vorhandenen Definitionen vorzunehmen und ihm zum Beispiel nur erlauben, fertige und ausgetestete Komponenten zu verwenden.

Mit der eindeutigen Identifizierung eines Benutzers könnte eine weitere sehr nützliche Funktion in die Oberfläche eingebaut werden: Speichern eines aktuellen Zustandes, wenn der Benutzer mit der Arbeit aufhören will oder das Terminal wechseln muß. So könnte das User Interface von einem Benutzer im gleichen Zustand wieder gestartet werden, in dem er es verlassen hatte.

6.2.2 Sicherung für Ausfall

Es wäre wünschenswert, eine Sicherung aller bisherigen Aktivitäten zu haben, wenn ein Recovery notwendig werden würde. Ein Grund dafür wäre zum Beispiel Stromausfall, Rechnerausfall, etc. Die graphische Oberfläche müßte dahingehend erweitert werden, daß nicht nur alle eingegebenen Daten gesichert werden, sondern auch alle Aktionen wie z. B. die Auswahl von Objekten oder das Anklicken von Buttons. Natürlich müßte die graphische Oberfläche dann auch in der Lage sein, diese gesicherten Daten bei einem Recovery zu verwenden.

6.3 Vermeidung von Deadlocks

Bei einer geschachtelten Client-Server-Umgebung, wie sie im APRICOT-System vorliegt, ist es leicht möglich, daß ein Deadlock, wie in den folgenden Beispielen gezeigt, auftritt.

Beispiel 1:
Komponente A enthält die Services S1 und S2, Komponente B die Services S3 und S4. S1 benötigt Sub-Service S3, S3 benötigt Sub-Service S2. Ein Deadlock tritt nun auf, wenn (Sub-)Service S3 (Sub-)Service S2 aufruft, da die Komponente A noch durch den Service S1 blockiert ist (Abbildung 6.1).

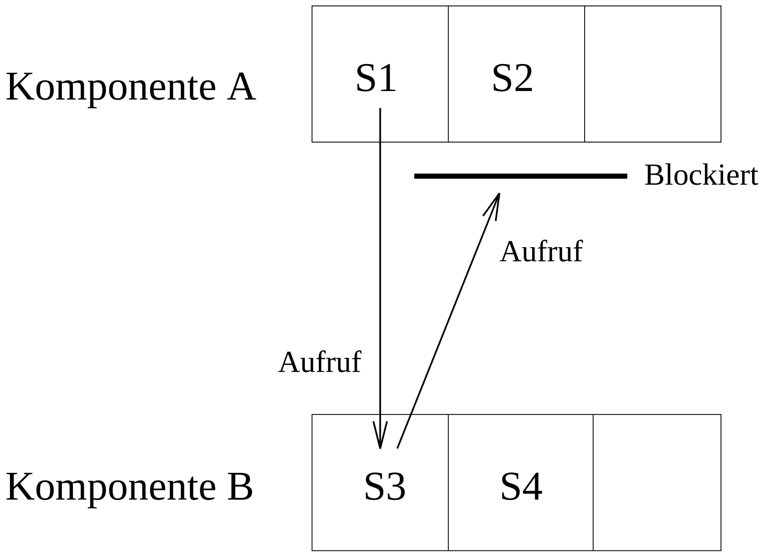

Abbildung 6.1: Beispiel 1: Auftreten eines Deadlocks

Beispiel 2:
Service S1 benötigt den Sub-Service S2, S2 benötigt S3, ..., Sx benötigt S1. Läßt man Service S1 ablaufen, tritt ein Deadlock auf.

Natürlich sollten solche Fälle nicht auftreten. Der Fall im Beispiel 2 beruht sicher auf einen Fehler des Service-Programmieres, der Fall im Beispiel 1 kann jedoch ohne weiteres einmal auftreten.

Man müßte nun Strategien entwickeln, um das Auftreten solcher Deadlocks vermeiden zu können. Für den Fall in Beispiel 2 ist das nicht sehr schwer. Man muß nur mittels dem ConTract Processing Repositories rekursiv alle von S1 benötigten Sub-Services bestimmen. Wenn unter diesen S1 selbst zu finden wäre, hätte man einen Fehler entdeckt, der mit Sicherheit zu einem Deadlock führen könnte.

Für Beispiel 1 ist eine geeignete Strategie nicht so leicht zu finden. Schließlich kann man nicht einfach prüfen, ob ein Sub-Service-Aufruf wiederum die aufrufende Komponente aufruft. Es wäre ja auch möglich, daß es eine zweite Komponente gibt, die den Service enthält. Würde diese aufgerufen werden, dann würde es keinen Deadlock geben.

6.4 Anzeige der dynamischen Serverkonfiguration

(Siehe dazu auch Abschnitt 4.3.2)

Unter diesen Punkt fällt die Erweiterung der graphischen Oberfläche in Hinblick auf die Anzeige der Server-Konfiguration. Momentan kann nur eine einzige, fest programmierte Server-Konfiguration angezeigt werden. Wünschenswert wäre eine Anzeige einer beliebigen Server-Konfiguration, die zur Laufzeit dynamisch erzeugt werden müßte.

6.5 Precompiler-Verwendung bei Services

Bei der Eingabe von neuen Services wird vorausgesetzt, daß der Programmierer das C-File des Service zur Verfügung stellt (siehe Abschnitt 4.3.1). Dadurch wird die Verwendung des Buttons *Editieren des Source Codes* stark eingeschränkt, da ein editieren des (fertigen) C-Codes nur in Ausnahmefällen sinnvoll ist. Deshalb wäre es gut, wenn es vom APRICOTS User Interface aus möglich wäre, ein File durch einen Precompiler übersetzen zu lassen. Dann könnte der Programmierer von Anfang an die Unterstützung der graphischen Oberfläche in Anspruch nehmen und müßte den Code nicht umständlich außerhalb der Oberfläche von einem Precompiler übersetzen lassen.

Probleme bei der Erstellung einer solchen Erweiterung sehe ich vor allem darin, daß verschiedene Precompiler unterstützt werden sollten. Jedes (Datenbank)-System verwendet seinen eigenen Precompiler mit oft auch verschiedenen Aufrufformaten.

6.6 Steuerung der ConTract-Ausführung

Diesen Aspekt führe ich nur der Vollständigkeit halber auf. Die Implementierung dieser Funktion wäre zu aufwendig geworden, um sie im Rahmen dieser Studienarbeit auszuführen. Deshalb ist sie aus dieser Arbeit ausgegliedert worden und wird nun als eigenständige Diplomarbeit realisiert werden [Eitler92].

Eine kurze Übersicht über diese Funktionen:

- Steuerung von ConTracts (starten, stoppen)

- Verwaltung von ConTracts
- Visualisierung von ConTracts (Darstellung eines ConTracts während seiner Abarbeitung)

Anhang A

User Interface: Funktionsübersicht

A.1 Hauptmenü

APRICOTS User Interface: Aufruf der Untermenüs für die statische Konfigurationsverwaltung, die dynamische Konfigurations- und Prozeß-Verwaltung sowie die ConTract-Verwaltung (*noch nicht implementiert*).

A.2 Funktionen für die statische Konfigurationsverwaltung

Static Configuration Management: Untermenü, Aufruf aller Funktionen zum Ansehen, Eintragen, Ändern und Löschen von Services, Resource Managers und Komponenten sowie Aufruf des Frames zum Definieren / Ändern der statischen Konfiguration.

Look Up Components: Zum Betrachten aller bereits definierten Komponenten.

Define a new Component: Anlegen einer neuen Komponenten-Definition.

Redefine existing Component Type: Ändern einer Komponenten-Definition.

Delete Component Type: Löschen einer Komponenten-Definition.

Look Up Resource Manager: Betrachten aller bereits definierten Resource Manager / Step Computation Server.

Define a new Resource Manager Type: Anlegen einer neuen Resource Manager / Step Computation Server Definition.

Redefine Resource Manager Type: Ändern einer Resource Manager / Step Computation Server Definition.

Delete Resource Manager Type: Löschen einer Resource Manager / Step Computation Server Definition.

Look Up Service: Betrachten aller bereits definierten Services.

Define new Service: Anlegen einer neuen Service Definition.

Redefine Service: Ändern einer Service Definition.

Select Required Sub-Services: Eintragen von Sub-Services, die von einem Service benötigt werden.

Delete Service: Löschen einer Service Definition.

Show Configuration: Anzeige einer Konfiguration (Definition). Funktion zum Löschen einer Konfiguration. Aufruf der Funktion zum Installieren einer Konfiguration.

Install New Configuration: Anlegen einer neuen Definition einer Konfiguration.

A.3 Funktionen für die dynamische Konfigurations- und Prozeß-Verwaltung

Dynamic Configuration and Process Management: Untermenü, Funktionsaufrufe zum Starten, Stoppen und Migrieren einer APRICOTS-Komponente und zum Betrachten der dynamischen Server-Konfiguration.

Start Component Instance: Starten einer APRICOTS-Komponente.

Running in Component: Anzeige aller Resource Manager / Step Computation Server, die in einer Komponente enthalten sind.

Resource Manager Interface Description: Anzeige des Service (in einer späteren Version: der Services), der (die) in einem Resource Manager / Step Computation Server enthalten ist (sind).

Required Sub-Service: Anzeige der Sub-Services, die von einem Service benötigt werden.

Stop Component: Stoppen einer laufenden Komponente.

Migrate Component Instance: Migrieren einer Komponente.

Show dynamic Server Configuration: Anzeige einer Server Konfiguration, die momentan abgearbeitet wird. In der jetzigen Version der Oberfläche ist es nur möglich, eine fest programmierte Konfiguration anzuzeigen.

Anhang B

User Interface: Zusammenhang der Funktionen

APRICOTS User Interface — Zusammenhang der Funktionen		
Funktionsname	wird aufgerufen von	ruft auf
APRICOTS User Interface	—	ConTract Management Dynamic Config. Man Static Config. Man.
ConTract Management	APRICOTS User Int.	*noch nicht impl.*
Define a new Component Type	Static Config. Man.	Define Res. Man. Redef. Res. Man. Res. Man. Int. Desc.
Define a new Resource Manager Type	Define new Comp. Redef. ex. Comp. Static Config. Man.	Define new Service Redefine Service
Define New Service	Define Res. Manag. Redef. Res. Manag. Static Configurat.	Sel. Req. Sub-Serv.
Delete Component Type	Static Configurat.	Running in Comp.
Delete Resource Manager Type	Static Configurat.	Res. Man. Int. Desc.
Delete Service	Static Configurat.	
Dynamic Configuration Management	APRICOTS User Int.	Migrate Component Show dyn. Ser. Conf. Start Component Stop Component
Install New Configuration	Show Configuration	—
Look Up Components	Static Configurat.	Running in Comp.
Look Up Resource Manager	Static Configurat.	Res. Man. Int. Desc.
Look Up Services	Static Configurat.	Required Sub-Serv.
Migrate Component Instance	Dynamic Configurat.	Running in Comp.

Funktionsname	wird aufgerufen von	ruft auf
Redefine existing Component Type	Select Cmp. to Red.	Define Res. Man. Redef. Res. Man. Res. Man. Int. Desc.
Redefine Resource Manager Type	Define new Comp. Redef. ex. Comp. Select RM to Redef	Define new Service Redefine Service
Redefine Service	Define Res. Man. Redef. Res. Man. Sel. Serv. to Red.	Sel. Req. Sub-Serv.
Required Sub-Services	Resource Man. Int.	—
Resource Manager Interface Description	Define new Comp. Delete Res. Man. Redef. ex. Comp. Running in Comp. Select RM Type Red	Required Sub-Serv.
Running in Component	Delete Component Migrate Cmp. Inst. Sel. Comp. to Red. Show Configuration Start Component Stop Component	Res. Man. Int. Desc.
Select Component Type to Redefine	Static Configurat.	Redef. ex. Comp. Running in Comp.
Select Required Sub-Services	Define New Service Redefine Service	—
Select Resource Manager Type to Red.	Static Configurat.	Redef. Res. Man. Res. Man. Int. Desc.
Select Service to Redefine	Static Configurat.	Redefine Service
Set Hostname	Show Configuration Static Configurat.	Show Configuration
Show Configuration	Set Hostname	Install New Config. Running in Comp. Set Hostname
Show Dynamic Server Configuration	Dynamic Configurat.	—
Start Component Instance	Dynamic Configurat.	Running in Comp.

Funktionsname	wird aufgerufen von	ruft auf
Static Configuration Management	APRICOTS User Int.	Define Component Define Res. Man. Define Service Delete Component Delete Res. Man. Delete Service Select Comp. to Red. Select RM to Redef. Select Serv. to Red. Set Hostname
Stop Component	Dynamic Configurat.	Running in Comp.

Literaturverzeichnis

[Bilger92] Anette Bilger
Modellierung und Simulation betrieblicher Abläufe als ConTracts mit dem APRICOT-System
Studienarbeit, IPVR, Universität Stuttgart, 1992

[BurRai90] Holger Burkert, Bernd Raichle
Entwicklung eines Daten- und Verarbeitungsmodells für den Kontext
Studienarbeit, IPVR, Universität Stuttgart, 1990

[CaSeCa86] L. F. Cabrera, S. Sechrest, R. Caceres
The Administration of Distributed Computations in a Networked Environment – An Interim Report
in: Proc. 6. Int. Conference on Distributed Computing Systems, IEEE, Cambridge, Massachusetts, 1986

[Eitler92] Marcus Eitler
Ereignisgesteuerter Ablaufmonitor für ConTracts im APRICOTS
Diplomarbeit, IPVR, Universität Stuttgart, 1992 (in Vorbereitung)

[Ing90/1] Ingres / Windows 4GL
Ingres Application Editor User's Guide for Ingres / Windows 4GL
Ingres Corporation, 1990

[Ing90/2] Ingres / Windows 4GL
Language Reference Manual for Ingres / Windows 4GL
Ingres Corporation, 1990

[Laube92] Dieter Laube
Transaktionsorientierte Kommunikationsmechanismen für das verteilte Transaktionssystem APRICOTS
Diplomarbeit, IPVR, Universität Stuttgart, 1992

[Laube92/1] Dieter Laube
make_apricot_component:
Werkzeug zur Generierung von APRICOTS-Komponenten
Arbeitspapier, IPVR, Universität Stuttgart, 1992

[LoSch87] P. C. Lockemann, J. W. Schmidt (Hrsg.)
Datenbank-Handbuch
Springer-Verlag, 1987

[Mansf90] Niall Mansfield
Das Benutzerhandbuch zum X Window System
Addison-Wesley, 1990

[Masman91] Ioannis Masmanidis
ConTract-Programmierung im APRICOT-System
Arbeitspapier, IPVR, Universität Stuttgart, 1991

[Rowe92] Lawrence A. Rowe
A Retrospective on Database Application Development Frameworks
in: Sigmod Record, Vol. 21, No. 1, March 1992

[Schill92] Alexander Schill
Strukturelle Verwaltung verteilter Programme: Ein Überblick über Konzepte und Systeme
in: Wirtschaftsinformatik, Heft 1, Februar 1992

[Schnei86] Matthias Schneider
Methoden und Werkzeuge zur Software-Entwicklung in einer objektorientierten Programmierumgebung
Universität Stuttgart, 1986, Kapitel 5

[Sommer87] Ian Sommerville
Software Engineering
Addison-Wesley Verlag, 1987

[WäReu91] Helmut Wächter, Andreas Reuter
The ConTract Model
in: A. K. Elmagarmid (Ed.):
Transaction Models for Advanced Database Applications
Morgan Kaufmann Publishers, 1991

[Wahl91] Peter Wahl
Interpretative Abwicklung von Prädikat-Transaktionsnetzen
Studienarbeit, IPVR, Universität Stuttgart, 1991

[XOpen91] X/Open CAE Specification
Distributed Transaction Processing: The XA Specification
X/Open Company Ltd., December 1991

[Zellmer92] Oliver Zellmer
APRICOTS User Interface — Manual (Arbeitstitel)
Arbeitspapier, IPVR, Universität Stuttgart, 1992 (in Vorbereitung)