9. Berechnungen in Amica

Eine Applikationsarchitektur in Amica besteht aus einer Menge von Rechenkomponenten, die Operationen beschreiben, die auf Datenobjekten oder mit Datenströmen arbeiten. Abbildung 9.1 zeigt den abstrakten Basistyp `DataOperation`, von dem alle konkreten Typen für Rechenkomponenten abgeleitet sind. Er ist eine Spezialisierung des Komponententyps `OperationCp` aus dem Erweiterungsmodul für ablaufbasierte Applikationen, das in Kapitel 6 beschrieben ist.

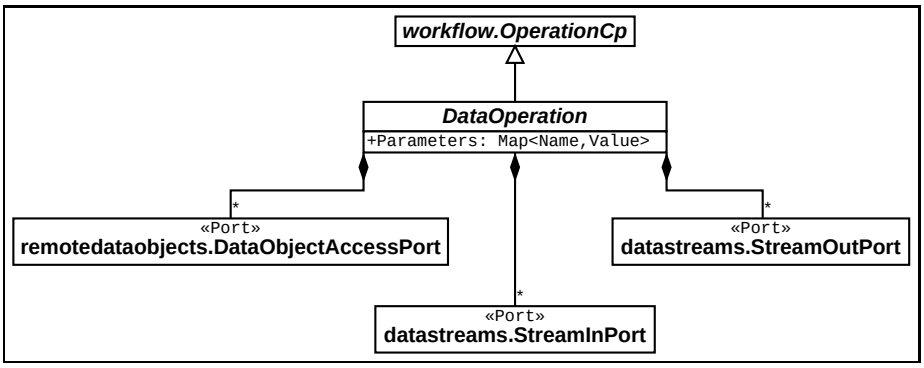

Abbildung 9.1.: Architekturelemente für Operationen auf Datenobjekten und -strömen

`DataOperation`-Komponenten greifen über Anschlüsse vom Typ `DataObjectAccessPort` auf Datenobjekte zu. Genauer gesagt, lassen sich Anschlüsse von diesem Typ über Konnektoren des Typs `DataAccess` mit den Anschlüssen `read` oder `write` von `DataObject`-Komponenten verbinden. Eingehende Datenströme werden über `StreamInPort`-Anschlüsse angebunden, während für ausgehende Datenströme `StreamOutPort`-Anschlüsse benutzt werden. Operationen sind parametrisierbar, wobei die Parameter in dem Attribut `Parameters` durch eine partielle Abbildung von Namen auf Werte gegeben ist. Da die eingesetzte Architekturbeschreibungssprache derart komplexe Attributtypen nicht unterstützt, wird in der Implementierung eine Sequenz von Zeichenketten benutzt.

Das Verhalten von `OperationCp`-Komponenten, das in Abbildung 6.2 auf Seite 73 beschrieben ist, besteht hauptsächlich daraus, dass für jedes eintreffende Aktivierungssignal die Operation einmal ausgeführt wird. Die hier eingesetzten `DataObject`-Komponenten sind aber auch mit Datenströmen, also mit über Kanälen fließende Datenfolgen, verbunden und es vereinfacht die Erstellung von Applikationsarchitekturen, wenn Operationen auch dadurch

ausgelöst werden, wenn Daten über einen Kanal an der Rechenkomponente ankommen. Anderenfalls müsste der Applikationsentwickler dafür sorgen, dass jede Rechenkomponente zuerst ein Aktivierungssignal bekommt, damit eine Operation mit Daten aus einem Kanal arbeiten kann.

Eine Operation kann sich aber auch über mehrere Daten, die aus einem Kanal ankommen, erstrecken. So könnte, die Operation z.B. eine Filterung oder eine Kompression darstellen. Das Verhalten der allgemeinen `OperationCp`-Komponente ist damit nicht ausreichend. Abbildung 9.2 beschreibt ein entsprechend erweitertes Verhalten von `DataOperation`-Komponenten. Ein `sin`-Ereignis beschreibt dabei das Eintreffen eines Datums über einen Kanal. Ist die Rechenkomponente bei dem Eintreffen eines solchen Datums inaktiv, also im Zustand `idle`, wird sie aktiviert und die eigentliche Operation ausgeführt. Ist sie beim Eintreffen aktiv, ändert sich ihr Zustand nicht. Die Operation, die gerade ausgeführt wird, behandelt dann das eingetroffene Datum. Ansonsten entspricht das Verhalten dem der `OperationCp`-Komponente.

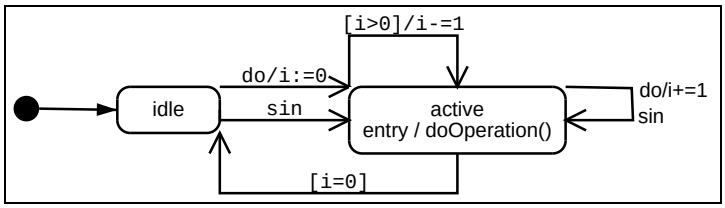

Abbildung 9.2.: Verhalten von Rechenkomponenten (`DataOperation`)

Für die Übersetzung und Ausführung von `DataOperation`-Komponenten stellt das entsprechende Erweiterungsmodul `DataOperations` nur eine abstrakte Basisfunktionalität und Hilfsfunktionen bereit. Es gibt zwei konkrete Operationstypen, welche diese Funktionalität einsetzen. Sie werden in den Kapiteln 9.2 und 9.3 beschrieben.

9.1. Zusammengesetzte Rechenkomponenten

Mit einer zusammengesetzten Rechenkomponente lassen sich Subapplikationsarchitekturen, die eine Berechnung auf Datenobjekte oder auf Datenströmen durchführen, in einer Komponente zusammenfassen. Abbildung 9.3 zeigt den Komponententyp mit der grafischen Darstellung der Komponente selber und ihrem grafischen Elementeditor.

Die Subapplikation wird als Attribut `subsystem` spezifiziert. Der Wert des Attributs ist eine beliebige Applikationsarchitektur, die das gesamte Vokabular der sie umfassenden Applikation benutzen kann. Insbesondere kann sie wiederum zusammengesetzte Rechenkomponenten enthalten. Die Beziehung zwischen der zusammengesetzten Rechenkomponente und ihrer internen Subapplikation wird über Anschlüsse spezifiziert. Genauer gesagt, werden Anschlüsse der Rechenkomponente mit Anschlüssen von Komponenten der internen Subapplikation identifiziert. In dem Editor ist dies durch Kästchen in der Subapplikation dargestellt, die mit dem Namen des äuße-

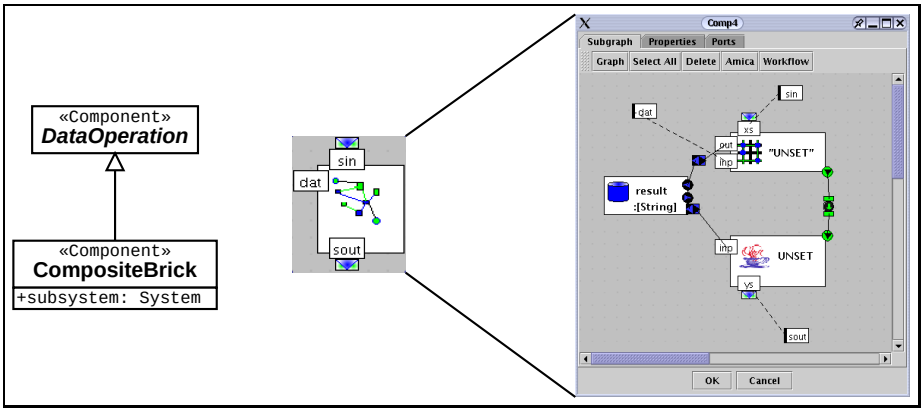

Abbildung 9.3.: Zusammengesetzte Rechenkomponente

ren Anschlusses beschriftet sind und die mit dem assoziierten Anschluss einer Komponente der inneren Subapplikation verbunden sind.

Bei der Übersetzung einer zusammengesetzten Komponente wird als erster Schritt die Komponente durch ihre interne Subapplikation ersetzt. Dabei werden die Namen der internen Architekturelemente um den Namen der zusammengesetzten Komponente als Präfix erweitert. Zusätzlich werden die Namen der Datenobjekte angepasst, die als Attribut der Komponente spezifiziert sind. Um die weiteren Probleme bei dem Ersetzen der Komponente zu lösen, wird auf den in Kapitel 5.2.2 beschrieben Ansatz zurückgegriffen.

9.2. Integration entfernter Rechendienste

Wie in Kapitel 7 beschrieben, wurde Amica für rechenintensive Applikationen entwickelt, die sich grobkörnig in Subapplikationen mit loser Kopplung zerlegen lassen. Diese Subapplikationen, die i.a. selber einen hohen Rechenaufwand erfordern, werden unter Einsatz entfernter Rechenressourcen ausgeführt. Die Aufgabe von Amica besteht hauptsächlich aus der Koordination der Subapplikationen. Sowohl die Subapplikationen als auch die Art der entfernten Rechenressourcen können von unterschiedlichster Natur sein:

- Im einfachsten Fall besteht eine Subapplikation aus Code, der auf einem entfernten Hochleistungsrechner ablauffähig ist. In diesem Fall muss Amica den Code auf dem Rechner installieren, die Eingabedaten zu dem Code transportieren, ihn entfernt starten und schließlich die Ergebnisse abholen.

- Alternativ kann die Subapplikation auch aus einer optimierten Implementierung für eine bestimmte Zielarchitektur bestehen, die in einer Bibliothek enthalten ist. So stehen beispielsweise für die Bibliothek LAPACK (Linear Algebra PACKage) eine Vielzahl von Implementierungen zur Verfügung, die für unterschiedliche Rechnertypen hoch optimiert

sind [LAP02]. Das Amica-System muss für die Ausführung der Applikation auf dem Zielrechner ein kleines Programm installieren, welches die Eingabedaten lädt, die Bibliotheksfunktion auf diesen ausführt und die Ausgabedaten entsprechend speichert. Dieses Programm muss mit der Bibliothek gebunden und dann ausgeführt werden. Das System IceT basiert auf diesem Ansatz [GS99].

- Eine Variation dieses Ansatzes besteht darin, dass keine Bibliothek von ausführbarem Code eingesetzt wird, sondern eine Bibliothek von Programmen in Quellcode, die für bestimmte Klassen von Rechnerarchitekturen optimiert sind. So bietet sich für Parallelrechner mit verteiltem Speicher eher der Einsatz von Prozessparallelität an, während für Systeme mit gemeinsamen Speicher sich häufig eher Datenparallelität eignet. Wenn die konkrete Zielarchitektur mit ihrer Anzahl der verfügbaren Prozessoren etc. und die Eingabedaten mit ihren Kardinalitäten bekannt sind, dann kann der Quellcode zu optimierten Implementierungen übersetzt werden. Eine prominente Programmiersprache, die einen derartigen Ansatz unterstützt ist High-Performance-Fortran [CMZ95].

- Ein unterschiedlicher Ansatz besteht darin, dass die Subapplikation über einen Dienstanbieter über ein applikationsspezifisches Protokoll ausgeführt wird. Ein Beispiel hierfür ist das System Mathematica der Firma Wolfram-Research, das einen derartigen entfernt aufrufbaren Dienstanbieter unterstützt [Wol02]. Dieser kann dann beispielsweise eine symbolische oder auch numerische Lösung von Differentialgleichungssystemen berechnen. Die Aufgabe von Amica bestünde darin, unter Einsatz des spezifischen Protokolls die gewünschte Aufgabe mit ihren Eingabedaten berechnen zu lassen und die erzeugten Ausgabedaten abzuholen.

Um diese unterschiedlichen Ansätze für Berechnungen einheitlich in Amica zu integrieren, wurde eine generische Modellierung der entfernten Rechenressourcen für den Applikationsentwickler gewählt. Jeder Einsatz eines entfernten Rechners ist ein Rechendienst. Der Applikationsentwickler wählt diesen Dienst über einen eindeutigen Namen aus. Er kann ihn parametrisieren und mit den Ein- und Ausgabedaten verbinden. Bei Aktivierung sorgt eine Infrastruktur für eine korrekte Ausführung des Rechendienstes.

9.2.1. Vokabular

Abbildung 9.4 zeigt den Komponententyp `RemoteService` mit seiner grafischen Repräsentation im Editor. Dieser entspricht einer Operation, die einen entfernten Rechendienst ausführt. Der Applikationsentwickler spezifiziert über das Attribut `Service` den gewünschten Rechendienst. Zusätzlich verbindet er die `RemoteService`-Komponente mit den gewünschten Datenobjekten und Kanälen. In der Repräsentation werden die oberen Anschlüsse (`sin`) für die eingehenden Datenströme und die unteren (`sout`) für die ausgehenden Datenströme benutzt. Die Anschlüsse auf der linken Seite (`dout`) werden mit den Datenobjekten verbunden. Der Anschluss rechts oben wird

für eingehende Aktivierungssignale benutzt, während der Anschluss rechts unten für die ausgehenden verwandt wird.

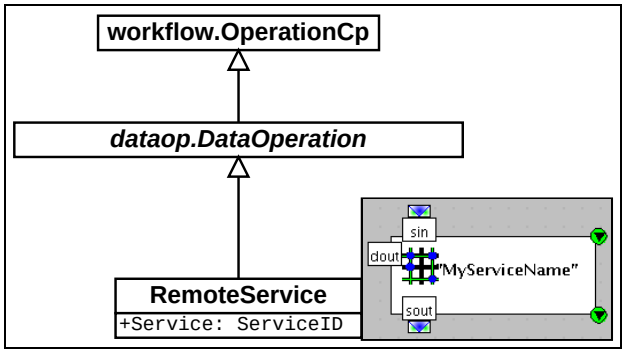

Abbildung 9.4.: Architekturelemente für die Nutzung entfernter Rechendienste in Amica

Mit dem vorgestellten Vokabular lässt sich nun das Beispiel aus dem Kapitel 7.1 entwickeln. Abbildung 9.5 zeigt die dazugehörige Applikationsarchitektur. Da der Schwerpunkt hier auf die Verdeutlichung der Nutzung entfernter Ressourcen liegt, wurde in der Applikationsarchitektur auf die Steuerung der Simulation verzichtet.

Zuerst werden die drei Sensoren gestartet. Dabei wird in der Applikation davon ausgegangen, dass es einen speziellen Rechendienst gibt, der einen Sensor aktiviert. Welcher Sensor aktiviert wird, wird über Parameter spezifiziert. Die aktivierten Sensoren liefern Messdaten als Datenströme. Geht man davon aus, dass die atomaren Daten der Datenströme auch eine Information bzgl. der Datenquelle enthalten, kann man alle 3 Datenströme mit der Strategie „Direkte Vermischung“ zusammenfassen. Der Ergebnisstrom wird einer Rechenkomponente übergeben, welche die Daten filtert und dem Datenobjekt `InputData` speichert. Wenn diese Rechenkomponente genügend Daten gesammelt hat, schickt sie ein Unterbrechungssignal an die Sensoren, die daraufhin terminieren.

Nachdem die Filterung beendet ist, wird die Rechenkomponente `Simulation` aktiviert. Diese führt die Simulation auf den Eingabedaten aus und erzeugt Ergebnisdaten. Ist sie beendet, werden die Ergebnisse visualisiert. Danach terminiert die Applikation

Diese Beispielapplikation verdeutlicht auch einige Nachteile der `RemoteService`-Komponenten. Es ist nicht davon auszugehen, dass immer derartig spezialisierte Dienste zur Verfügung stehen. Im allgemeinen werden allgemeine Dienste angeboten werden, die der Applikationsentwickler evtl. um eigenen Code erweitern muss. Ein Beispiel für einen derartig allgemeinen Rechendienst wird in Kapitel 10.2 vorgestellt. Er erlaubt die entfernte Ausführung nebenläufiger Programme, die der Applikationsentwickler zur Verfügung stellt.

Desweiteren ist es nicht garantiert, dass die Rechendienste miteinander kompatibel sind. Es ist evtl. notwendig, die Ergebnisse eines Rechendienstes anzupassen, bevor sie von einem anderen Rechendienst als Eingabe benutzt

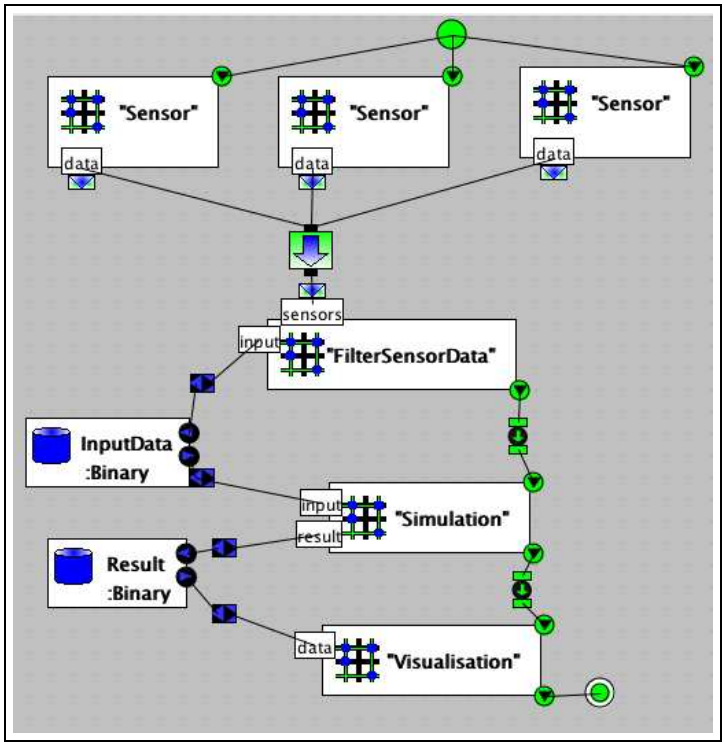

Abbildung 9.5.: Applikationsarchitektur eines Beispiels einer typischen rechenintensiven Applikation

werden können. Schließlich können Rechendienste nicht auf lokale Resourcen, wie z.B. Dateien, zugreifen und auch die Interaktion mit dem Benutzer ist nicht einfach zu realisieren. Für diese Problemfälle gibt es einen weiteren Rechenkomponententyp, der in Kapitel 9.3 vorgestellt wird.

Die Übersetzung einer `RemoteService`-Komponente besteht aus der Erzeugung eines Applikationsobjekts, dessen Klasse eine Spezialisierung der Klasse `OperationNode` aus dem Erweiterungsmodul für arbeitsablaufbasierte Applikationen ist. Erhält dieses Applikationsobjekt zur Laufzeit ein Aktivierungssignal, kontaktiert es die Infrastruktur und lässt diese den spezifizierten Rechendienst ausführen.

9.2.2. Infrastruktur

Das Hauptaufgabe der Infrastruktur besteht darin, für eine Dienstspezifikation des Applikationsentwicklers zur Laufzeit einen möglichst guten Rechendienst zu erzeugen und diesen auszuführen. Hierfür müssen zunächst für eine Dienstspezifikation alle geeigneten Rechendienste bestimmt werden. Von diesen wird dann die beste Alternative gewählt werden, welche i.a. diejenige sein wird, die den schnellsten Rechendienst anbietet[1]. Da unterschied-

[1]Die schnellste Alternative muss nicht zwangsläufig die beste sein. Besteht eine Applikation z.B. aus drei Berechnungen ($P_1 \mid\mid P_2$); P_3, wobei P_1 und P_2 nebenläufig berechnet werden können, P_3 aber auf P_1 und P_2 warten muss. Sei P_1 nun deutlich rechenaufwendiger

lichste Rechendienste zu unterstützen sind, wurde das generische Fabrik-Entwurfsmuster eingesetzt. Abbildung 9.6 zeigt schematisch die hierfür eingesetzten Objekte.

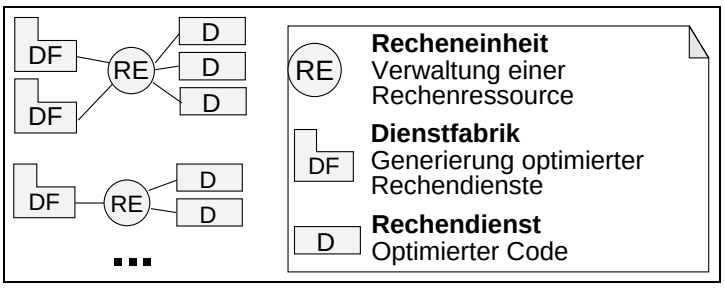

Abbildung 9.6.: Schematische Darstellung der Infrastruktur für die Nutzung entfernter Rechendienste

Rechendienste werden von *Dienstfabriken* erzeugt. Jede Dienstfabrik ist genau einer *Recheneinheit* zugeordnet, welche die konkreten Rechenressourcen verwalten. Eine konkrete Umsetzung besteht z.B. aus einem massiv parallelen Feldrechner, für den ein numerisches Programm in Form von Quellcode vorliegt. Die Dienstfabrik übersetzt dieses Programm in optimierten Code, wobei sie ihr Wissen über die Eingabedaten und über den speziellen Feldrechner in die Optimierung mit einbezieht. Dieses optimierte Programm stellt dann den Rechendienst dar, der von der Recheneinheit auf den Rechner geladen und ausgeführt wird.

Eine alternative Umsetzung ist die Verwendung einer optimierten Bibliothek für einen Parallelrechner mit verteiltem Speicher. Die Dienstfabrik erzeugt hier ein kleines Programm, das nur die gewünschte Bibliotheksfunktion aufruft, übersetzt es und bindet es mit der Bibliothek. Das gebundene Programm ist dann der Rechendienst, der von der Recheneinheit auf den verfügbaren Prozessoren ausgeführt wird.

Wird nun ein Applikationsobjekt für eine `RemoteService`-Komponente aktiviert, so muss es zuerst eine geeignete Dienstfabrik finden. Abbildung 9.7 zeigt die hierfür notwendigen Schritte. Beim Aufsetzen der Infrastruktur melden sich alle Dienstfabriken mit dem von ihnen unterstützten Dienst bei dem Vermittlungsdienst von Amica an, der direkt auf dem CORBA-Trading-Service[OMG97] der OMG aufsetzt. Um nun für einen bestimmten Typ von Rechendiensten die vorhandenen Dienstfabriken zu lokalisieren, wendet sich ein `RemoteService`-Applikationsobjekt mit dem Namen des benötigten Rechendienstes an den Vermittlungsdienst. Es erhält darauf eine Liste geeigneter Fabriken.

Um nun die bestmögliche Dienstfabrik bestimmen zu können, sendet das Applikationsobjekt eine Beschreibung des Problems an die Dienstfabriken,

als P_2, dann ist es nicht nötig, für P_2 die leistungsfähigsten Rechenressourcen zu verbrauchen. Noch komplexer wird die Zuordnung von Rechenressourcen, wenn man zusätzlich annimmt, dass deren Verbrauch mit echten Kosten verbunden ist, die von dem Benutzer getragen werden. Ein Projekt, das sich mit dieser Problematik befasst, ist *Economy Grid* [Eco02].

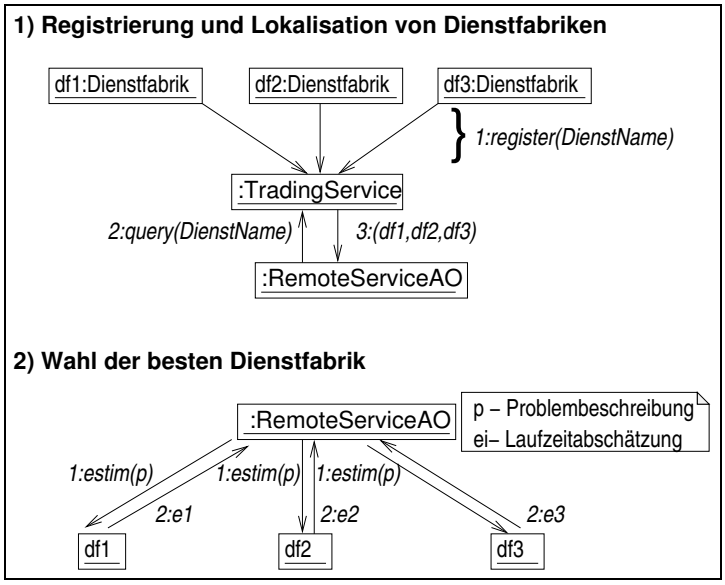

Abbildung 9.7.: Registrierung und Auswahl von Fabrikobjekten

welches von dem Rechendienst zu bearbeiten ist. Diese Problembeschreibung besteht aus Referenzen auf die Datenobjekte, welche die Eingabedaten enthalten und den von dem Applikationsentwickler bestimmten Parametern. Evtl. Datenstromkanäle werden nicht übergeben, da eine Leseoperation ihren Inhalt verändert. Anhand dieser Problembeschreibung senden die Dienstfabriken eine Laufzeitabschätzung für einen von ihnen für dieses Problem generierbaren Rechendienst zurück.

Diese Laufzeitabschätzung kann auf einem Modell des Algorithmus basieren. In [FGR97] wurde beispielsweise das Laufzeitverhalten eines Optimierungsverfahrens für Neuronale Netze mathematisch modelliert, wobei neben den Problemparametern auch Kennzahlen des eingesetzten Rechnersystems, wie Anzahl und Güte der Prozessoren und Geschwindigkeit des Kommunikationssystems mit einbezogen wurde. Anhand dieses Modells lässt sich sowohl das Laufzeitverhalten als auch eine optimale Anzahl einzusetzender Prozessoren bestimmen. Eine Dienstfabrik kann dieses Wissen einsetzen, um einen im Sinne des Modells optimalen Rechendienst zu erzeugen.

Neben dem Wissen um das algorithmische Verhalten der erzeugbaren Rechendienste geht auch die aktuelle Auslastung und Leistung der Recheneinheit in diese Abschätzung mit ein, die mit der Dienstfabrik assoziiert ist. Dies ermöglicht zusätzlich eine automatische Lastverteilung in der Infrastruktur, da die aktuell belasteten Rechenressourcen vermieden werden.

Nachdem eine Dienstfabrik gewählt wurde, muss der Rechendienst erzeugt und ausgeführt werden. Abbildung 9.8 zeigt beispielhaft die einzelnen Schritte. Das `RemoteService`-Applikationsobjekt kontaktiert die ausgewählte Dienstfabrik und lässt sie für das zu behandelnde Problem einen Rechendienst `R1` erzeugen. Dieser registriert sich bei der mit der Dienstfabrik assoziierten Recheneinheit. In diesem Beispiel benutzt ein Rechendienst alle von der

Recheneinheit verwalteten Rechenressourcen, so dass immer nur ein Rechendienst zur Zeit berechnet werden kann. Zum Zeitpunkt der Registrierung von `R1` sei gerade ein anderer Rechendienst `R2` aktiv. `R1` muss daher warten, bis `R2` seine Berechnungen abgeschlossen hat. Ist `R2` fertig, benachrichtigt er die Recheneinheit, die daraufhin `R1` aktiviert.

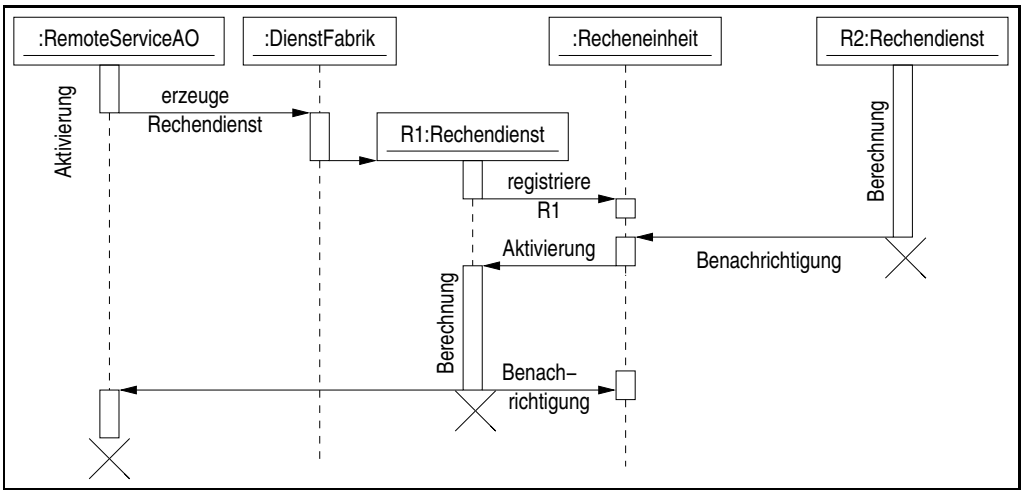

Abbildung 9.8.: Ausführung eines entfernten Rechendienstes

Hat `R1` seine Berechnungen beendet, benachrichtigt er die Recheneinheit und das Applikationsobjekt, das seine Erzeugung veranlasst hat. Diese Benachrichtigung besteht nur aus einem Signal und enthält keine Ergebnisdaten. Die Ergebnisse der Berechnungen sind in Datenobjekten oder in Datenströmen abgelegt und stehen so weiteren Rechendiensten zur Verfügung.

Zusammengefasst ermöglicht dieser Ansatz die optimierte Nutzung heterogener Ressourcen, indem spezielle Implementierungen erzeugt werden, welche sowohl die Charakteristika der eingesetzten Hardware als auch des aktuellen Problems berücksichtigen. Zusätzlich bietet er eine automatische Lastverteilung. Er beinhaltet aber auch einige inhärente Probleme, die im folgenden erläutert werden.

9.2.2.1. Probleme der Laufzeitabschätzung

Um eine gute Leistung der Applikation zu erzielen, ist die Wahl der richtigen Dienstfabrik von grundlegender Bedeutung. Als Instrument zur Entscheidungsfindung dient dabei hauptsächlich die Laufzeitabschätzung, welche von der Dienstfabrik anhand der Problembeschreibung geliefert wird. Bei der Bestimmung der Abschätzung einer konkreten Implementierung sind die folgenden Kriterien zu beachten:

Berechenbarkeit
In der Klasse der μ-rekursiven Funktionen ist es schon aufgrund des Halteproblems im allgemeinen nicht möglich, die Terminierung und die Laufzeit

eines Algorithmus abzuschätzen. In diesem Fall gibt es zwei Lösungsansätze zur Bewertung einer Implementierung:

1. Es wird ein Maß eingesetzt, welches gar nicht erst versucht, die Laufzeit zu bestimmen, sondern nur die Leistung der eingesetzten konkreten Rechenressourcen und die Güte des Algorithmus für die gewählten Parameter und Eingabedaten berechnet. So wird beispielsweise für die Bewertung eines Dienstes zur Übersetzung und Ausführung paralleler Programme nur die Anzahl der zur Verfügung stehenden Prozessoren, ihre Leistung und ihre aktuelle Auslastung eingesetzt.

2. Es wird versucht, die Laufzeit durch mathematische Modellierung oder anhand von Erfahrungswerten abzuschätzen. Der erste Ansatz ist dann möglich, wenn die genaue Implementierung und die Charakteristika der eingesetzten Hardware bekannt ist. Der zweite Ansatz ist möglich, wenn sich die Eingabedaten in Klassen mit vergleichbaren Laufzeiten einordnen lassen.

Nutzt man spezifische Rechendienste, wie numerische Algorithmen, ist die Abschätzung der Rechenzeit häufig möglich. Es gibt auch Vorschläge, sich auf die Klasse primitiv-rekursiver Funktionen zu beschränken [EFGR97, EG95], die sich gut analysieren läßt. Obwohl dieser Ansatz eine starke Einschränkung darstellt, eignet er sich doch für numerische Algorithmen. Die Klasse der primitiv-rekursiven Funktionen entspricht der Klasse der LOOP-N Programme, also der Programme, die sich mit Schleifen mit fester Anzahl von Schleifendurchläufen bilden lassen. Viele numerische Verfahren lassen sich mit derartigen Schleifen formulieren und fallen damit in die Klasse der primitiv-rekursiven Funktionen darstellen.

Für primitiv-rekursive Funktionen lassen sich immer Laufzeitabschätzungen angeben, die eine obere Grenze der Laufzeit darstellen und somit eine Vergleichbarkeit zwischen unterschiedlichen Dienstfabriken erlauben. Somit ist dieser Ansatz für numerische Algorithmen, die im Quellcode vorliegen, gut geeignet. Wenn der Rechendienst dagegen sehr generisch ist, und z.B. auch die Integration von Code erlaubt, der von dem Benutzer zur Verfügung gestellt wird, dann ist dieser Ansatz nicht durchführbar. In einem solchen Fall muss der Anwender eine verlässliche Laufzeitabschätzung liefern. Ist dieses nicht möglich, kann die zu berechnende Funktion nicht als Kriterium für die Laufzeitabschätzung benutzt werden.

Aktuelle Last und Existenz von Ressourcen

Die Laufzeit einer Implementierung hängt stark von den zur Verfügung stehenden Rechenressourcen ab. Um diese abschätzen zu können, fragt eine Dienstfabrik seine assoziierte Recheneinheit nach ihrem aktuellen Status. Das dabei eingesetzte Protokoll ist nicht durch Amica vorgegeben, sondern wird für jeden konkreten Rechenresourcentyp neu konzipiert. Der Grund für dies Entscheidung liegt in der hohen Heterogenität. Die Rechenressource kann einerseits ein Spezialrechner, wie z.B. der Feldrechner MasPar, sein, auf dem Applikationen im Batch-Betrieb laufen, oder ein heterogener Cluster aus Workstations, die gleichzeitig von anderen Anwendern mit einer höheren Priorität

interaktiv genutzt werden können. Dies erschwert es, ein allgemeines Maß für die Last zu definieren.

Wahl der Implementierung
Für den gleichen Rechendienst können einem Fabrikobjekt unterschiedliche Implementierungen zur Verfügung stehen, die auf unterschiedlichen Algorithmen beruhen. In Abhängigkeit der Eingabedaten und der Parameter können so die geeigneten gewählt werden. So gibt es z.B. für die Verarbeitung dünn besetzter Matrizen andere Verfahren als für dicht besetzte. Das hierfür benötigte Wissen muss dem Fabrikobjekt manuell hinzugefügt werden.

9.2.2.2. Aktivierung durch Datenströme

Das Verhalten von Rechenkomponenten ist, wie in Kapitel 9 beschrieben, derart definiert, dass auch das Eintreffen von Stromdaten über einen Kanal eine Aktivierung und damit eine Berechnungsoperation auslöst. Konkret bedeutet dies für die Nutzung entfernter Rechendienste, dass sich ein `RemoteService`-Applikationsobjekt bei den Strömungskanälen der Infrastruktur registrieren muss, welche die eingehenden Datenströme der ursprünglichen `RemoteService`-Komponente verwaltet. Erhält das Applikationsobjekt nun Daten von diesen Kanälen, erzeugt es, wie oben beschrieben, einen Rechendienst.

Hier sind zwei Probleme zu betrachten. Erstens muss der Rechendienst auch das Datum erhalten, welches das Applikationsobjekt erhalten hat, um seine Berechnungen korrekt durchzuführen. Dieses wurde aber schon vom Kanal versendet und von dem Applikationsobjekt verbraucht. Meldet sich der Rechendienst danach bei dem Kanal an, wird er das Datum nicht noch einmal erhalten. Zweitens ist sicherzustellen, dass der Kanal nicht während der Erzeugung des Rechendienstes weitere Daten versendet, die dann ebenfalls verloren gehen.

Um diese Probleme zu umgehen, registriert sich der Rechendienst nicht bei dem Strömungskanal sondern bei seinem Applikationsobjekt. Dieses dient somit als Stellvertreter des Kanals und sorgt dafür, dass keine Daten verloren gehen. Dieser Stellvertreteransatz enthält eine zusätzliche Indirektionsstufe, die einen erhöhten Kommunikationsaufwand bedingt. Unter der Annahme, dass die Granularität der Applikation grob ist, und damit der Rechenaufwand im Vergleich zu dem Kommunikationsaufwand gross ist, fällt der zusätzliche Kommunikationsaufwand bzgl. der Gesamtleistung der Applikation kaum ins Gewicht.

9.3. Lokale Adapter- und Interaktionskomponenten

Auch wenn sich der beschriebene `RemoteService`-Komponententyp gut für die Nutzung unterschiedlichster entfernter Rechenressourcen eignet, so ergeben sich in der praktischen Anwendung innerhalb einer Applikation doch ein paar Probleme. So ist i.A. nicht davon auszugehen, dass Rechendienste stets

miteinander kompatibel sind. Ein Rechendienst für die Berechnung der Inversen einer Matrix kann z.B. ein anderes Format für die Repräsentation von Matrizen einsetzen als ein Rechendienst für die Bestimmung einer Determinanten. Möchte man nun beide Rechendienste nacheinander ausführen, wobei der zweite Rechendienst als Eingabe das Ergebnis des ersten Rechendienstes benutzt, dann muss die Repräsentation der Matrix zwischen der Ausführung der Rechendienste angepasst werden. Da es nicht realistisch ist, davon auszugehen, dass diese Anpassung ebenfalls immer von Rechendiensten vorgenommen werden kann, wird ein Komponententyp benötigt, der es dem Applikationsentwickler ermöglicht, eigenen Code ausführen zu lassen. Solche Komponenten dienen dann als *Adapter* zwischen den Rechendiensten.

Ein zweiter Nachteil des `RemoteService`-Komponententyps besteht darin, dass es schwierig ist, mit dem Benutzer zu interagieren. Es ist zwar durchaus denkbar, dass ein entfernter Rechendienst unter Einsatz des X-Windows-System ein interaktives Fenster auf dem Rechner des Benutzers öffnet; dieser Ansatz ist aber sowohl schwierig in der Handhabung, da der Benutzer sein System entsprechend konfigurieren muss, als auch mit einem meist spürbaren Kommunikations- und Rechenaufwand verbunden. Einfacher wäre es, eine Komponente einzusetzen, die es erlaubt, interaktiven Code direkt auf der Maschine des Benutzers auszuführen.

Ein dritter Nachteil besteht darin, dass entfernte Rechendienste nur mit einigem Aufwand auf Ressourcen zugreifen können, die lokal auf der Maschine des Benutzers vorliegen. So werden i.A. die Eingabedaten eines Nutzers in seinem lokalen System als Datei vorliegen und sich nicht in einem Datenobjekt des Amica-Systems befinden. Einer der ersten Schritte einer Applikation besteht meist darin, die Daten von dem lokalen System entweder in ein entferntes Datenobjekt oder in einen Strömungskanal zu speisen. Hierfür eignen sich entfernte Rechendienste nicht.

Die Lösungsansätze für alle drei Problemfelder beruhen stets auf der lokalen Ausführung von Code, der von dem Applikationsentwickler bereit gestellt wird. Für die Lösung dieser Problematik präsentiert Abbildung 9.9 den Komponententyp `JavaOperation`, der es ermöglicht, Java-Code lokal als Rechenoperation ausführen zu lassen.

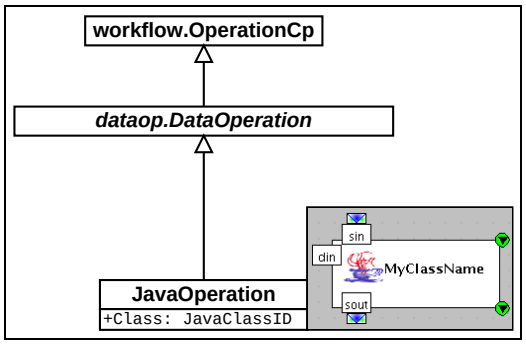

Abbildung 9.9.: Java-basierte Operationen auf Datenobjekte und Datenströme

Das Attribut `Class` spezifiziert die Klasse, von der zur Laufzeit ein Objekt

erzeugt wird, welche die Operation durchführt. Diese Klasse muss die Schnittstelle `JavaOperation` unterstützen, die in Abbildung 9.10 dargestellt ist. Sie enthält zwei Methoden. Die Methode `init()` initialisiert ein Objekt mit den folgenden Parametern:

- das Applikationsobjekt `ao`, welches durch die Übersetzung der `JavaOperation`-Komponente entstanden ist,
- die in dem Attribut `parameters` des Komponententyps `DataOperation` spezifizierten Parameter als Abbildung von Namen auf Werten, die beide durch Zeichenketten gegeben sind,
- eine Abbildung `dataobjects` von den Namen der `DataAccess`-Anschlüsse auf die Namen der mit den Anschlüssen verbundenen Datenobjekte,
- eine Abbildung `incomingStreams` von den Namen der `StreamInPort`-Anschlüsse auf die Namen der mit diesen verbundenen Kanal-Konnektoren und
- eine entsprechende Abbildung `outgoingStreams` für die ausgehenden Datenströme.

Die zweite Methode `doOperation()` führt die von dem Objekt bereitgestellte Operation bei jeder Aktivierung aus.

```
interface JavaOperation
{
   void init(JavaOperationAO ao,
             Map<String,String> parameters,
           Map<String,String> dataobjects,
           Map<String,String> incomingStreams,
           Map<String,String> outgoingStreams);

   void doOperation();
}
```

Abbildung 9.10.: Schnittstelle für in Adapterkomponenten eingebetteten Java-Code

Komponenten vom Typ `JavaOperation` werden zu `JavaOperationAO`-Applikationsobjekten übersetzt. Diese sind eine Unterklasse der Applikationsobjekte für `DataOperation`-Komponenten. Sie erweitern sie um ein Objekt *op* der in dem Attribut `Class` spezifizierten Klasse, das mit den entsprechenden Werten initialisiert ist. Zur Laufzeit ruft das Applikationsobjekt bei jeder Aktivierung die Methode *doOperation()* auf *op* auf.

Um den Applikationsentwickler bei der Programmierung des Javacodes zu unterstützen, ist es möglich, über das Applikationsobjekt, welches das Objekt

verwaltet, auf eine Laufzeitunterstützung zuzugreifen, die entsprechende unterstützende Funktionalität enthält. Sie ermöglicht es, über den Namen eines Datenobjekts eine CORBA-Referenz auf das Objekt zu erhalten. Der Javacode kann dieses dann wie ein lokales Java-Objekt benutzen. Ist das Datenobjekt noch nicht existent und ist es kein externes Datenobjekt, dann wird es von der Laufzeitunterstützung automatisch erzeugt. Um den Kommunikationsaufwand zu reduzieren, speichert die Laufzeitunterstützung Referenzen auf Datenobjekte und braucht sie so bei einer erneuten Anfrage nicht wieder über die Infrastruktur zu ermitteln. Die notwendigen Informationen über die Datenobjekte sind der Laufzeitunterstützung durch die Applikationsobjekte bekannt, die für die Datenobjekte bei der Übersetzung erzeugt werden.

Auch die CORBA-Referenzen der Kanäle der Datenströme können über den Namen der entsprechenden Kanalkonnektoren von der Laufzeitunterstützung bestimmt werden. Da die Nutzung dieser Kanäle allerdings etwas komplexer ist, wird dem Applikationsentwickler hier weitere Unterstützung angeboten. Statt auf die Kanäle direkt über CORBA zuzugreifen, ist es möglich, lokale Hilfsobjekte zu benutzen, die komfortablere Nutzungsmöglichkeiten erlauben. Abbildung 9.11 zeigt die zugehörigen Klassen für einen Datenstrom aus Zeichenketten.

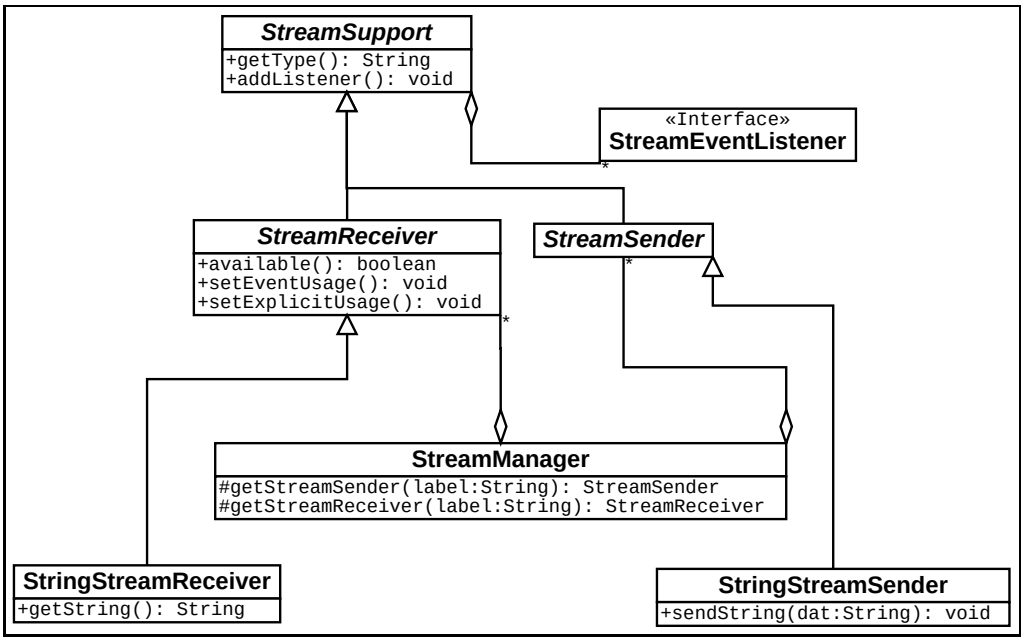

Abbildung 9.11.: Lokale Unterstützung beim Zugriff auf Datenströme

Der Zugriff auf diese Hilfsobjekte geschieht über ein `StreamManager`-Objekt, das über das `JavaOperation`-Applikationsobjekt verfügbar ist. Über den Namen des Anschlusses *port* liefert die Methode `getStreamSender()` ein Hilfsobjekt, um Daten an den mit *port* verbundenen Kanal zu schicken. Die Methode *getStreamReceiver()* liefert analog ein Hilfsobjekt, um Daten zu empfangen.

Um Daten zu senden, gibt es für jeden Elementtyp *T* eines Datenstroms eine spezialisierte *T*`StreamSender`-Klasse, die eine Methode `send`*T*`()` enthält. In dem Klassendiagramm in Abbildung 9.11 sind exemplarisch nur die Klassen für Datenströme aus Zeichenketten aufgeführt.

Für den Empfang von Daten gibt es zwei Alternativen, die von der abstrakten Klasse `StreamReceiver` und ihren konkreten Unterklassen unterstützt werden. Zum einen ist es möglich, mit der blockierenden Methode `getString()` explizit auf Daten zu warten. Zum anderen wird ein Ereignismodell unterstützt, mit der Interessenten beim Empfang von Daten oder bei der Terminierung des Datenstroms benachrichtigt werden. Welche dieser beiden Alternativen benutzt wird, wird in dem entsprechenden `StreamReceiver`-Objekt mittels `setEventUsage()` und `setExplicitUsage()` gewählt. Eine gleichzeitige kombinierte Nutzung wird als nicht sinnvoll erachtet, da die gewünschte Semantik[2] unklar ist. Wenn von dem Modus des blockierenden Wartens auf den Ereignismodus gewechselt wird, werden alle evtl. gepufferten Daten sofort als Ereignis verschickt.

Da es auch bei ausgehenden Datenströmen möglich ist, dass der Datenstrom aufgrund von Empfängern zusammenbricht, unterstützen auch `StreamSender`-Hilfsobjekte den Ereignismechanismus. Interessenten, die sich bei ihnen anmelden, erhalten eine Benachrichtigung, wenn der verbundene Kanal eine fehlerhafte Unterbrechung signalisiert.

9.4. Weitere Elementtypen

Mit den bisher vorgestellten Elementtypen lassen sich schon einfache verteilte nebenläufige Applikationen erstellen. Diese sind aber zwei Beschränkungen unterworfen. Erstens gibt es keine Elementtypen, die es ermöglichen, den Kontrollfluss in Abhängigkeit des Systemzustands zu steuern. Es ist also nicht möglich, in Abhängigkeit des Ergebnis von Berechnungen oder aufgrund von Interaktion mit dem Benutzer den Kontrollfluss zu manipulieren.

Zweitens beschränkt die statische Struktur der Applikationsarchitektur den Grad der Nebenläufigkeit von Applikationen. Die maximale Anzahl der nebenläufigen Kontrollflüsse in einem Algorithmus hängt häufig von den Eingabedaten ab und kann damit erst zur Laufzeit bestimmt werden. Soll z.B. ein Film berechnet werden, ist es evtl. möglich, jedes Bild unabhängig von den anderen Bildern zu berechnen. Die maximale Nebenläufigkeit entspricht damit der Anzahl der Bilder, die aber erst anhand der Filmbeschreibung zur Laufzeit bekannt ist. Mit den bisher vorgestellten Elementtypen wird aber nur eine statische nebenläufige Implementierung unterstützt.

Die folgenden zwei Kapitel beschreiben Elementtypen, die Lösungen für beide Problembereiche enthalten.

[2] Eine Möglichkeit besteht darin, dass alle Daten sowohl als Ereignis als auch bei dem blockierenden Warten angeboten und damit dupliziert werden. Alternativ ist es auch denkbar, dass blockierendes Warten immer Vorrang vor der Ereignisversendung genießt, was aber schwierig zu realisieren ist.

9.4.1. Bedingte Verzweigungen

Zur strukturierten Programmierung existieren in einschlägigen Programmiersprachen eine Vielzahl von Konstrukten, um den Kontrollfluss in Abhängigkeit des Systemzustands zu steuern. Hierzu zählen die Schleifenkonstrukte und die bedingten Verzweigungen. Da sich die bedingten Verzweigungen für die Fallstudien als ausreichend erwiesen hatten, wurde auf eine Realisierung von Schleifenkonstrukten verzichtet.

Abbildung 9.12 enthält den Konnektortyp `DataObjectSwitch` mit seiner grafischen Repräsentation, der es erlaubt, den Kontrollfluss in Abhängigkeit des Wertes eines Datenobjekts zu steuern. Er ist eine Spezialisierung des abstrakten Konnektortyps `Switch` aus dem Erweiterungsmodul für arbeitsablaufbasierte Applikationen, das in Kapitel 6 beschrieben wurde. Die Rolle `dataobject` wird mit dem `read`-Anschluss eines Datenobjekts verbunden. Aktuell muss dieses Datenobjekt eine Zeichenkette speichern, also das Attribute `type` auf `String` gesetzt sein.

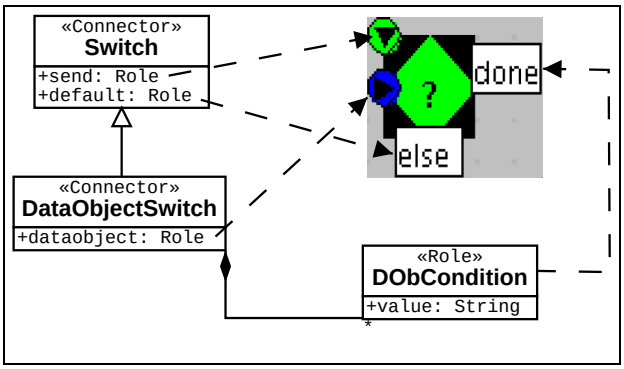

Abbildung 9.12.: Bedingte Verzweigung aufgrund des Inhalts eines Datenobjekts

Ein `DataObjectSwitch`-Konnektor kann beliebig viele Rollen vom Typ `DObCondition` besitzen. Jede dieser Rollen ist über das Attribut `value` parametrisierbar. Sie können mit `do`-Anschlüssen von Komponenten des Typs `OperationCp` aus dem Erweiterungsmodul für arbeitsablaufbasierte Applikationen verbunden werden.

Erhält der Konnektor ein Aktivierungssignal, ermittelt er zunächst den aktuellen Wert v des mit ihm verbundenen Datenobjekts. Dann sucht er in seinen `DObCondition`-Rollen nach einer Rolle R, dessen `value`-Attribut den Wert v besitzt. Findet er eine solche, dann schickt er ein Aktivierungssignal an die mit R verbundene Operationskomponente. Findet er keine, schickt er ein Aktivierungssignal an die mit der `default`-Rolle verbundene Operationskomponente. Wenn keine Komponente mit der ausgewählten Rolle verbunden ist, geht das Aktivierungssignal verloren.

Bei der Übersetzung wird aus dem `DataObjectSwitch`-Konnektor ein Applikationsobjekt erzeugt, das genau das oben beschriebene Verhalten implementiert.

9.4.2. Dynamische Nebenläufigkeit nach dem Farm-Muster

Die Architekturbeschreibungssprache Acme, die als Grundlage für die Beschreibung von Applikationsarchitekturen eingesetzt wird, unterstützt nur statische Applikationsarchitekturen. Es ist also nicht möglich, die Applikationsarchitektur zur Laufzeit zu verändern. Wie oben erörtert, ist es allerdings wünschenswert, einen dynamischen Grad an Nebenläufigkeit zu unterstützen, der von den Eingabedaten abhängt. Ein Lösungsweg für diesen Gegensatz von statischer Architektur und dynamischer Nebenläufigkeit stellen Nebenläufigkeitsmuster dar [DGTY95, Bra94].

Diese basieren darauf, dass die Kontrollstrukturen für nebenläufige Algorithmen häufig eine hohe Regularität aufweisen. Eines der am häufigsten eingesetzten Muster ist das Farm-Muster. Abbildung 9.13 beschreibt das Muster links als Funktion `farm` in Pseudocode. Als Eingabe dient eine Menge `xs` von Eingabedaten und eine Funktion `f`. Das Ergebnis der Funktion ist eine neue Menge `ys`, die dadurch entsteht, dass auf jedes Element aus *xs* die Funktion *f* angewendet wird.

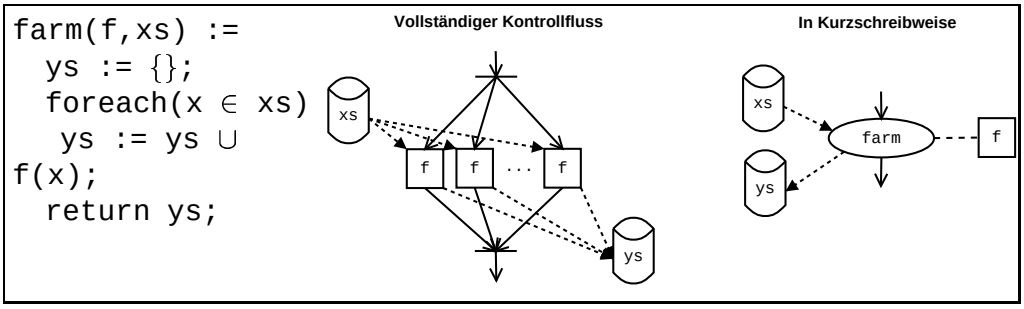

Abbildung 9.13.: Architekturoperator für das Farm-Entwurfsmuster

Da die Auswertungen der Funktion `f` voneinander unabhängig sind, können sie nebenläufig ausgeführt werden. In der Abbildung ist die entsprechende Kontrollstruktur in der Mitte dargestellt. Die Anzahl der Funktionsauswertungen ist abhängig von der Größe der Menge `xs`. Da die entstehende Struktur sehr regulär ist, ist es aber sofort klar, wie sie zu erstellen ist, wenn die Anzahl der Elemente bekannt ist. Daher kann die Kontrollstruktur auch mit einer Kurzdarstellung, wie z.B. die rechte Grafik der Abbildung, vollständig dargestellt werden. Das Element `farm` wird dann zur Laufzeit durch die mittlere Struktur ersetzt.

In Amica unterstützt der Komponententyp `Farm` dieses Nebenläufigkeitsmuster. Die nebenläufig auszuwertende Funktion wird dabei durch eine interne Applikationsarchitektur beschrieben. Eine `Farm`-Komponente ist eine Operation, die bei Aktivierung die folgenden Schritte ausführt:

1. Zuerst werden soviele Kopien der internen Applikationsarchitektur erzeugt, wie Elemente in dem Eingabedatenobjekt vorhanden sind.

2. In jeder Kopie wird lokal ein Datenobjekt für das zugeordnete Eingabedatum und eines für das Ausgabedatum erzeugt. Zusätzlich werden

die Komponenten in den Kopien mit anderen Datenobjekten und Datenströmen anhand der Bindungen zwischen den Anschlüssen der `Farm`-Komponente und den Anschlüssen der internen Applikationsarchitektur verbunden.

3. Nun werden alle Kopien gestartet und es wird gewartet, bis alle Kopien terminiert sind.

4. Danach werden die Ergebnisdaten aus den lokalen Datenobjekten der Kopien gesammelt und in das Datenobjekt für die Ergebnisdaten übertragen. Abschließend werden alle Kopien gelöscht und die Operation ist beendet.

Abbildung 9.14 zeigt den Komponententyp `Farm` und ein Beispiel mit dem zugehörigen Elementeditor. Der Elementeditor zeigt die angewählte `Farm`-Komponente aus dem hinteren Fenster. Er besitzt einen Karteireiter `Function`, um die interne Applikationsarchitektur zu bearbeiten.

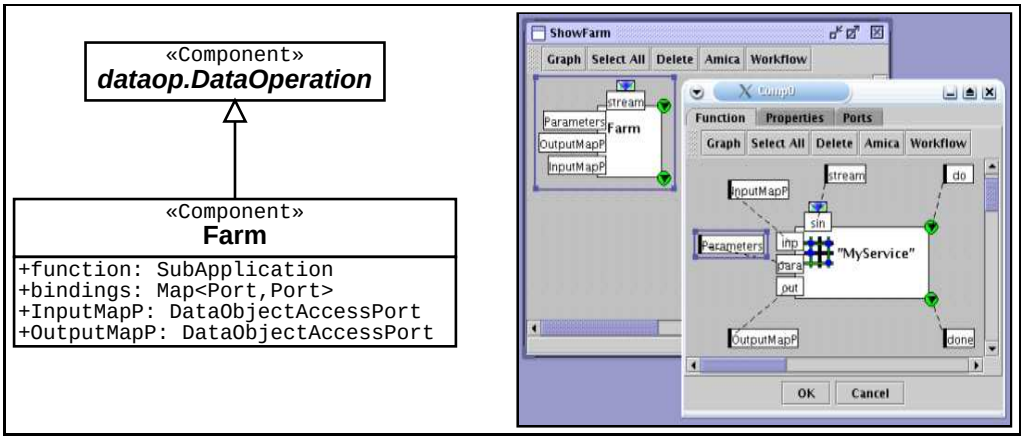

Abbildung 9.14.: Architekturelement für das Master-Worker Paradigma

Über den Anschluss `InputMapP` wird die Farmkomponente mit einem Datenobjekt verbunden, das alle Eingabedaten, also die `xs` aus Abbildung 9.13, enthält. Als Datentyp wurde dabei keine Menge sondern eine Abbildung gewählt. Über den Anschluss `OutputMapP` wird die Komponente mit dem Datenobjekt verbunden, das nach der Ausführung die Ergebnisdaten enthalten soll. Die Anschlüsse der `Farm`-Komponente werden an Anschlüsse von Komponenten der internen Applikationsarchitektur gebunden.

Zur Laufzeit wird die Farm-Komponente konzeptionell durch soviele Kopien der internen Applikationsarchitektur ersetzt, wie es Elemente in dem mit `InputMapP` verbundenen Datenobjekt gibt. Diese Kopien werden nun anhand der Bindungen zwischen den Anschlüssen der `Farm`-Komponente und der internen Applikationsarchitektur analysiert und angepasst. Dabei werden die folgenden Regeln angewendet:

- Die interne Applikationsarchitektur wird gestartet, indem ein Aktivierungssignal an den Anschluss α gesendet wird, der mit dem Anschluss

`do` für eingehende Aktivierungssignale der `Farm`-Komponente gebunden ist. Um nun alle Kopien F_i der internen Applikationsarchitektur gleichzeitig zu aktivieren, wird ein Aktivierungssignal für jedes F_i an den zugehörigen Anschluss α_i gesendet.
In dem in der Abbildung dargestellten Beispiel ist der Anschluss `do` der `Farm`-Komponente an den `do`-Anschluss der `RemoteService`-Komponente gebunden, welche die einzige Komponente der internen Applikationsarchitektur ist.

- Die interne Applikationsarchitektur gilt als terminiert, wenn sie ein Aktivierungssignal über den Anschluss ω sendet, der mit dem Anschluss `done` der `Farm`-Komponente gebunden ist. Die Operation, der `Farm`-Komponente ist dann beendet, wenn jeder Anschluss ω_i jeder Kopie F_i ein Aktivierungssignal gesendet hat.
  In dem Beispiel ist der `done`-Anschluss der `Farm`-Komponente mit dem `done`-Anschluss der `RemoteService`-Komponente verbunden. Die Funktion der internen Applikationsarchitektur besteht damit nur aus der Ausführung eines entfernten Rechendienstes.

- Da jede Kopie F_i ihr eigenes Eingabedatum erhält, wird für jede Kopie ein neues Datenobjekt I_i erzeugt. Dessen Wert entspricht dem F_i zugeordneten Eingabewert. Der `read`-Anschluss von I_i wird mit dem Anschluss von F_i verbunden, der mit dem Anschluss `InputMapP` gebunden ist.
  In dem in der Abbildung dargestellten Beispiel ist das der Anschluss `inp`.

- Der gleiche Ansatz wird bei den Ergebnisdaten eingesetzt. Für jede Kopie wird ein eigenes Datenobjekt erzeugt, welches das Ergebnis der Berechnung speichern soll. Dessen `write`-Anschluss wird mit dem Anschluss der internen Applikationsarchitektur verbunden, der mit `OutputMapP` gebunden ist.
  In dem Beispiel ist dies der Anschluss `out` der `RemoteService`-Komponente.

- Alle sonstigen Anschlüsse der Kopien, die mit Anschlüssen der `Farm`-Komponente für den Zugriff auf Datenobjekte und Datenströme gebunden sind, werden direkt mit den Rollen verbunden, mit denen auch die gebundenen Anschlüsse der `Farm`-Komponente verbunden sind.
  In dem abgebildeten Beispiel sind die beiden Anschlüsse `Parameters` und `stream` an die Anschlüsse `para` und `sin` der internen `RemoteService`-Komponente gebunden.

- Es sind keine Bindungen zwischen Anschlüssen der `Farm`-Komponente und der internen Applikationsarchitektur erlaubt, die nicht von den obigen Regeln erfasst werden.

Bei der Übersetzung der Applikationsarchitektur wird aus einer `Farm`-Komponente ein entsprechendes Applikationsobjekt ao_{farm} erzeugt. Die interne Ap-

plikationsarchitektur wird zu einem Geflecht von Applikationsobjekten übersetzt, das von ao_{farm} als Muster gespeichert wird. Wird ao_{farm} zur Laufzeit nun aktiviert, wird das oben beschriebene Verhalten durchgeführt. Die Kopien der internen Applikationsarchitektur werden durch Kopieren des gespeicherten Musters erzeugt.

Neben dem hier realisierten Farm-Muster existieren noch viele andere typische Kontrollstrukturmuster für nebenläufige Berechnungen. Eine häufig eingesetzte Variante des Farm-Musters ist das Master-Worker-Muster. Bei diesem wird eine feste Anzahl von Arbeitern, die nebenläufig ihre Arbeit verrichten können, von einer Kontrollinstanz, dem Master, mit Aufgaben versorgt. Dieses Muster eignet sich deshalb sehr gut für parallele Implementierungen, da eine Zuordnung von je einem Arbeiter pro Prozessor bei feingranularen Aufgaben eine automatische Lastverteilung bewirkt. So setzt bspw. das System MW nur dieses Muster zur Entwicklung nebenläufiger Programme ein [GKLY00].

Ein anderes typische Muster ist das Faltungsmuster [Bra94]. Hier wird eine Menge von Daten reduziert, indem eine Funktion auf je zwei Elemente angewendet wird und als Ergebnis ein neues Element liefert. Wenn die Menge also $2n$ Elemente besitzt, können n Funktionsauswertungen nebenläufig durchgeführt werden. Ein Beispiel ist die Maximumssuche in einer Menge. Die auszuführende Funktion vergleicht zwei Elemente und liefert das größere zurück. Mit dem Faltungsmuster und einer ausreichenden Menge von Prozessoren, lässt sich das Maximum einer Menge mit 2^n Elementen in n Schritten bestimmen.

Dies und viele weitere dieser Muster wurden im Rahmen der Skeleton-Forschung untersucht[Bra94, DGTY95]. Dabei wurden meistens funktionale Sprachen eingesetzt, da Funktionen höherer Ordnung die Spezifikation und Implementierung solcher Muster vereinfachen. Ein Beitrag dieser Arbeit ist die Übertragung solcher funktionalen Entwurfsmuster auf architekturbasierte Systeme. Dieses wurde am Beispiel des Farm-Musters demonstriert. Der hier verwendete Ansatz läßt sich aber auch auf andere Muster übertragen.