Ein Überblick über die Programmiersprache

Emerald

Emerald ist eine objektorientierte Sprache zur Programmierung verteilter Applikationen. Sie wurde in den Jahren 1985-1987 an der Universität Washington entwickelt, beschrieben und implementiert. Diese Ausarbeitung beschreibt die Eigenschaften der Sprache und vergleicht die Implementation von Verteiltheit mit anderen Ansätzen.

Frank Pilhofer
fp@informatik.uni-frankfurt.de
Wintersemester 1994/1995

Dieses Dokument ist auch im WWW abrufbar.
`http://www.uni-frankfurt.de/~fp/AlI/Emerald/`

Inhaltsverzeichnis

1. Einleitung 3

2. Grundlagen 3

2.1. Objektorientierung 3

2.2. Polymorphismus 4

2.3. Mobilität 4

3. Bisherige Ansätze 5

3.1. Xerox RPC 5

3.2. EPL 5

3.3. Smalltalk 6

4. Die Sprache Emerald 6

4.1. Objekte 6

4.2. Sprachkonstrukte 8

4.3. Weitere Möglichkeiten in Emerald 10

5. Implementation 10

5.1. Betriebssystem-Grundlagen 10

5.2. Objekte 11

5.3. Mobilität 12

5.4. Code Sharing 12

5.5. Auffinden von Objekten 13

5.6. Traffic Reduction 13

5.7. Zusammenfassung 13

6. Resumée 14

7. Bemerkung 15

8. Literaturverzeichnis 16

1. Einleitung

Nach der Ablösung von Großrechnersystemen durch vernetzte Workstations sind verteilte Systeme alltäglich geworden. Dabei beschränkt sich die Verteiltheit dieser Systeme meistens auf verteilte Dateisysteme. Jeder Rechner, oder eine kleine Anzahl von Servern, ist für einen kleinen Teil in der Dateisystemhierarchie verantwortlich, und mittels NFS[1] [10] werden alle kleinen Teile zu einem Ganzen zusammengesetzt.

Die Programmierung verteilter Applikationen, die gleichzeitig kooperativ auf mehreren Rechnern des Netzes laufen, ist allerdings noch immer eine Aufgabe für Spezialisten. Verteilte Anwendungen müssen zwangsweise Daten über Rechnergrenzen hinweg austauschen. Bei diesem Austausch muß der Programmierer ständig von Hand auf Kommunikationsprimitiven zugreifen, stets müssen Spezial- und Fehlerfälle abgeprüft werden. Kurzum, für effektive Programmierung verteilter Anwendungen existiert bislang noch unzureichende Unterstützung.

Dabei gibt es genügend Beispiele, in denen verteilte Anwendungen von Vorteil sind. Bei Überlastung eines Rechners können Teile von Anwendungen auf andere Rechner verschoben werden. Rechenintensive Programme können auf mehreren Rechnern parallel laufen und somit ihre Antwortzeit deutlich verkürzen. Sicherheitskritische Dienste können von mehreren Rechnern angeboten werden, damit diese Dienste auch bei Ausfall eines Teils des Netzwerks erreichbar sind.

Es wurden schon mehrere Möglichkeiten erdacht, diese Unterstützung zu liefern. Bestehende Systeme haben jedoch entweder den Nachteil, daß sich der Aufruf einer nicht-lokalen Prozedur von dem Aufruf einer lokalen Prozedur unterscheidet, oder daß der Zugriff auf lokale Prozeduren unnötig ineffizient ist.

Emerald [3] ist ein weiterer Versuch, die verteilte Programmierung zu unterstützen. Das Konzept einer konsequent objektorientierten Umgebung, in der jedes Konstrukt von einer einfachen Zahl bis hin zu einer kompletten Applikation prinzipiell vom gleichen abstrakten Typ "Objekt" ist [1], wird in Emerald auf verteilte Systeme übertragen. Die Semantik der Sprache unterscheidet nicht zwischen lokalen und entfernten Objekten, so daß das Netzwerk für den Programmierer vollkommen transparent ist. Gleichzeitig sind alle Objekte beliebiger Größe beweglich und können jederzeit auf Wunsch des Programmes oder nach Entscheidung eines Betriebssystemes auf andere Rechner verschoben werden [2]. Dabei wird in Emerald versucht, der inhärente Performanceverlust dieses allgemeinen Konzeptes durch Intelligenz in Compiler und Betriebssystem sowie durch enge Zusammenarbeit beider Teile zu vermindern.

2. Grundlagen

Alle drei hier beschriebenen Eigenschaften, Objektorientierung, Polymorphismus und Mobilität werden von Emerald implementiert. An dieser Stelle sollen diese Stichwörter kurz allgemein erläutert werden.

2.1. Objektorientierung

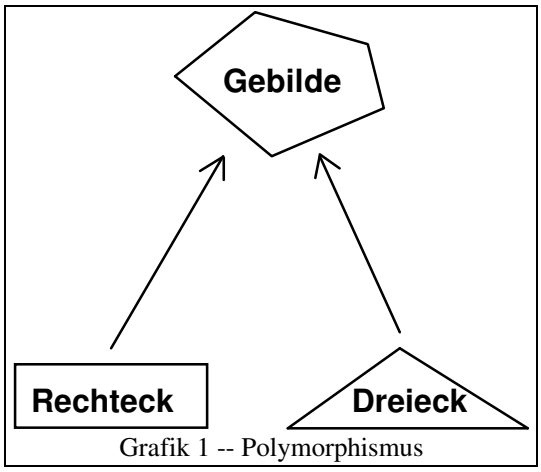
In einem objektorientierten System hat ein Objekt eine genau spezifizierte Schnittstelle zur Außenwelt. Andere Objekte können nur über Funktionen dieser Schnittstelle auf das Objekt zugreifen, so daß die interne Repräsentation nicht sichtbar ist. Diese Abkapselung der inneren von der äußeren Darstellung hat den gewichtigen Vorteil, daß sich die Implementation eines Objektes ändern kann, ohne daß sich das Objekt nach außen hin anders verhält. Dieses Verhalten ist zwar auch in prozeduralen Modellen möglich, doch lädt die externe Sichtbarkeit der Implementation zum Mißbrauch der inneren Strukturen ein.

---

[1] Das Network File System erlaubt, Verzeichnisse eines fremden Rechners in die lokale Verzeichnishierarchie einzuhängen. Dies geschieht (fast) völlig transparent für Benutzer und Applikationen, die keine Kenntnis vom NFS benötigen um es zu benutzen.

2.2. Polymorphismus

Objektorientierte Systeme implementieren meist auch Polymorphismus [3,11]. Polymorphismus heißt "die Fähigkeit, verschiedene Formen anzunehmen" und bedeutet, daß Prozeduren eines Objektes aufgerufen werden können, ohne den genauen Typ des Objektes zu kennen. Eng verbunden mit diesem Konzept ist der Begriff der Konformität. Ein Objekt verhält sich konform zu einem anderen Objekt, wenn es mindestens dessen Funktionalität anbietet. Ein Objekt kann durch ein konformes Objekt ersetzt werden.



Grafik 1 -- Polymorphismus

Das Standardbeispiel für Konformität und Polymorphismus ist eine Bibliothek mit Grafikfunktionen, in der ein Objekt *Gebilde* existiert, welches die Funktionen *drehen* und *bewegen* anbietet. Die Objekte *Rechteck* und *Dreieck* verhalten sich konform zum *Gebilde*, da auch sie diese beiden Funktionen anbieten können. Wir können daher an jeder Stelle, an der wir das *Gebilde* verwenden können, auch ein *Rechteck* oder ein *Dreieck* verwenden. Dabei stellt der Polymorphismus sicher, daß bei jedem Aufruf der Funktion *drehen* auf einem *Gebilde* die richtige Funktion aufgerufen wird, also *Rechteck.Drehen*, falls das Objekt ein Rechteck ist, und *Dreieck.Drehen*, falls es sich um ein Dreieck handelt. Konformität ist trivialerweise reflexiv und transitiv, aber nicht symmetrisch. Zu beachten ist, daß das Gebilde nicht als eigene Implementation vorhanden sein muß; das Gebilde ist ein abstrakter Typ [1], der nur existiert, um die Beschreibungen des Rechteckes und des Dreieckes zu vereinen.

2.3. Mobilität

Mobilität ist eine wichtige Eigenschaft in verteilten Systemen. Üblicherweise wird Mobilität verlangt, um bei Überlastung eines Rechners Aufgaben oder auch Teile davon auf andere Rechner auszulagern. Es gibt jedoch noch andere Gründe, die Mobilität rechtfertigen. Mobilität kann nötig werden, damit Anwendungen auf spezielle Gegebenheiten eines Rechners, wie z.B. Numbercruncher-Hardware oder Grafikprozessoren Rücksicht nehmen können. Bei völliger Transparenz eines Netzwerkes wird diese Eigenschaft unumgänglich. Von grobkörniger Mobilität spricht man, wenn nur ganze Prozesse auf einmal bewegt werden können; in einem feinkörnig mobilen System sind auch Teile von Prozessen beweglich.

Mobilität ist allerdings nicht einfach zu implementieren, da ein Prozeß selten völlig von dem Quellrechner losgelöst werden kann. Hardware wie eine Festplatte, auf der temporäre Dateien lagern oder Ein/Ausgabe-Terminals sind unbeweglich, worauf stets Rücksicht genommen werden muß. Diese Probleme potenzieren sich beim Übergang von grobkörniger auf feinkörnige Mobilität. Bei der Bewegung eines Objektes ändert sich die komplette Umgebung des Objektes. Prozeduren, die vormals lokal waren, sind nun entfernt, während eventuell andere, früher entfernte Prozeduren plötzlich lokal sind. Ebenfalls müssen sich Objekte des Quellrechners, die vormals lokal auf das sich bewegende Objekt zugegriffen haben, auf die neue Situation einstellen.

Bisherige Systeme, die Mobilität implementieren, lösen derartige Probleme, in dem auf alle Objekte mit allgemeinen Mechanismen zugegriffen wird, auch wenn sich das Zielobjekt auf dem gleichen Rechner wie das aufrufende Objekt befindet. Dies resultiert in einer schlechten Performance.

3. Bisherige Ansätze

In diesem Abschnitt sollen kurz Ansätze besprochen werden, aus denen bei der Implementation von Emerald "gelernt" wurde. Die Vorteile und Nachteile jedes Systems werden genannt.

3.1. Xerox RPC

RPC ist die Abkürzung von Remote Procedure Call, also entfernter Prozeduraufruf. Dieser zuerst von Xerox beschriebene Ansatz [7] ist heute in der Unix-Welt etabliert und wird häufig angewandt. RPC erlaubt dem Programmierer, auf Services zuzugreifen, die ihm auf einem anderen (oder auch lokalen) Rechner angeboten werden. Er verbindet sich explizit mit diesem Service, wobei auf seiner Seite automatisch Funktionen bereitgestellt werden, welche die Aufrufparameter einpacken, den entfernten Service aufrufen, die Rückgabe abwarten und den Rückgabewert zurückliefern. Diese Interfacefunktionen können im folgenden wie eine lokale Prozedur aufgerufen werden, so daß die Entferntheit des Aufrufes dem Programmierer verborgen bleibt.

Als Vorteil von RPC kann genannt werden, daß auch fremde Services, die nicht vom Programmierer selber ausgehen, angesprochen werden können. Durch ein fest definiertes Format für den Datenaustausch kann der RPC auch zwischen hardwarefremden Systemen stattfinden[2] [9]. Die freie Verfügbarkeit einer RPC-Implementation von Sun, die auf der Programmiersprache C aufbaut [8], hat stark zur Verbreitung beigetragen; Sun-RPC ist mittlerweile für alle Unix-Systeme erhältlich, mit entsprechenden Hilfsmitteln wird die Implementation auf Applikationsseite sehr vereinfacht.

Das System hat jedoch mehrere gravierende Nachteile. Der Programmierer muß die exakte Lokation des Service kennen, um sich mit ihm verbinden zu können. RPC ist ein immobiles System, es stellt keine Möglichkeiten zur Prozessmigration zur Verfügung, und auch dem Serviceanbieter ist es nicht erlaubt, sich zu bewegen. Der entfernte Service muß bereits beim Verbindungsaufbau existieren. Eine verteilt arbeitende Applikation muß beim Start jeweils feststellen, ob alle benötigten Services vorhanden sind, und falls dem nicht so ist, bleibt dem Programm keine andere Möglichkeit, als mit einem Fehler abzubrechen. Der schwarze Peter landet wieder beim Programmierer, der mittels umständlicher Betriebssystemaufrufe alle entfernten Services von Hand starten muß.

3.2. EPL

Die Eden Programming Language [4], die ebenfalls an der Universität Washington entwickelt wurde, war ein erster Versuch, die Netzwerktransparenz als festen Teil einer Programmiersprache zu etablieren. Eden bietet wie die Sprache C++ die Objektorientierung als Zugabe zu dem von Concurrent Euclid übernommenen prozeduralen Modell, wobei Objekte frei über Rechnergrenzen hinweg verschiebbar sind.

Dem Programmierer ist es freigestellt, ob Prozeduren als Objekte implementiert werden, und es ist theoretisch möglich, völlig auf Prozeduren zu verzichten. Dies hört sich positiv an, wird aber in der Praxis durch verschiedene Implementationsschwächen erschwert. Jedes Objekt ist in Eden ein eigenständiger Unix Prozeß. Auch kleine Objekte enthalten zusätzlich zu ihrer eigenen Implementation noch Teile des runtime systems zur Kommunikation, so daß die minimale Größe eines Objektes in der Größenordnung 100.000 bytes liegt. Ein zusätzliches Eden Objekt verschlingt daher einen überproportionalen Anteil an Ressourcen.

Zudem wird zum Zugriff auf ein Objekt stets ein allgemeines Modell verwendet. Auch innerhalb eines Rechners werden Daten über rechnerübergreifende Kommunikationskanäle ausgetauscht. Prozeduraufrufe eines Eden Objektes dauern auch innerhalb eines Rechners Millisekunden. Für ein extensiv objektorientiertes System ist diese Performance indiskutabel [2].

---

[2] Mit der Beschreibungssprache XDR (External Data Representation) werden unterschiedliche Datenformate wie z.B. eines Integers automatisch zwischen Systemen umgewandelt.

Auch die Mobilität von Objekten ist eingeschränkt, da Eden keine on-the-fly Bewegung unterstützt. In der Praxis wird ein Objekt, d.h. ein Unix Prozeß, auf einem Rechner zum Absturz gebracht und auf einem anderen Rechner neu gestartet. Falls Zugriffe auf das Objekt ausgeführt wurden, so gehen diese verloren.

3.3. Smalltalk

Smalltalk [6] wird hier wegen seines objektorientierten Ansatzes erwähnt. Insbesondere die Implementation von Polymorphismus in Smalltalk ist interessant. Die Sprache verzichtet völlig auf Typisierung von Variablen, so daß zur Kompilierzeit keine Entscheidungen bezüglich der Konformität getroffen werden können; der Compiler muß jedem Aufruf einen Laufzeittest zur Typüberprüfung voranstellen. Erst bei der Ausführung des Programmes, wenn messages an ein Objekt verschickt werden wird festgestellt, ob das Objekt diese message versteht und im Fehlerfall das Programm abgebrochen.

Dieser Ansatz ist fehlerträchtig, und Programmierung ist aufwendig, da auch Tippfehler jedesmal bei der Ausführung einen Programmabbruch auslösen. Weiterhin muß auf Kosten der Effizienz ein allgemeiner Mechanismus zum Prozeduraufruf verwendet werden.

4. Die Sprache Emerald

Bei der Implementation der Programmiersprache Emerald wurden die Fehler der oben aufgeführten Ansätze berücksichtigt, und es wurde versucht, diese nicht zu wiederholen. Es wurden mehrere Grundsätze festgelegt, denen die Sprache genügen sollte [2]:

- Ein **einheitliches Objektmodell** für alle Sprachkonstrukte. Neben Prozedurobjekten im herkömmlichen Sinne ist auch jeder abstrakte Typ und jeder Prozeß ein Objekt. Das Objektmodell soll **Polymorphismus** ermöglichen.
- **Transparenz des Netzwerkes**. Der Aufruf eines entfernten Objektes unterscheidet sich sprachlich nicht von den Aufrufen lokaler Objekte.
- **Mobilität**. Jedes Objekt darf zu jedem Zeitpunkt über Rechnergrenzen hinweg bewegt werden, oder sich selbst bewegen.
- **Effizienz**. Der Zugriff auf lokale Objekte soll ohne die Verwendung von unnötig aufwendigen Kommunikationsmechanismen stattfinden.

Einzelne dieser Punkte sind an sich nichts neues. Smalltalk besitzt das einheitliche Objektmodell, in EPL ist das Netzwerk transparent, und prozedurale Programmiersprachen wie C++ bieten Effizienz beim Objektzugriff [11]. Die feinkörnige Mobilität einzelner Objekte jedoch wird mit Emerald erstmals propagiert.

Bei der Implementation wurde auf die Effizienz besonderen Wert gelegt. In vielen Systemen werden erst andere Ideale verwirklicht, dann wird an der Effizienz gefeilt. Die Entwickler haben herausgestellt, daß Performance nicht nachträglich geschaffen werden kann; sie muß von Anfang an geplant werden. Es wird propagiert, die Effizienz von Regelfällen zu optimieren, auch wenn dies zusätzliche Probleme in Sonderfällen bereitet. Die aufwendigere Zusatzbehandlung der Sonderfälle wird oftmals durch den Zeitgewinn im Regelfalle mehr als begründet.

4.1. Objekte

Objektorientierte Sprachen wie Smalltalk und das im Objektkonzept von Simula abstammende C++ haben stets das Konzept von *Klassen* verfolgt. An einer Stelle wird die Klasse als Schablone definiert, an anderer Stelle können mit dem Operator *new* Instanzen dieser Klasse erzeugt werden. Dem zugrunde liegt die Idee, daß sich gleich verhaltende Objekte in einer Klasse zusammenfassen lassen können, so daß das Verhalten nur an einer Stelle beschrieben werden muß.

4.1.1. Konstruktion

Emerald folgt einer anderen Philosophie, in der jedes Objekt durch die Ausführung eines *Konstruktors* erzeugt wird. Alle Eigenschaften des Objektes, wie lokale Daten, Definition der Schnittstelle und der Programmtext der Operationen sind Teil des Konstruktors. Bei der Ausführung des Konstruktors entsteht das beschriebene Objekt.

Dies hat natürlich den Effekt, daß zunächst bei der Ausführung eines Konstruktors nur genau ein einziges Objekt erzeugt wird. Da es keine Möglichkeit zum Kopieren gibt, muß zur Herstellung mehrerer Objekte vom gleichen Typ der Konstruktor in eine Schleife gestellt werden, oder es muß ein übergeordnetes Objekt mit der einzigen Intention definiert werden, neue Objekte des untergeordeten Typs zu erzeugen, was natürlich im Programmtext ziemlich häßlich aussieht. Hier ein Beispiel für ein Objekt-Erzeuger-Objekt:

```
const IntegerNodeCreator ← immutable object INC
    export new

    const IntegerNodeType ← type INType
      function getValue → [Integer]
      operation setValue [Integer]
    end INType

    operation new[val : Integer] → [aNode : IntegerNodeType]
      aNode ← object IntegerLiteral
          export getValue, setValue
          monitor
            var value ← val

            operation getValue → [v : Integer]
              v ← value
            end getValue
            operation setValue [v : Integer]
              value ← v
            end setValue
          end monitor
        end IntegerLiteral
    end new
  end INC
```

Programm 1 - Ein Objekt-Erzeuger-Objekt

Programm 1 erzeugt das neue Objekt *IntegerNodeCreator*. Dieses exportiert die Operation *new*, welche beim Aufruf das Objekt *IntegerLiteral* erzeugt. Letzteres exportiert Funktionen, um den Wert des Literals zu setzen und zu lesen, verhält sich also wie eine ganz normale Variable. Die Operation new hat einen deklarierten abstrakten Rückgabetyp *IntegerNodeType*, ebenfalls mit den genannten Operationen. Ein *IntegerLiteral* ist daher typkonform zu *IntegerNodeType*, und darf deshalb letzteren Typ annehmen. Es sei nochmals darauf hingewiesen, daß obige Prozedur der einzige Mechanismus ist, mehrere Objekte eines Typs zu erzeugen.

4.1.2. Referenzen

Wie beschrieben ist ein Konstruktor der einzige Weg um ein Objekt zu erzeugen. Bei der Konstruktion wird für jedes Objekt eine netzwerkweit eindeutige Identifikation generiert. Es ist sichergestellt, daß sich ein Objekt immer genau auf einem Rechner befindet, also eine feste Lokation besitzt. Alle Zugriffe auf ein Objekt erfolgen über Referenzen; bei jeder "Weitergabe" des Objektes, z.B. in einem Prozeduraufruf, wird nur eine weitere Referenz auf das Objekt erzeugt. Die Tatsache, daß ein Objekt nicht kopiert wird, erspart ansonsten unvermeidbare Konsistenzprobleme.

4.1.3. Prozesse

Ein Objekt kann zusätzlich zu der Beschreibung seiner Schnittstelle einen optionalen Prozeß enthalten. Dieser wird bei der Ausführung des Objektkonstruktors asynchron gestartet, und läuft parallel zu anderen Prozessen. Ein Objekt mit Prozeß wird als *aktiv* bezeichnet, im Gegensatz zu *passiven* Objekten ohne Prozeß. Der Prozeß darf seinerseits nach belieben neue Objekte erzeugen, andere Objekte aufrufen etc. Diese Variante des *Multithreading* fordert Synchronisationsprimitiven, da mehrere Prozesse gleichzeitig auf ein Objekt zugreifen können.

Im Gegensatz zu vereinbarter Synchronisation mittels *Semaphoren* sind Objekte in Emerald in einen *bewachten* (monitored) und einen unbewachten Teil unterteilt. Prozeduren innerhalb des bewachten Teils haben exklusiven Zugriff auf die Variablen im bewachten Teil, und Prozeduren des unbewachten Teils ist es nicht erlaubt, auf bewachte Variablen zuzugreifen. Im obigen Beispiel befinden sich die Operationen *setValue* und *getValue* sowie die Variable *val* im bewachten Teil des Objektes. Das äußere Objekt *INC* hat keinen bewachten Teil, da die Operation *new* unkritisch ist. Der Programmierer muß jedoch selber dafür sorgen, daß alle kritischen Operationen innerhalb eines bewachten Teils stattfinden.

4.1.4. Unveränderliche Objekte

Falls ein Objekt keinen inneren Zustand hat, oder sich über die Zeit hinweg nicht verändert, so kann der Programmierer das Objekt als *unveränderlich* (immutable) deklarieren. Prozeduren eines unveränderliches Objektes erzeugen bei Eingabe bestimmter Werte stets den gleichen Rückgabewert, sie sind also Funktionen im mathematischen Sinne. Der Sinn dieser Deklaration ist, daß das System bei einem unveränderlichen Objekt weiß, daß es keine Konsistenzprobleme mit Kopien geben kann. Falls auf ein entferntes unveränderliches Objekt zugegriffen wird, so kann dieses einfach auf den lokalen Rechner kopiert werden. Prinzipiell sind unveränderliche Objekte auf allen Rechnern lokal, und überall kann daher mit lokalen Mechanismen auf das Objekt zugegriffen werden.

4.1.5. Polymorphismus

Emerald unterstützt Polymorphismus über die anfangs erwähnte Konformitätsrelation [3]. Falls sich ein Objekt zu einem anderen Typkonform verhält, so kann es auch den anderen Typ annehmen. Andererseits erlaubt Emerald keine Vererbung und deshalb auch keine Überladung von Eigenschaften. In typkonformen Objekten muß deshalb der Programmcode gleicher Prozeduren mehrmals aufgeführt werden.

4.2. Sprachkonstrukte

4.2.1. Bewegen von Objekten

In Emerald existieren 4 Befehle, um Objekte zu bewegen. Zwar ist eine völlige Transparenz des Netzwerkes in vielen Fällen von Vorteil, in anderen Fällen ist es allerdings auch nützlich, eine Kontrolle über die Lokation von Objekten zu haben, z.B. um das System bei der Ressourcenverteilung zu unterstützen. Der Programmierer verteilter Anwendungen kann mit diesen Primitiven helfen, die Anzahl entfernter Objektaufrufe zu minimieren [2].

- **move X to Y** bewegt das Objekt X auf den Rechner, auf dem sich Objekt Y befindet. Da auch ein Rechner ein Objekt ist, ist es ebenfalls möglich, das Objekt explizit auf einen benannten Rechner zu bewegen.
- **fix X at Y** bewegt Objekt X ebenfalls auf den Rechner, auf dem sich Objekt Y befindet, verhindert allerdings, daß X wieder von dem Rechner fortbewegt wird.
- **unfix X** hebt den vorigen Befehl wieder auf
- **refix X at Y** hebt ein fix auf, bewegt das Objekt und führt ein neues fix aus.

Der Befehl **move** ist ziemlich schwach, das System ist noch nicht einmal verpflichtet, das Objekt zu bewegen. Überhaupt ist der Befehl eine ziemlich unsichere Sache. Während der Ausführung eines move könnte sich Y beispielsweise selber bewegen, oder ein zweiter move-Befehl könnte X sofort an einen anderen Rechner weiterleiten.

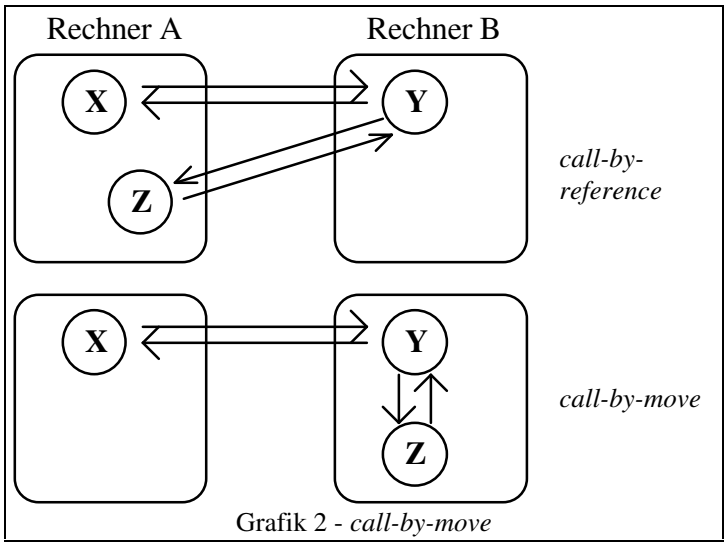
Die restlichen drei Befehle sind stärker, allerdings muß der Programmierer auch hier mehrere Fälle bedenken. Zwar kann X auf den Rechner von Y fixiert werden, doch Y kann sich noch immer bewegen. Erst die Kombination "*fix Y at Y, fix X at Y*" stellt sicher, daß sich anschließend beide Objekte auf dem gleichen Rechner befinden.

4.2.2. Gruppenbildung

Bei der Bewegung eines Objektes muß auch entschieden werden, wieviel bewegt werden soll. Offensichtlich ist es nicht sinnvoll, nur das einzelne Objekt auf den neuen Rechner zu verschieben. Falls dem bewegten Objekt lokale Variablen nicht mitbewegt werden, so erfordert jeder weitere Zugriff einen aufwendigen entfernten Prozeduraufruf. Lokale Variablen, die nicht außerhalb eines Objektes verändert werden, können daher unmittelbar mitbewegt werden.

In anderen Fällen ist die Entscheidung schwieriger, z.B. bei Objekten, die dem bewegten Objekt nahestanden, d.h. zwischen denen viele Daten ausgetauscht werden. Soll das zweite Objekt bei der Bewegung gleich mitgeschickt werden, oder wird das erste Objekt beim nächsten Aufruf wieder zurückgeholt? Emerald bietet dem Programmierer das Konzept der *Zugehörigkeit*, um die Relationen zu beschreiben. Eine Variable kann als *zugehörig* (attached) definiert werden. Wird ein Objekt bewegt, so folgen ihm alle Objekte, die mit einer zugehörigen Variable referenziert werden.

4.2.3. Parameterübergabe



Grafik 2 - *call-by-move*

Wie bereits erwähnt werden bei einem Prozeduraufruf nur Referenzen eines Objektes übergeben. Falls sich das aufrufende und das aufgerufene Objekt auf verschiedenen Rechnern befinden, hat das zur Folge, daß das aufgerufene Objekt für jeden Zugriff auf die Parameter wieder einen entfernten Zugriff ausführen muß (Grafik 2 oben). Analog befindet sich das Ergebnis der Prozedur bezüglich dem lokalen Objekt auf einem entfernten Rechner, und die aufrufende Prozedur muß stets mit entfernten Mechanismen auf den Rückgabewert zugreifen. Es kann von Vorteil sein, die Prozedurparameter beim Aufruf auf den entfernten Rechner und die Rückgabewerte auf den Rechner des Aufrufers zu bewegen. In Grafik zwei wird eine einfache Situation dargestellt. Objekt X ruft eine Prozedur des Objektes Y auf, und übergibt das lokale Objekt Z als Parameter, welches von Y genau einmal verwendet wird. Falls Z nicht bewegt wird, so müssen vier rechnerübergreifende Sprünge vollzogen werden, falls Z mitbewegt wird, nur zwei.

Die Sprache stellt dem Programmierer drei Varianten des klassischen *call-by-reference* zur Verfügung. Alle Varianten sind optional, der Programmierer kann die Entscheidung auch dem Compiler oder dem Betriebssystem überlassen. Die Entscheidung, einen Parameter zu bewegen hängt primär von zwei Faktoren ab; einerseits von der Größe des Parameters, andererseits von der Anzahl der Zugriffe des aufgerufenen Objektes auf den Parameter.

- *call-by-move* bewegt die Prozedurparameter auf den Rechner, auf dem sich das aufgerufene Objekt befindet. Diese Entscheidung kann für einzelne Parameter getroffen werden.
- *call-by-visit* entspricht *call-by-move*, allerdings werden die Prozedurparameter bei Beendigung des Aufrufes zurück zum Aufrufer bewegt.
- *call-by-move-return* bewegt die Rückgabewerte der Prozedur zum Aufrufer und kann mit den anderen beiden Varianten gemischt werden.

Alle drei Varianten bewegen die entsprechenden Objekte mit dem *move*-Befehl, was bedeutet, daß die vom Programmierer beschriebenen Übergabemöglichkeiten nicht explizit befolgt werden müssen. Wiederum ist es dem Compiler und Betriebssystem freigestellt, auf die Bewegung zu verzichten.

4.3. Weitere Möglichkeiten in Emerald

Die Sprache besitzt noch einige aus der Verteiltheit resultierende Eigenschaften, die hier kurz Erwähnung finden sollten, insbesondere netzwerkbezogene Sprachmöglichkeiten, mit denen ein Programmierer auf die Dynamik des Netzes reagieren kann.

In verteilten Umgebungen können Rechner unabhängig voneinander ausfallen, und Objekte, die sich auf diesem Rechner befanden, werden unerreichbar. Ein Programmierer kann für jedes Objekt einen *unavailable handler* definieren, der bei dem Versuch, ein unerreichbares Objekt auszuführen, aufgerufen wird. Analog kann ein *failure handler* definiert werden, um Fehler wie division by zero, Stapelüberlauf oder at-runtime festgestellte ungültige Prozeduraufrufe abzufangen. Falls in einem Objekt ein Fehler auftritt, dieses jedoch keine Fehlerbehandlung definiert, so wird der Fehler so lange zurückgegeben, bis ein fehlerbehandelndes Objekt erreicht wird.

Eine Anwendung kann darüber hinaus einen *NodeEventHandler* installieren, der jedesmal aufgerufen wird, wenn ein Rechner des Netzwerkes ausfällt oder erneut verfügbar wird. Die Anwendung kann mit dieser Information flexibel auf Veränderungen reagieren, und Aufgaben auf neue Rechner auslagern, oder die Bewegung von Objekten auf unsichere Rechner verhindern.

5. Implementation

In diesem Abschnitt werden die Mechanismen vorgestellt, mit denen Emerald viele der angesprochenen Probleme umgeht oder meistert. Einige schwerwiegende Probleme, die sich aus der Mobilität ergeben, werden erst bei dieser Diskussion aufgedeckt.

5.1. Betriebssystem-Grundlagen

Das Emerald Betriebssystem ist ein einzelner Unix-Prozeß, der parallel zu anderen Unix-Anwendungen auf einem Rechner laufen kann. Auf jedem Rechner des Netzwerkes muß ein solches Betriebssystem von Hand gestartet werden. Das Betriebssystem verwaltet lokal alle Emerald-Objekte, führt Kontextwechsel zwischen Emerald-Prozessen aus und kümmert sich beim Aufruf entfernter Objekte um die explizite Lokation und den Datenaustausch. Das Aufsetzen auf dem Unix-System bringt natürlich einige Vorteile für die Implementation des Betriebssystems, das hardwarenahe Aufgaben wie I/O nicht selbst zu definierten braucht.

Innerhalb eines Rechners teilen sich Emerald-Objekte einen Adreßraum, der vom Betriebssystem dynamisch verwaltet wird. Theoretisch sind Sicherheitsprobleme zu befürchten, doch garantiert die Sprache mit dem objektorientierten Konzept und Abfragen zur Laufzeit, daß keine ungültigen Zugriffe auftreten können. Dieses Teilen des Adreßraumes trägt auch erheblich zur Performancesteigerung bei, da Objekte innerhalb eines Rechners direkte Lesezugriffe und Prozeduraufrufe in andere Objekte tätigen können. Objekte in eigenen Adreßräumen wären besser geschützt, doch Datenaustausch müßte aufwendig über *shared Memory* und *Messages* erfolgen [12].

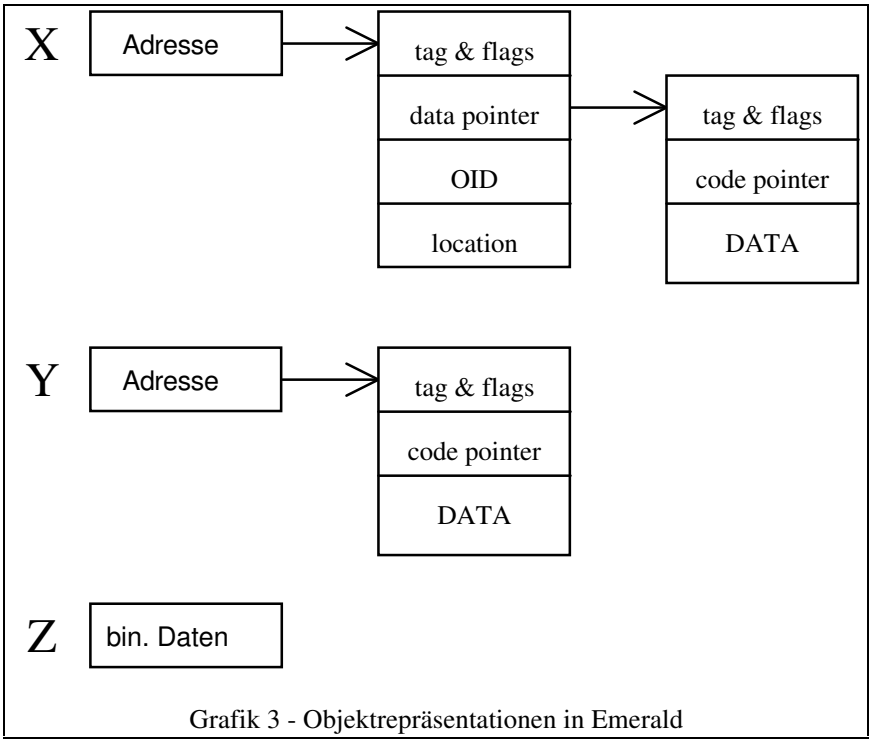
Generell sind der Emerald-Compiler und das Betriebssystem eng gekoppelt. In vielen Fällen wird der Compiler Code erzeugen, der direkt auf Strukturen des Betriebssystems zugreift, was wiederum ("it's not a bug, it's a feature") aufgrund der Konzeption der Sprache keine Sicherheitslücke ist, sondern vielmehr hilft, die Performance zu verbessern. Weiterhin erzeugt der Compiler ausführliche Tabellen, die das Betriebssystem zur Vereinfachung und Ermöglichung von Mobilität benötigt.

Der Emerald-Compiler erzeugt direkt ausführbare Maschinenbefehle, die vom Betriebssystem in den Speicher geladen, und dort direkt von der Hardware des Rechners ausgeführt wird.

5.2. Objekte

Alle Objekte in Emerald werden mit dem gleichen Mechanismus eines Objektkonstruktors erzeugt, allerdings sind die erzeugten Maschinenbefehle nicht immer gleich. Obwohl Emerald netzwerkunabhängige Zugriffsmechanismen anbietet, wird diese Eigenschaft nur von einer relativ kleinen Zahl von Objekten benötigt. Innerhalb des Emerald-Betriebssystems gibt es vier Modelle verschiedenen Aufwandes und Komplexität, und zur Übersetzungszeit wählt der Compiler aus, welches Modell im jeweiligen Fall ausreicht.

- Ein *globales* Objekt kann unabhängig bewegt und kann von allen Rechnern des Netzwerkes angesprochen werden. Ein Zugriff auf ein globales Objekt benötigt eventuell einen entfernten Prozeduraufruf.
- Ein *lokales* Objekt ist vollständig in ein anderes Objekt eingebettet; Referenzen eines lokalen Objektes werden niemals aus dem beinhaltenden Objekt heraus exportiert. Ein solches Objekt wird immer mit dem beinhaltenden Objekt mit verschoben. Ein Zugriff auf ein lokales Objekt kann immer inline[3] erfolgen.
- Ein *unveränderliches* Objekt kann ohne Probleme auf jeden Rechner kopiert werden. Falls ein unveränderliches Objekt nicht lokal ist, so wird eine lokale Kopie angelegt. Der weitere Zugriff kann wie beim lokalen Objekt inline erfolgen.
- Ein *direktes* Objekt ist eine einfache Datenstruktur, deren Organisation zur Kompilierzeit vollständig erschlossen werden kann. Prozeduraufrufe in ein direktes Objekt können durch Zugriffe auf die Datenelemente ersetzt werden.



Grafik 3 - Objektrepräsentationen in Emerald

Grafik 3 zeigt die interne Repräsentation von Objekten. Objekt X ist global; der Wert in X ist ein Zeiger auf einen internen *Object Descriptor*. Auf einem Rechner existiert eine Tabelle mit allen Deskriptoren aller globalen Objekte, die dem Rechner bekannt sind. An den Flags am Anfang des Deskriptors erkennt das System, daß es sich um den Deskriptor eines globalen Objektes handelt. In den Flags ist u.a. vermerkt, ob sich das Objekt auf dem lokalen Rechner oder auf einem entfernten Rechner befindet. Im ersten Falle, wie er in der Grafik dargestellt ist, enthält der *data pointer* einen Zeiger auf das Objekt im lokalen Adreßraum, der Zugriff kann also lokal abgewickelt werden. Ist das

[3] Direkt. Es ist kein Umweg über das Betriebssystem notwendig.

Objekt nicht lokal, so wendet sich das System mit der eindeutigen *OID*[4] an das Betriebssystem des in *location* angegebenen Rechner, um einen entfernten Prozedurauf auszuführen.

Y ist ein lokales Objekt, die Adresse in Y zeigt direkt auf das Objekt selber. Anhand der Flags erkennt das System die Lokalität und führt inline-Aufrufe aus. Objekt Z ist ein direktes Objekt wie z.B. eine einfache Ganzzahl; Z enthält keine Adresse auf die Daten sondern die Daten selber.

Insgesamt muß erst bei dem Aufruf eines globalen Objektes das Emerald-Betriebssystem um Hilfe gebeten werden. Der Compiler erzeugt einen Test, der in den Flags des Objektheaders abfragt, ob das Objekt lokal ist. Ist der Test erfolgreich, so wird der Aufruf unter Umgehung des Systems ausgeführt. In vielen Fällen, wie z.B. bei unveränderlichen oder lokalen Objekten kann der Compiler auch entscheiden, daß das Zielobjekt immer lokal ist, und kann daher den Test wegfallen lassen.

Beim Zugriff auf unveränderliche Objekte fügt der Compiler weiterhin einen Test ein, ob bereits eine lokale Kopie existiert. Schlägt der Test fehl, so wird das Betriebssystem angewiesen, eine Kopie des Objektes zu finden und eine neue lokale Kopie zu erzeugen.

5.3. Mobilität

Mobilität schafft eine neue Dimension von Problemen. Vormals lokale Zugriffe müssen auf einmal über Rechnergrenzen hinweg ausgeführt werden, sowohl in das Objekt hinein als auch aus dem Objekt heraus. Ein schwerwiegendes Problem sind Prozesse. Falls das sich bewegende Objekt einen Prozeß besitzt, so muß dieser eingefroren und auf dem neuen Rechner an der gleichen Stelle fortgesetzt werden. Gleiches gilt für externe Prozeduraufrufe, die in dem Objekt momentan aktiv sind. Dadurch wird es möglich, daß ein ehemals lokaler Prozeduraufruf plötzlich eine entfernte Rückkehr aus der Prozedur erfordert. Auf dem Stack und in den Prozessorregestern befinden sich Referenzen, sprich explizite Hardwareadressen, auf ehemals lokale Objekte, die auf einmal nicht mehr lokal sind. Weiterhin, da sich Emerald-Objekte einen Adreßraum teilen, wird das Objekt auf dem Zielrechner eine andere Hardwareadresse erhalten wie zuvor. Das Objekt muß daher *reloziert* werden, d.h. alle Sprungadressen, auch die innerhalb eines Objektes, müssen angepaßt werden.

Insgesamt kann die Bewegung eines Objektes sehr aufwendig sein. Die sich ergebenden Probleme werden von Emerald hingenommen. Compilergenerierte Tabellen, die das Layout eines Objektes[5] beschreiben, helfen dem Betriebssystem, ein Objekt zu relozieren. Der Stack und Prozessorregister werden nach Objektreferenzen abgesucht; und Referenzen auf ehemals lokale Objekte werden durch Referenzen auf die entsprechenden Objektdeskriptoren des neuen Systems ersetzt. Analog können natürlich auch ehemals entfernte und nun lokale Referenzen durch direkte Adressen ersetzt werden.

Bei der Bewegung von Objekten werden in den Tabellen des Quellrechners und des Zielrechners automatisch und sofort die entsprechenden neuen Zustände eingetragen, d.h. Flags und Pointer werden gemäß Grafik 3 umgehängt. Drittobjekte, die an der Bewegung nicht beteiligt waren, können sich auf diese Weise sofort auf die neue Situation umstellen.

5.4. Code Sharing

Obwohl für die Erzeugung eines Objektes ein individueller Objektkonstruktor ausgeführt wird, teilen sich dennoch Objekte vom gleichen Typ den gleichen Maschinencode. Da sich der Code nicht zur Laufzeit ändern kann, wird er vom System wie ein unveränderliches Objekt behandelt. Falls ein Objekt bewegt wird, so werden zunächst nur die Daten des Objektes an den Zielrechner übertragen. Der Zielrechner stellt fest, ob bereits eine lokale Kopie des Pro-

---

4 Objekt Identification; Eine systemweit (d.h. rechnerübergreifend) eindeutige Identifikation wird benötigt, um ein entferntes Objekt zu finden. Das Betriebssystem eines Rechners pflegt eine Tabelle mit den OIDs aller lokalen und von lokalen Objekten referenzierten entfernten Objekten.

5 Damit ist hier gemeint, daß die Tabelle alle Adressen im Objekt beschreibt, die bei der Relokation angepaßt werden müssen.

grammcodes existiert; falls nicht, so wird eine Kopie vom Quellrechner angefordert. Prinzipiell könnte der Programmcode auch sofort mit den Daten mitgeschickt werden, allerdings kann angenommen werden, daß der Code öfter bereits auf dem Zielsystem vorhanden ist als nicht. Dieses Prinzip der *lazy evaluation* spart im Durchschnittsfall also Zeit.

5.5. Auffinden von Objekten

Der Aufenthaltsort von Objekten ist in Emerald nicht immer bekannt. Globale Objekte können zu jedem Zeitpunkt auf einen anderen Rechner migrieren. Da das Objekt selber keine Kenntnis davon hat, wie viele und welche anderen Objekte es referenzieren, ist es nicht möglich, diese Referenzen stets auf dem neuesten Stand zu halten. Auch wenn diese Information bekannt wäre, würde der Prozeß des Erneuerns größtenteils redundant sein; das Objekt kann sich schließlich mehrfach bewegen, ohne daß ständig alle Referenzen benötigt werden.

Emerald befürwortet ein System mit weitergeleiteten Adressen (forwarding addresses). Beim Aufruf eines globalen, nicht residenten Objektes wird die *location*-Information im Objektdeskriptor (siehe Grafik 3) ausgewertet. Bei diesem Rechner wird angefragt, ob das Objekt mit der bekannten OID dort vorhanden ist. Im Erfolgsfall ist damit alles erledigt. Falls sich das Objekt jedoch weiterbewegt hat, so befragt das Zielsystem seine eigene Datentabelle, und schickt die Anfrage an den dort notierten Rechner weiter. Irgendwann wird das Objekt gefunden. Parallel zum eigentlichen Prozeduraufruf wird eine Meldung an den aufrufenden Rechner zurückgeschickt, die die aktuelle Lokation des Objektes enthält. Diese Meldung läuft auf dem gleichen Pfad wie der ausgesandte Aufruf, so daß alle Rechner in der Kette ihre Information auffrischen können. Zukünftige Aufrufe können nun direkt, ohne den Umweg über andere Rechner, ausgeführt werden.

5.6. Traffic Reduction

Beim Aufruf eines Objektes auf einem entfernten Rechner müssen die Emerald Betriebssysteme auf beiden Rechnern Daten austauschen, was mit einem auf UDP basierenden, sicheren Protokoll[6] geschieht. Es ist bekannt, daß der Overhead bei der Versendung eines Datenpaketes im Vergleich mit der Zeitdauer für die eigentliche Datenübertragung relativ groß ist. Es ist daher deutlich effizienter, ein großes Paket als viele kleine Pakete zu verschicken. Diese Tatsache wird insbesondere bei der Aufruf einer entfernten Prozedur mittels *call-by-move* (siehe Grafik 2) verwendet. Das verwendete Aufrufprotokoll, also der Befehl an das entfernte Betriebssystem, eine Prozedur auszuführen, ist relativ kompakt und läßt noch viel freien Platz in einem Paket (die maximale Paketgröße von ca. 1500 bytes ist von UDP vorgeschrieben). Dieser Platz wird genutzt, um Prozedurparameter mit dem Aufrufbefehl huckepack mitzuschicken. Sofern die Parameter passiv und kleiner als ca. 1000 bytes sind, entsteht kaum zusätzlicher Aufwand bei der Datenübertragung, und der zusätzliche Overhead im Betriebssystem, immerhin muß das Objekt aus den Tabellen des einen Rechners gelöscht und in die des anderen integriert werden, wird im Regelfall durch die verminderte Zahl von rechnerübergreifenden Zugriffen auf das Objekt belohnt.

5.7. Zusammenfassung

Im Gegensatz zum traditionellen Verständnis eines Compilers, der eine strikte Umsetzung von Programmtext in Maschinencode vornimmt, ist der Compiler des Emerald-Systems völlig anders konstruiert. Der Programmtext wird global auf seine Zusammenhänge hin analysiert, und auf der Basis dieser Informationen werden jeweils optimale Zugriffsstrategien erzeugt. Das Betriebssystem verläßt sich völlig auf den Compiler. Das Betriebssystem erhält vom Compiler Daten, die zur Behandlung von Objekten nötig sind und gestattet dem erzeugten Maschinencode zwecks besserer Performance die Manipulation von systeminternen Strukturen. Weiterhin ist der Compiler verpflichtet, an geeigneten Stellen Sprünge ins Betriebssystem vorzunehmen, um z.B. einen Kontextwechsel auszulösen.

An vielen Stellen des Designs fließen performance-orientierte Überlegungen ein, wie z.B. die Entscheidung, kleine Parameter vorausschauend, meist auch ohne explizite Anweisung des *call-by-move* vom Programmmierer, huckepack

---

6 UDP ist primär ein unsicheres Protokoll, welches keine *exactly-once* Übertragung gewährleistet. Pakete können verloren gehen oder mehrfach beim Empfänger ankommen. Wenn ein Paket ankommt, so ist allerdings seine Korrektheit gewährleistet, da Übertragungsfehler entdeckt werden.

mit dem Prozeduraufruf mitzuschicken. Ein gutes Beispiel für die Philosophie bei der Entwicklung von Emerald, den Regelfall auch dann zu optimieren, wenn der Spezialfall aufwendiger wird, ist die Entscheidung, daß sich alle Objekte auf einem Rechner den gleichen Adreßraum teilen. Zwar wird das Bewegen von Objekten erheblich kompliziert (bei eigenen Adreßräumen wäre eine Relokation, d.h. das Umrechnen von Sprungadressen, nicht notwendig), doch bei lokalen Aufrufen wird viel Zeit gespart. Daß lokale Prozeduraufrufe um Größenordnungen häufiger sind als Objektbewegungen rechtfertigt den zusätzlichen Aufwand.

Die Entwickler von Emerald können die Effizienz der Sprache auch eindrucksvoll bestätigen. In lokalen Umgebungen wird fast die gleiche Performance wie von einer prozeduralen Sprache erreicht, und eine deutlich höhere Performance als vergleichbare objektorientierte Systeme wie Smalltalk. In verteilten Umgebungen existieren noch nicht allzuviele Vergleichsmöglichkeiten. Die Migrationszeiten und entfernte Aufrufzeiten von EPL werden allerdings deutlich unterboten. Timing-Analysen zeigen, daß bei einem entfernten Prozeduraufruf nur 12% der Zeit auf durch Verteilung entstehenden Overhead zurückzuführen sind, 88% der Zeit wird allein für den Austausch von Datenpaketen benötigt [2].

6. Resumée

Die Sprache Emerald stammt ursprünglich aus dem Jahre 1986. Verschiedene Dissertationen haben sich mit verteilter Programmierung auf Basis von Emerald beschäftigt, doch neuere Dokumente als aus dem Jahre 1989 waren nicht aufzufinden. Obwohl Beschreibung und Implementation der Sprache frei verfügbar sind, sind keine Applikationen bekannt, die größere Aufgaben als Tests vollbringen. Es muß also die Frage gestellt werden, was bei dem Design der Sprache falsch gemacht wurde, warum sich Emerald nicht besser verbreitet hat.

Von der Konzeption hat Emerald durchaus viele Vorteile. Bis heute ist Programmierung auf einem transparenten Netzwerk noch nirgendwo so elegant und effizient gelöst wie in dieser Sprache. Auch kann es kaum an der Objektorientierung liegen, schließlich feiert die ebenfalls objektorientierte Sprache C++ gerade ihren Siegeszug.

Die ablehnende Haltung der Sprache gegenüber dürfte vielmehr von der Syntax stammen. Die Syntax entspricht keiner verbreiteten Sprache, und die Umstellung ist vergleichsweise umständlich. Das verbreitete Klassenkonzept wurde nicht implementiert, was mit nicht nachvollziehbaren Erklärungen begründet wird. Die Tatsache, daß Deklaration des Interfaces und Implementation der Prozeduren fest verkoppelt sind, fördert nicht gerade die Übersichtlichkeit. Für jeden Typ von Objekt ein Objekt-Erzeuger-Objekt definieren zu müssen, erschwert den Überblick weiter. Das in Programm 1 gezeigte Beispiel zeigt auch eindrucksvoll, wieviel Typdeklarationen und Namen benötigt werden, einfach nur um mehrere Objekte gleichen Typs zu erzeugen.

Auch schmerzt die fehlende Unterstützung von Vererbung. Der Programmierer wird dadurch in vielen Fällen zur Wiederholung von Programmtext gezwungen, da gleich zu implementierende Prozeduren in allen Objekten neu programmiert werden müssen, anstatt daß die Prozeduren vom Vorgänger ererbt werden.

Ein Nachteil ist weiterhin, daß alle Aktionen von einer zentralen Stelle ausgelöst werden müssen. Eine Applikation, die auf einem Rechner gestartet wird, kann sich selber zwar auf beliebig viele Rechner ausbreiten; trotzdem ist es nicht möglich, Services anderer Applikationen in Anspruch zu nehmen oder selber Services anzubieten. Hierzu müßten zur Laufzeit externe Objekte angebunden werden, was von der Sprache nicht unterstützt wird.

Und gerade dies ist ein Jammer, denn die Möglichkeiten einer ORB[7]-Architektur auf Basis einer verteilten Sprache sind überzeugend. Beim Zugriff auf einen Service würde sich der Serviceanbieter, üblicherweise ein unveränderliches Objekt, auf den Rechner des Klienten, oder zumindest einen Rechner im lokalen Netz, kopieren, dort seine Aufgaben lokal mit besserer Performance verrichten, und würde nach Abschluß der Dienstleistung automatisch wieder gelöscht werden.

Ein letzter Nachteil von Emerald ist unvermeidlich; alle angeschlossenen Rechner im Netzwerk müssen die gleiche Hardware, insbesondere den gleichen Prozessor, verwenden. Da Emerald-Prozeduren Assemblerbefehle sind, und der Code der Prozeduren zwischen Rechnern ausgetauscht werden muß, dürfen keine fremden Plattformen stören. Wiederum kann dieser Nachteil als Vorteil verstanden werden, da aufwendige Konvertierungen wie bei RPC und XDR hier nicht nötig sind.

Ein Wort noch zur Sicherheit. Zwar garantiert der Emerald-Compiler die Generierung von betriebssystemsicheren Code, doch ist dies ein Schutzmechanismus, der schnell umgangen werden kann. Unix-Systemadministratoren würden die Hände über dem Kopf zusammenschlagen, wenn es einem Anwender möglich ist, mit ein paar Maschinenbefehlen das gesamte Betriebssystem auszulesen und zu verändern. Die strenge Abkapselung des Betriebssystemes gegen mutwilliges Einbrechen, wie sie in Unix vorhanden ist [12], existiert in Emerald überhaupt nicht. Weiterhin verfügt Emerald bislang noch nicht über Rechnerschutzmechanismen. Jede Applikation kann sich auf alle Rechner verteilen, auf denen ein Emerald-Betriebssystem läuft. Ein "Virus", der sich selber auf alle Rechner kopiert, ist nur wenige Zeilen lang. Zwar sind diese beiden Punkte innerhalb eines verläßlichen, beschränkten Netzwerkes nicht von Belang; in großen Datennetzen mit unkontrollierbaren Benutzerzugriffen aber ist die Anwendung nicht mehr vertretbar.

Trotz alledem hat Emerald gezeigt, daß ein generelles Objektmodell, Netzwerktransparenz und Performance keine Widersprüche sind, sondern gemeinsam existieren können. Bei dem Vergleich der Syntax wurden oft Vergleiche zur bequemeren Handhabung von Objekten in C++ hergestellt, wobei berücksichtigt werden muß, daß Emerald zeitlich vor C++ entstanden ist. Es wäre sicherlich interessant, zu versuchen, die Vorteile beider Sprachen zu verbinden; die Intelligenz und Flexibilität des Emerald-Compilers und die gewohnten Syntaxregeln von C++.

7. Bemerkung

Eine Beispielimplementation der Sprache existiert für die Architekturen ULTRIX, SUN und HP9000. Im Rahmen dieser Ausarbeitung wurde versucht, das in K&R-C geschriebene System zu übersetzen, was jedoch aufgrund von Komplikationen schnell aufgegeben wurde. Es wurde jedoch offensichtlich, daß ein genaues Layout der beteiligten Rechner in das System hineinkompiliert werden mußte -- ein offensichtlich unbefriedigender Zustand. Vorteil oder Nachteil, jedenfalls könnte ein kompiliertes System nur von priviligierten Usern (root) gestartet werden.

---

[7] Object Request Broker -- ein Verwalter für Dienstleistungen verschiedener Server, auf die ein Client zugreifen kann.

8. Literaturverzeichnis

[1] Norman C. Hutchinson. *Emerald: An Object-Based Language for Distributed Programming*. PhD thesis, Technical Report 87-01-01, Department of Computer Science, University of Washington, Seattle, January 1987.

[2] Eric Jul. *Object Mobility in a Distributed Object-Oriented System*. PhD thesis, Technical Report 88-12-06, Department of Computer Science, University of Washington, Seattle, December 1988.

[3] Rajenda K. Raj, Ewan Tempero, Henry M. Levy, Andrew P. Black, Norman C. Hutchinson, Eric Jul. *Emerald: A General-Purpose Programming Language*. Software Practice and Experience, 1990.

[4] A. P. Black. *The Eden Programming Language*. Technical Report 85-09-01, Department of Computer Science, University of Washington, Seattle, September 1985.

[5] Andrew P. Black, Norman C. Hutchinson, Eric Jul, Henry M. Levy and Larry Carter. *Distribution and Abstract Types in Emerald*. IEEE Transactions on Software Engineering, 13(1):65-76, January 1987. Also Technical Report 86-02-04, Department of Computer Science, University of Washington, Seattle, February 1986.

[6] Glenn Krasner. *Smalltalk-80: Bits of History, Words of Advice*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1983.

[7] Courier. *The Remote Procedure Call Protocol*. XEROX Corporation, XSIS 038112, December 1981.

[8] *RPC: Remote Procedure Call Protocol Specification*. Sun Microsystems, RFC-1050, April 1988.

[9] *XDR: External Data Representation Standard*. Sun Microsystems, RFC-1014, June 1987.

[10] *NFS: Network File System Protocol Specification*. Sun Microsystems, RFC-1094, March 1989.

[11] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Reading, Massachusetts, March 1986.

[12] Andrew S. Tannenbaum. *Operating Systems, Design and Implementation*. Prentice Hall International, London, 1990