Eine Java Virtual Machine für Aos

Autor: R. Laich
Diplomarbeit am Institut für Computersysteme ETH Zürich
ETH-Zentrum, CH-8092 Zürich

Betreuung: P. Reali, Prof. J. Gutknecht

Wintersemester 2000/2001

Zusammenfassung

Die vorliegende Arbeit befasst sich mit der Konstruktion einer Java Virtual Machine (*JVM*) für das Active-Oberon-System (*Aos*).
Ein bestehendes Gerüst einer JVM für Native-Oberon (vergleiche [2]) wurde weiterentwickelt und auf Aos portiert. Es wurden Konzepte für Interfaces, Exception-Handling sowie für das Multithreading in dieser JVM erarbeitet und in einer Implementierung umgesetzt. Teile des JDK 1.2 wurden für diese JVM implementiert.
Mit der aktuellen Version der JVM können beliebige Java-Programme ausgeführt werden, welche nur die bereits implementierten Funktionalitäten des JDK's benutzen.
Üblicherweise läuft eine JVM auf einem bestimmten Betriebssystem und hat ihre eigenen Laufzeitstrukturen. Die JVM für Aos setzt direkt auf den Laufzeitstrukturen von Aos auf. Java-Objekte und Aos-Objekte besitzen dieselbe Struktur. Das erlaubt ein hohes Mass an Interopabilität zwischen Java und Active-Oberon.
Die JVM für Aos ist deutlich langsamer als die JVM von SUN. Bis jetzt wurden noch keine Optimierungen vorgenommen.

Inhaltsverzeichnis

Kapitel 1

Einführung

1.1 Java

Java stellt eine Programm-Plattform dar. Diese besteht aus der *Programmiersprache Java* [13], einer *virtuellen Maschine* (vergleiche Abschnitt 2.2), welche "übersetzte" Javaprogramme ausführen kann, sowie einer Menge von *Bibliotheken* mit standardisierten Eigenschaften.
Java wurde Ende 1995 von SUN Microsystems eingeführt. Seither hat die Java-Plattform eine riesige Verbreitung erfahren.
Java wurde ursprünglich konzipiert, um Software für vernetzte elektronische Geräte zu bauen. Dazu sollte Java auf verschiedensten Plattformen eingesetzt und die Programme über das Netz verteilt werden können. Das Herunterladen eines solchen Programms und das anschliessende Ausführen sollte für den Klienten kein Sicherheitsrisiko darstellen. Diese Eigenschaften machten Java zu der Sprache für die Programmierung von verteilten Applikationen. Insbesondere das Konzept der Applets, Applikationen, die in HTML-Seiten eingebettet und über das Netz heruntergeladen werden können, hat viel zur Popularität von Java beigetragen.
Seit der Einführung hat sich Java ständig weiterentwickelt. Einerseits wurde die Sprache ergänzt (innere Klassen, anonyme Klassen,..). Andererseits sind die Bibliotheken, welche eng mit der Programmiersprache verwoben sind, immer mächtiger geworden. Die Standard-Ausgabe von Java ist längst nicht mehr auf kleinen Geräten ausführbar.
Schliesslich hat SUN zusätzliche, auf Java basierende Konzepte eingeführt, welche auf ein grosses Echo gestossen sind und die Stellung von Java zusätzlich gefestigt haben (Servlets, Enterprise Java Beans EJB, Java Intelligent Network Infrastructure JINI,..). Für die Entwicklung von verteilten Anwendungen (zum Beispiel Dienste im Internet) wird immer öfter auf Java gesetzt. Namhafte Forschungsprojekte, die sich mit Infrastruktur für verteilte Systemen und mobilem Code auseinandersetzen, benutzen Java als Ausgangspunkt für ihre Arbeit.

1.2 Oberon

Oberon wurde von N. Wirth und J. Gutknecht als Betriebssystem für Einzelplatzrechner entwickelt [34]. Das Oberon-System beinhaltet die Programmier-

sprache Oberon, mit welcher das System entwickelt wurde. Teil des Systems ist zudem eine automatische Speicherverwaltung. Sowohl das System als auch die Programmiersprache zeichnen sich durch grosse Kompaktheit und Einfachheit aus.
An der ETH-Zürich ist Oberon ein System für die Erforschung neuer Konzepte und Implementierungen. Der BlackBox-Komponenten-Builder ist ein kommerzielles Produkt, welches aus Oberon entstanden ist [3]. Ausgehend vom Oberon-System wurde auch eine kommerzielle Java VM für eingebettete Systeme (embedded Systems) gebaut [16].
Eine neue Entwicklung der ETH ist das Active-Oberon-System (Aos), welches es erlaubt, nebenläufige Programme auszuführen [24]. Neben den Laufzeitstrukturen des Betriebssystems wurde auch die Programmiersprache erweitert.

1.3 Eine Java VM für Aos

Im Zusammenhang mit Java wird vielerorts Forschung betrieben. Ein wichtiger Aspekt dieser Forschung ist die effiziente Implementierung der Java-Plattform. In dieser Arbeit sollte versucht werden, eine JVM, welche auf den Laufzeitstrukturen von Aos basiert, zu konstruieren.
Einerseits wäre es interessant, Java-Programme auf Aos laufen lassen zu können. Dadurch würde ein grosser Software-Pool erschlossen.
Andererseits wäre es interessant zu zeigen, dass das Betriebssystem direkt die wesentlichen Funktionen der JVM (oder einer beliebigen VM) übernehmen kann. Aos würde zu einem JavaOS!
Last but not least ist die Implementierung einer Java-Plattform ein Schritt zu einem grundlegenden Verständnis dieser Plattform.
Ein erster Anlauf eine solche JVM zu bauen, wurde in [2] unternommen. Die Arbeit an einer Oberon-JVM wurde von P. Reali weitergeführt.
In dieser Arbeit sollte diese JVM weiterentwickelt werden. Insbesondere sollten Java-Interfaces und die Behandlung von Ausnahmen (Exceptions) umgesetzt werden. Das bestehende System sollte auf Aos angepasst werden. Zudem sollte eine limitierte API zur Verfügung gestellt werden, mit welcher einfache Programme geschrieben werden können.

1.4 Ausgangspunkt

1.4.1 Plattform

Das Zielsystem für diese JVM war Aos. Aos ist auf Intel-PC's verfügbar und ist in der Lage, mehrere Prozessoren zu verwalten.
Als Entwicklungsumgebung diente einerseits Aos, beziehungsweise Native-Oberon unter Aos. Andererseits wurde auch Native-Oberon für Linux eingesetzt. Die Arbeit auf Linux ermöglichte einen schnellen Wechsel zwischen der Entwicklungsumgebung für die Java-API und der Entwicklungsumgebung für Active-Oberon.
Ausgangspunkt für die Java-API war die Implementierung von GNU-Classpath Version 0.001 [6]. Diese Library unterstützt die Funktionalitäten von Java 1.1 und implementiert einen grossen Teil der Funktionalität des JDK 1.2 (vergleiche

[17]). Für die Übersetzung der Class-Files wurde der Java-Compiler jikes von IBM eingesetzt (siehe [19]).

1.4.2 Vorhandenes Gerüst der JVM

Wie in 1.3 erwähnt, wurde bereits an einer JVM für Native-Oberon gearbeitet. Aus dieser Arbeit war ein Grundgerüst vorhanden, welches benutzt werden konnte. Folgende Funktionalitäten waren vorhanden.

Laden von Klassen

Class-Files konnten gelesen werden. Aus den Klassen konnten Native-Oberon-Typen und Module alloziert werden. Die Methoden wurden ebenfalls gelesen und ein grosser Teil der Instruktionen wurde in Assembler übersetzt. Die erzeugten Methoden wurden anschliessend gelinkt.
Diese Funktionalitäten waren bis zu diesem Zeitpunkt jedoch kaum getestet. Es war zu Beginn dieser Arbeit noch nicht möglich ein *echtes* Java-Programm auszuführen.

Einbindung in das Oberon-System

In einem separaten Modul waren Prozeduren vorhanden, welche der JVM die Laufzeitstrukturen des Native-Oberon-Systems zugänglich machten. Insbesondere wurden Methoden für das Erstellen und Verwalten eines Native-Oberon-Typdescriptors und Modulen zur Verfügung gestellt. Aber auch low-level Prozeduren für das Linken waren vorhanden.
Beim Anbinden an Aos zeigte sich, dass die Unterschiede zwischen Native-Oberon und Aos zwar nicht sehr zahlreich, aber dennoch einschneidend sind. Die Anpassung verlief nicht so reibungslos, wie es wünschbar gewesen wäre.

Werkzeuge

Für die Ausgabe der geladen Klassen sowie des erzeugten Codes waren Prozeduren vorhanden. Diese waren für die Fehlersuche von grosser Bedeutung. Zudem konnte aus einer Klasse ein Oberon-Module generiert werden (sogenannte *Stubs*). Diese Stubs konnten als Schnittstelle zu den Java-Klassen verwendet werden.

Kapitel 2

Laufzeitsysteme von Java und Aos

2.1 Einführung

In diesem Kapitel wird eine Übersicht über die *Laufzeitstrukturen* von Java und Aos gegeben. Unter Laufzeitstrukturen wird grundsätzlich alles verstanden, was das System zur Verfügung stellt, um das Funktionieren der Programme zu gewährleisten. Dazu gehört die Struktur der Speicherobjekte sowie die Art und Weise, wie Methoden aufgerufen werden. Aber auch Exception-Handling-Mechanismen und Unterstützung für nebenläufige Programme werden von einem Laufzeitsystem zur Verfügung gestellt.
Im Java-System sind nicht die Laufzeitstrukturen an sich spezifiziert, sondern die Funktionen, die von der Laufzeitumgebung realisiert werden sollen. Die JVM ist diese Laufzeitumgebung. Im Abschnitt 2.2 ist eine Übersicht über die JVM gegeben.
In Abschnitt 2.3 sind die Laufzeitstrukturen von Aos beschrieben.

2.2 Die Java Virtual Maschine (JVM)

Die JVM macht Javaprogramme unabhängig von einer bestimmten Prozessorarchitektur und einem bestimmten Betriebssystem.
Die JVM simuliert eine Maschine mit garantierten, einheitlichen Eigenschaften (Instruktionssatz, Wortgrösse, System-Calls,..). Eine JVM kann für verschiedene, reelle Maschinen gebaut werden und stellt den Programmen eine einheitliche Schnittstelle zu dieser realen Maschine zur Verfügung. Somit ist es möglich, Programme, welche für eine JVM geschrieben sind, auf jeder realen Maschine laufen zu lassen, für welche eine JVM existiert.
Die nachfolgende Beschreibung soll einen leicht verständlichen Überblick über die wichtigsten Eigenschaften der JVM geben. Es ist nicht das Ziel, eine JVM mit allen Details zu beschreiben. Für eine solche Beschreibung sei auf die Spezifikation der JVM [23] verwiesen. Eine formale Beschreibung liegt in [29] vor. In der Implementierungsbeschreibung (vergleiche 4) werden einzelne Aspekte der JVM ausführlich beschrieben.

Javaprogramme werden in eine plattformunabhängige Darstellung, sogenannte *Class-Files*, übersetzt. Diese Class-Files beinhalten die Informationen über die Klassen sowie die Übersetzung der Methoden in JVM-Instruktionen (*Bytecode*). Eine JVM, welche der Spezifikation genügt, muss in der Lage sein, anhand der Class-Files die entsprechenden Klassen zu laden (*loading*) und deren Methoden und Felder zugreifbar zu machen (*linking*). Anschliessend muss der Bytecode entsprechend der Spezifikation ausgeführt werden. In jedem dieser Vorgänge müssen bestimmte Tests durchgeführt werden, um die Integrität des Codes zu gewährleisten.
Die Spezifikation sagt nichts darüber aus, wie diese Aufgaben ausgeführt werden müssen. Eine JVM kann somit in irgend einer beliebigen Art und Weise implementiert werden, solange die obigen Anforderungen erfüllt werden.

2.2.1 Das Class-File Format

Ein Class-File enthält die Definition genau einer Klasse oder eines Interfaces. Zu dieser Definition gehört eine Symboltabelle (in [23] als Constant-Pool bezeichnet), mit deren Hilfe symbolische Verweise aufgelöst werden können (Namen, Konstanten Methoden, Klassen). Die Zugriffsflags für die Klasse oder das Interface (`abstract`, `final`, `public`, `interface`,...) sowie die Basis-Klasse und die Interfaces, welche implementiert werden, sind enthalten. Selbstverständlich müssen auch alle deklarierten Felder und Methoden im Class-File vorliegen. Der eigentliche Code der Methoden ist dabei als Attribut einer Methoden-Beschreibung festgehalten.
Das nachfolgende Beispiel soll zeigen, wie die statische Struktur einer Java-Klasse in einem Class-File dargestellt wird.

```
public class Point{
  private int x;
  private int y;
  public static int distance( Point a, Point b ){ .. }
  ..
}
```

In Abbildung 2.1 ist die Struktur der Klasse `Point` als UML-Objektdiagramm aufgezeichnet. Einige Attribute wurden der Einfachheit halber weggelassen.
Die Klasse `Point` ist `public` zugreifbar. Die Klasse wird über eine Struktur `ClassInfo` in der Symbol-Tabelle beschrieben. Der Name der Klasse ist ein eigener Eintrag in der Symbol-Tabelle. Die Struktur `ClassInfo` enthält einen Zeiger auf diesen Eintrag. Analog zur Klasse `Point` ist die Basis-Klasse beschrieben. Interfaces werden keine implementiert. Die Klasse verfügt über zwei `private` Felder. Diese werden mit einer `FieldInfo` Struktur beschrieben. Eine `FieldInfo` Struktur enthält die Zugriffsflags für ein Feld, sowie zwei Verweise in die Symbol-Tabelle. Der eine Verweis zeigt auf einen Eintrag mit dem Namen des Feldes. Der andere Eintrag enthält eine Zeichenkette, welche den Datentyp des Feldes beschreibt (sogenannter *Field-Descriptor*). Zusätzlich kann ein Feld Attribute enthalten. Ein mögliches Attribut wäre ein konstanter Wert dieses Feldes. In einem solchen Attribut wäre wiederum ein Verweis in die Symbol-Tabelle vorhanden, um auf den tatsächlichen Wert zuzugreifen.
Sehr ähnlich zur `FieldInfo` Struktur ist die `MethodInfo` Struktur zur Beschreibung von Methoden. Jedoch unterscheidet sich der *Method-Descriptor* vom

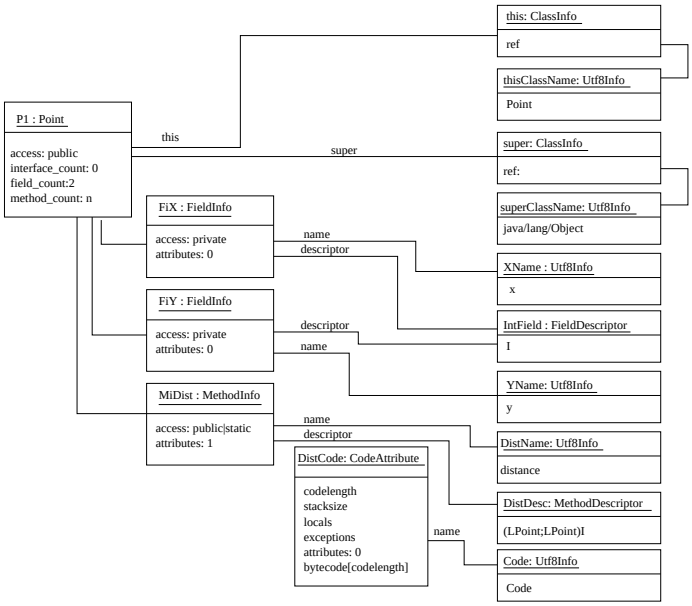

Abbildung 2.1: Vereinfachtes UML-Objektdiagramm eines Class Files

Field-Descriptor. Im Method-Descriptor werden sämtliche Parameter und der Rückgabewert einer Methode als Zeichenkette codiert. Die Aktionen der Methode sind in einem `CodeAttribute` festgehalten. Dieses Attribut enthält neben dem Bytecode auch Informationen über die maximale Grösse des Operandenstacks, die Anzahl lokaler Variablen sowie eine Tabelle für das Exception-Handling.
Die Exceptions, welche in einer Methode auftreten können, sind in einem speziellen `ExceptionAttribute` aufgeführt.

Verifikation von Class-Files

Die Kompilation eines Javaprogramms zu Class-Files und das Ausführen solcher Class-Files in einer JVM sind zwei völlig unabhängige Prozesse. Eine JVM kann nicht davon ausgehen, dass bei der Kompilation die Korrektheit des Programms geprüft wurde und folglich keine Fehler in einem Class-File vorhanden sind. Beispielsweise können Class-Files über das Netz geladen werden. Bei dieser Übertragung könnten die Dateien verändert werden, so dass das Class-File fehlerhaft wird. Es ist auch möglich, dass in einem Programm Abhängigkeiten existieren, die nicht auf jedem Zielrechner aufgelöst werden können.
Damit das Ausführen eines Class-Files kein Sicherheitsrisiko darstellt, muss die JVM eine solche Datei gewissen Tests unterziehen. Dieser Vorgang wird als *Verifikation* bezeichnet. In der Spezifikation ist die Verifikation in vier Schritte aufgeteilt. Die einzelnen Schritte müssen aber nicht strikt voneinander getrennt sein.

Erster Schritt: In einem ersten Schritt wird lediglich geprüft, ob das Class-File *wohlgeformt* ist. Es wird untersucht, ob alle Attribute vorhanden sind und

ob sie die richtige Länge haben.

Zweiter Schritt: Im zweiten Schritt muss sichergestellt werden, dass `final` Klassen nicht erweitert und `final` Methoden nicht überschrieben werden. Alle Klassen ausser `Object` müssen eine direkte Basis-Klasse haben. Die Einträge in der Symboltabelle (Constant-Pool) müssen wohlgeformt sein. Die Syntax für Descriptoren muss korrekt sein, alle Attribute eines Eintrags müssen vorhanden sein und die Indizes in die Symbol-Tabelle müssen auf einen gültigen Wert mit dem richtigen Typ zeigen.

Dritter Schritt: Bis zu diesem Punkt wurde die statische Struktur (*static Constraints*) eines Class-Files verifiziert. In diesem Schritt wird das dynamische Verhalten geprüft. Dazu muss der Byte-Code analysiert werden und Datenflussanalysen werden durchgeführt. Dieser Schritt wird als *Bytecode-Verifikation* bezeichnet. Die Bytecode-Verifikation muss sicherstellen dass:

- An einer bestimmten Stelle im Programm der Operanden-Stack immer dieselbe Größe hat und die Werte auf dem Stack immer dieselben Typen aufweisen. Das heisst, wenn es auf verschiedene Arten möglich ist, an eine bestimmte Stelle im Code zu gelangen, so darf sich das nicht auf die Grösse des Operanden-Stacks auswirken. Für das `finally` Konstrukt muss diese Forderung abgeschwächt werden.
- Keine lokale Variable benutzt wird, bevor deren Wert und Typ bekannt ist.
- Alle Prozeduren mit den passenden Argumenten aufgerufen werden.
- Nur typkompatible Zuweisungen vorgenommen werden.
- Alle Instruktionen die richtige Anzahl von Argumenten im Operanden-Stack oder in den lokalen Variablen vorfinden. Zudem müssen die Argumente den richtigen Typ haben.

Aus dieser Liste von durchgeführten Tests kann unschwer entnommen werden, dass die Bytecode-Verifikation der aufwändigste Schritt der gesamten Verifikation darstellt.

Vierter Schritt: Im letzten Schritt werden symbolische Referenzen zu Feldern, Methoden und Klassen, beziehungsweise Interfaces geprüft. Die Struktur einer solchen Referenz ist in Abbildung 2.2 dargestellt.

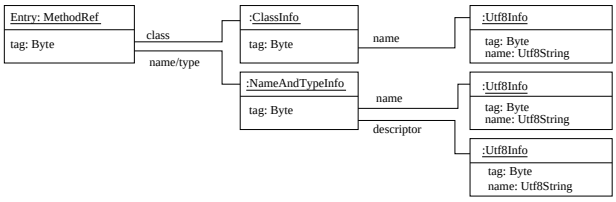

Abbildung 2.2: UML-Objektdiagramm einer symbolischen Referenz auf eine Methode

Für die Prüfung der symbolischen Referenz wird das entsprechende Objekt (Klasse, Methode, Feld,..) gesucht. Anschliessend muss geprüft werden, ob das gefundene Objekt einen passenden Descriptor aufweist und der Zugriff auf dieses Objekt erlaubt ist.

2.2.2 Der JVM-Instruktionssatz

Eine JVM-Instruktion besteht aus einem ein Byte grossen Code, welcher die auszuführende Operation definiert (*Opcode*). Diesem Opcode folgen keine, ein oder mehrere Operanden. Viele Instruktionen holen ihre Operanden von einem Stack (operand stack) und schreiben das Resultat dorthin zurück.
Die Grösse des Opcodes von einem Byte beschränkt die Grösse des Instruktionssatzes.
Im Folgenden werden die verschiedenen Kategorien von Instruktionen beschrieben. Für jede Kategorie wird ein Beispiel einer Instruktion gegeben. Die Beispiele sind in der Form `Instruction<Parameter1>...<ParameterN>` Aufgeschrieben. Dabei ist `Instruktion` der Opcode der Instruktion (ein Byte im Bytecode) und `Parameter1 ...ParameterN` sind Parameter, die vom Bytecode gelesen werden. Die Instruktionen verwenden häufig auch Parameter vom Stack. Die Semantik einer Instruktion wird in einem Pseudocode erklärt. Im Pseudocode werden einige Funktionen und Variablen verwendet, welche in Tabelle 2.1 erklärt sind.

Aus diesen Beispielen wird ersichtlich, dass einige Instruktionen komplizierte

Funktion oder Variable	Bedeutung
`Push4`	Ein vier Byte grosser Wert soll auf den Stack geschrieben werden.
`Pop4`	Ein vier Byte grosser Wert soll vom Stack entfernt werden.
`Locals[i]`	Variable an der Position `<index>` in der Tabelle der lokalen Variablen.
`ConstantPool[i]`	Information-Rekord an der Stelle `<i>` im Constant-Pool.
`InitObject(ClassRef)`	Anhand einer Referenz auf eine Klasse muss eine Instanz erzeugt werden.
`GetField(ObjectRef,FieldRef)`	Anhand einer Referenz auf eine Objekt-Instanz und einer Referenz auf ein Feld muss der Wert dieses Feldes ermittelt werden.
`pc`	*Program-Counter*; aktuelle Position im Programm.
`GetMethod(MethodRef)`	Anhand einer Referenz auf eine Methode wird die Adresse dieser Methode gefunden.
`GetParamCount(MethodRef)`	Anhand einer Referenz auf eine Methode wird die Anzahl Parameter bestimmt.
`FindHandler(pc, Exception)`	Anhand des Program-Counters und einer Exception wird ein passender Exception-Handler gefunden.

Tabelle 2.1: Pseudocode-Funktionen

Abläufe darstellen.

Load/Store-Instruktionen

Für jede Funktion wird ein sogenanntes *Frame* aufgebaut. In einem Frame ist eine Tabelle mit den lokalen Variablen enthalten. Die Parameter einer Funktion werden in diesem Zusammenhang ebenfalls als lokale Variable bezeichnet.
Die Aufgabe der Load/Store-Instruktionen ist es, lokale Variable auf den Stack zu schieben (push) sowie Werte vom Stack in die lokalen Variablen zu speichern und vom Stack zu entfernen (pop). Load/Store-Instruktionen sind *typisiert*, das heisst, im Opcode der Instruktion wird der Datentyp des Operanden festgelegt. Für den Zugriff auf die ersten vier lokalen Variablen sind spezielle Instruktionen vorhanden, die kein zusätzliches Argument `<index>` benötigen.
Beispiel: Mit `LLoad` und `LStore` werden Long-Variablen geladen und gespeichert.

```
LLoad <index>    : Push4( Mem[index] );       (*high*)
                   Push4( Mem[index+1] );     (*low *)
LStore <index>   : Locals[index+1]:= Pop4();  (*low *)
                   Locals[index]:= Pop4()     (*high*)
```

Arithmetische Instruktionen

Für arithmetische Operationen stehen viele Instruktionen zur Verfügung (Addition, Subtraktion, Multiplikation, Division, Bitoperationen, Vergleichsoperationen,..).
Typischerweise werden die Argumente für die Instruktionen vom Operandenstack geholt und das Resultat der Operation wieder auf den Stack geschrieben. Es wird zwischen Integer-Arithmetik und Floatingpoint-Arithmetik unterschieden. Die Argumenttypen sind ebenfalls im Opcode der Instruktionen codiert. Für die Datentypen `byte, short, char, boolean` werden die Instruktionen für den Typ `Integer` verwendet!
Beispiel: `LSub` subtrahiert zwei Werten des Typs Long.

```
LSub : value1 := Pop4();
       value1 := value1|(Pop4()<<32);
       value2 := Pop4();
       value2 := value2|(Pop4()<<32);
       result := value1 - value2;
       Push4((result>>32)&0xFFFF);
       Push4(result&0xFFFF);
```

Typumwandlungen

Die Instruktionen für die Typumwandlung erlauben eine Umwandlung zwischen den verschiedenen numerischen Typen. Solche Umwandlungen können den Wertebereich einer Variablen erweitern (*widening conversion*) oder verkleinern (*narrowing conversion*). Im ersten Fall kann es zu Verlust von Genauigkeit kommen (Umwandlung einer Integervariable in eine Floatvariable). Im zweiten Fall kann die Information über die Grösse der Zahl verloren gehen (Umwandlung einer Longvariablen in eine Integervariable).
Beispiel: Ein `int`-Werte wird in einen `short`-Wert umgewandelt.

```
i2s : value := Pop4();
      Push4(value&0xFF);
```

Instruktionen für die Erzeugung und Manipulation von Objekten

Der Instruktionssatz der JVM enthält Instruktionen für die Erzeugung eines neuen Objektes und von Arrays (eindimensionale und mehrdimensionale Arrays). Ferner wird der Zugriff auf die Felder von Objekten durch spezielle Instruktionen realisiert. Obwohl sowohl Klasseninstanzen als auch Arrays Referenztypen darstellen, werden separate Instruktionen für den Umgang mit diesen Objekten verwendet. Wichtige Instruktionen, welche ebenfalls in diese Kategorie fallen, sind die Instruktionen für den Typtest einer Instanz (`Checkcast`, `Instanceof`).
Beispiel: Eine neue Objektinstanz wird erzeugt.

```
new <high><low>  : index := (high<<8)|low;
                   ClassRef := ConstantPool[index];
                   ObjectRef := InitObject(ClassRef);
                   Push4(ObjectRef);
```

Beispiel: Es wird auf ein Klassenfeld der Grösse vier Byte zugegriffen.

```
getfield<high><low>  : index := (high<<8)|low;
                       FieldRef := ConstantPool[index];
                       ObjectRef := Pop4();
                       value := GetField( ObjectRef, FieldRef );
                       Push4( value );
```

Stackmanipulation

In gewissen Fällen ist es nötig, Elemente auf dem Operandenstack zu duplizieren oder zu vertauschen. Dazu stehen spezielle Instruktionen zur Verfügung.
Beispiel: Der oberste Wert auf dem Stack wird dupliziert.

```
Dup :  value := Pop4();
       Push4(value);
       Push4(value);
```

Verzweigungen im Programmablauf

Verzweigungen werden durch Sprung-Instruktionen realisiert. Solche Sprünge können bedingt oder unbedingt sein. Die Sprung-Instruktionen sind teilweise typisiert.
Beispiel: Es wird bedingt an eine spezifizierte Adresse gesprungen.

```
Ifequal <high><low>  : value := Pop4();
                       IF value = 0 THEN pc := (high<<8)|low
                       ELSE pc := pc+3;
```

Beispiel: Es wird unbedingt an eine spezifizierte Adresse gesprungen.

```
Jsr <high><low>      : returnAddress := pc+3;
                       Push4(returnAddress);
                       pc := (high<<8)|low;
```

Aufrufe von Methoden werden nicht mit der Instruktion `Jsr` (Jump to Subroutine) bewerkstelligt. Zu diesem Zweck stehen eigene Instruktionen zur Verfügung. Es gibt drei verschiedene Methodenaufrufe (`Invokestatic, Invokespecial, Invokevirtual`), je nachdem wie eine Methode deklariert ist. Mit einer `Return`-Instruktion wird aus einer Methode zurückgesprungen und das Resultat auf den Operandenstack derjenigen Methode geschrieben, welche diesen Aufruf gemacht hat (*Caller*). Die `Return` Instruktionen sind typisiert.
Beispiel: Eine private Instanzmethode wird aufgerufen und ein `int`-Wert wird zurückgegeben.

```
InvokeSpecial <high><low>  : index := (high<<8)|low;
                             MethodRef := ConstPool[index];
                             ObjectRef := Caller.Pop4();
                             MethodEntry:= GetMethod(
                                           MethodRef);
                             ParameterCnt := GetParamCount(
                                           MethodRef)
                             Callee.Locals[0]:= ObjectRef;
                             i := 1;
                             WHILE i < ParameterCnt DO
                               Callee.Locals[i] := Caller.Pop4()
                             END;
                             Caller.nextPc := pc+3;
                             pc := MethodEntry;

Ireturn                    : value := Callee.Pop4();
                             Caller.Push4( value );
                             pc := Caller.nextPc;
```

In diesem Beispiel wird deutlich, dass in einer einzigen Instruktion eine komplexe Prozedur beinhaltet sein kann.
Die Methoden werden immer nach demselben Muster aufgerufen. Die genaue Semantik der Funktionen `GetMethod()` ist unterschiedlich.
Beim Auftreten von Ausnahmen (*Exceptions*) wird der normale Programmfluss ebenfalls unterbrochen und an einer anderen Stelle fortgefahren. Mit der Instruktion `Athrow` wird anhand eines Exception-Objektes und des Program-Counters die entsprechende Routine für die Behandlung der Ausnahme gesucht.
Beispiel: Eine Exception wird geworfen.

```
Athrow : Exception := Pop4();
         Handler := FindHandler(pc, Exception);
         Push4(Exception);
         pc := Handler;
```

Für die Behandlung von `finally` wird das Instruktionspaar `Jsr, Ret` verwendet.

2.2.3 Laden, Linken, Initialisieren

Für das Laden, Linken und Initialisieren steht eine spezielle Klasse zur Verfügung, der sogenannte *ClassLoader*. Beim Starten erzeugt die JVM eine erste Klasse

mit dem standard ClassLoader. Die geladene Klasse wird initialisiert und die Methode `main()` wird aufgerufen. Alle nachfolgenden Aktionen werden von dieser Methode initiiert. Alle weiteren Klassen werden dynamisch geladen.
Für den Ladevorgang kann entweder der JVM-ClassLoader oder ein benutzerdefinierter ClassLoader verwendet werden. Der gesamte Ladevorgang besteht aus dem Laden (*Loading*), Linken (*Linking*) und Initialisieren (*Initialisation*).

Loading: Unabhängig vom verwendeten ClassLoader muss zuerst die entsprechende Datei mit der Klassendefinition gefunden werden. Anschliessend wird das Class-File gelesen. Falls die zu ladende Klasse eine direkte Basisklasse besitzt, wird die symbolische Referenz zu dieser Klasse aufgelöst und die Basisklasse wird rekursiv geladen. Analog wird mit den implementierten Interfaces verfahren. Nachdem der Ladevorgang der Basisklassen und Interfaces abgeschlossen ist, wird die Klasse als geladen markiert und der verwendete ClassLoader wird vermerkt.

Linking: Nach dem Laden wird eine Klasse gelinkt. Dieser Vorgang beinhaltet die Überprüfung der static Constraints wie in 2.2.1 beschrieben. Anschliessend werden die statischen Felder alloziert und initialisiert (*Preparation*). Die Preparation wird gefolgt von der sogenannten *Resolution*, dem Auflösen der symbolischen Referenzen auf Felder, Methoden, Klassen und Interfaces. Die Struktur einer solchen Referenz ist in Abbildung 2.2 dargestellt. Das Überprüfen einer solchen symbolischen Referenz wurde als Teil der Verifikation (2.2.1) beschrieben. Bei der Resolution muss diese Referenz durch einen konkreten, implementatierungsabhängigen Wert ersetzt werden (Beispielsweise eine Speicheradresse). Zu einer Klasse können auch Methoden, welche in einer kompilierten Programmiersprache geschrieben wurden, hinzugebunden werden (*Binding*). Das kann auf unterschiedliche Art und Weise gemacht werden und die JVM-Spezifikation gibt keine konkreten Richtlinien vor.
Allerdings existiert eine standardisierte Schnittstelle für solche *native Methoden*. Diese ist in [20] beschrieben.

Initialisation: Der gesamte Ladevorgang wird mit dem Initialisieren einer Klasse abgeschlossen. Dabei wird die Prozedur `clinit()` (*static Initializer*) ausgeführt.

2.2.4 JVM und Java-API

Bis jetzt wurde beschrieben, was die JVM können muss, um Class-Files zu lesen und auszuführen. Zu diesen Eigenschaften müssen noch einige grundsätzliche Dienste für die Implementierung der Java-API zur Verfügung gestellt werden.
Als erstes sei auf die automatische Speicherverwaltung verwiesen. Das System räumt nicht mehr benutzte Objekte selbständig ab (Garbage-Collection).
Ferner beinhaltet die Java-Plattform mächtige Bibliotheken. Einige Klassen dieser Bibliotheken müssen als Teil der Sprache betrachtet werden. Die Klasse `Object` ist die Basisklasse aller möglichen Klassen. Alle Exception-Klassen sind Spezialisierungen der Klasse `Throwable`. Die verschiedenen Objekte müssen als *Monitore* verwendet werden können, welche in verschiedenen `Threads` benutzt

werden. Die Java-API erlaubt die Eigenschaften der Typen abzufragen (*Reflection*). Dazu muss auf die JVM-interne Struktur von Objekten zugegriffen werden.
Eigentlich ist die JVM eine von der Sprache Java unabhängige Einheit. Diese Beispiele verdeutlichen, dass aber eine enge Verflechtung vorhanden ist. Eine JVM muss die Java-Basistypen zur Verfügung stellen. Klassen `Object, Exception, Thread` sind ebenfalls unabdingbar. Die Schnittstellen der Java-API Version 1.2 sind in [17] beschrieben. Mittlerweile liegt bereits die Version 1.3 vor [18].

2.3 Oberon Laufzeitstrukturen

Die Aos-JVM sollte die Laufzeitstrukturen von Aos ausnutzen. Es sollte gezeigt werden, dass ein Betriebssystem mit geeigneten Schnittstellen direkt die wesentlichen Aufgaben einer VM übernehmen kann.
In diesem Abschnitt sollen Laufzeitstrukturen von Aos erklärt werden, soweit sie für diese Arbeit von Bedeutung sind. Eine umfassende Dokumentation des Aos-Systems würde den Rahmen diese Berichtes bei weitem sprengen.
Die Beschreibungen beziehen sich auf Aos! In Native-Oberon sind die Laufzeitstrukturen ähnlich, aber nicht genau gleich!

2.3.1 Die Speicherorganisation von Aos

Damit die JVM die Laufzeitstrukturen von Aos benutzen kann (insbesondere der Garbage-Collector), müssen Java-Objekte gleich wie Aos-Objekte dargestellt werden. Im Folgenden wird die Speicherorganisation von Aos kurz erklärt.

In Aos werden zwei Speicher-Regionen unterschieden, der Stack und der Heap. Auf dem Stack werden die lokalen Variablen und Parameter von Prozeduren alloziert. Alle Objekte liegen im Stack direkt nebeneinander.
Dynamische Datenstrukturen werden auf dem Heap alloziert. Dabei wird ein Speicherblock mit der entsprechenden Grösse reserviert. Wie in Abschnitt 1.2 erwähnt, verfügt Aos über eine automatische Speicherverwaltung. Jede Speicherallokation auf dem Heap wird vom System registriert. Speicherobjekte, welche nicht mehr *benötigt* werden, werden vom System wieder abgeräumt (Garbage-Collection). Damit das funktionieren kann, müssen zur Laufzeit die Strukturen der Speicherobjekte bekannt sein. Oberon kennt drei verschiedene Typen von Heap-Speicherobjekten.

- **System-Blöcke:**
  Diese stellen Speicherblöcke ohne zusätzliche Information dar. Sie werden verwendet für die Allokation von `ARRAY`'s von Basistypen `LONGINT, INTEGER, CHAR, ...` Das Laufzeitsystem merkt sich den Beginn und das Ende des Speicherblocks.

- **RECORD-Blöcke:**
  Diese werden für die Allokation von `RECORD`s verwendet. Zu jedem Rekord-Typ muss zusätzlich eine Beschreibung dieses Datentyps (*Typdescriptor*) vorhanden sein. Mit Hilfe dieses Typdescriptors kann der Garbage-Collector alle zugreifbaren Speicherobjekte markieren.

In Aos wurden zusätzlich sogenannte *Protected-Objects* definiert. Diese können von einem Thread gelockt werden. Ein Protected-Object wird wie ein "normaler" Rekord alloziert. Jedoch besitzt jedes solche Objekt fünf vordefinierte Zeiger [1]. Zudem verfügt jedes Protected-Object über einen Zähler, mit welchem festgehalten werden kann, wie häufig ein Objekt gelockt wurde (*Lockcount*).

- **ARRAY-Blöcke:**
  Diese werden für die Allokation von `ARRAY`s verwendet. In einem Array-Block muss die Dimension des oder der Arrays vermerkt sein. Zusätzlich muss ein Zeiger auf den Typdescriptor der Arrayelement vorhanden sein.

Der Typdescriptor eines `RECORD`'s beinhaltet die Grösse des Objektes, die Position der Zeiger innerhalb dieses `RECORD`'s, eine Tabelle mit den Basistypen und eine Methodentabelle. Anhand der Tabelle mit den Basistypen kann ein Typtest vorgenommen werden. Das Überschreiben von typgebundenen Methoden wird mit dieser Methodentabelle realisiert. Wenn ein spezialisierter Typ eine Methode überschreibt, wird die Adresse seiner eigenen Methode im Typdescriptor gespeichert. Im Typdescriptor des Basistyps zeigt derselbe Eintrag auf die Einsprungadresse der Methode, welche vom Basistypen implementiert wurde.
Anhand eines konkreten Beispiels soll der Speicherlayout veranschaulicht werden.

```
TYPE
     ObjectDesc = RECORD END;

     Node = POINTER TO NodeDesc;
     NodeDesc = RECORD(ObjectDesc)
                left, right : Node;
                key : LONGINT
     END;
     NodeSet = POINTER TO ARRAY 4 OF Node;
```

Wie diese Typen bei einer Allokation im Speicher liegen, ist in Abbildung 2.3 dargestellt.

2.3.2 Aufruf von Oberon-Prozeduren (*Calling-Conventions*)

In 2.3.1 wurde erklärt, wie Oberon-Objekte im Speicher liegen. In diesem Abschnitt wird erklärt, wie Prozeduren im Oberon-System aufgerufen werden und wie Parameter übergeben werden.
Grundsätzlich werden die Parameter für Prozeduren von links nach rechts auf den Stack geschrieben. Jede Adresse eines Parameters muss ganzzahlig durch vier teilbar sein. Bei einem Referenzparameter wird die Adresse des Objektes auf den Stack geschrieben. Bei Werteparametern wird das vollständige Objekt auf den Stack geschoben. Eine Funktion wird durch die Instruktion `call addr` aufgerufen. Durch diese Instruktion wird die Adresse der nächsten Instruktion

[1] Je ein Zeiger auf den Thread, der das Objekt gerade gelockt hat, auf den Beginn und das Ende der Liste mit den wartenden Threads und auf den Beginn und das Ende der Bedingungen um einen wartenden Thread zu reaktivieren

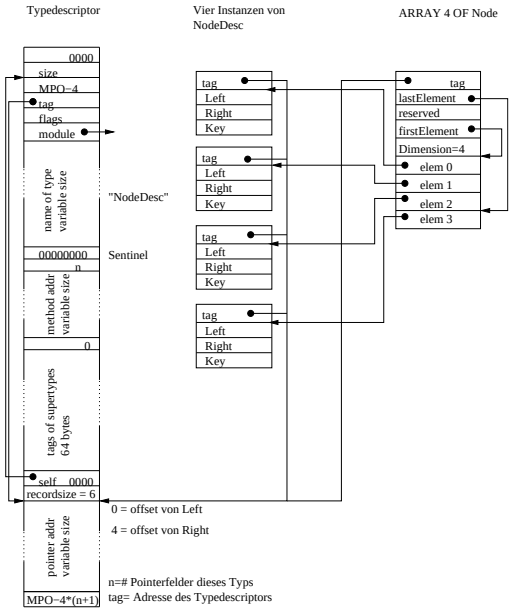

Abbildung 2.3: Darstellung des Speicherlayouts von `NodeSet`.

in dieser Prozedur auf den Stack geschoben und der Programm-Counter auf den Beginn der auszuführenden Prozedur gesetzt. Die Prozedur wird mit der Instruktionsfolge

```
push ebp          //save old frame-pointer
mov ebp, esp      //make stack pointer new frame pointer
```

eingeleitet (*Prolog*) und mit der Instruktionsfolge

```
mov esp, ebp      //remove local variables
pop esp           //restore old frame-pointer
ret sizeOfParams  //pop retAddr
                  //remove parameters from stack
                  //continue execution at retAddr
```

beendet (*Epilog*). Die Variablen und Parameter werden relative zum Frame-Pointer (`ebp`) adressiert. Der letzte Parameter kann mit der Adressierung `8[ebp]` angesprochen werden. Die erste lokale Variable ist an der Adresse `-4[ebp]`. Rückgabewerte von Prozeduren werden in Register gespeichert und können dort vom Caller abgeholt werden.
Die Tabelle 2.2 beschreibt, in welche Register die Rückgabewerte geschrieben werden.

Im folgenden Beispiel soll die Calling-Convention veranschaulicht werden.

```
PROCEDURE Foo( x, y, z : LONGINT ):LONGINT
VAR a,b,c : LONGINT;
BEGIN
  ..
END Foo;
```

Datentyp	Grösse in Bytes	Register
POINTER	4	EAX
CHAR, SHORTINT	1	AL
INTEGER	2	AX
LONGINT, SET	4	EAX
HUGEINT	8	EAX (low) EDX (high)
REAL	4	ST(0)
LONGREAL	8	ST(0)

Tabelle 2.2: Übergabe von RETURN-Werten bei Oberon

In 2.4 ist der Zustand auf dem Stack dargestellt, wie er sich in den verschiedenen Phasen präsentiert. Bei typgebundenen Prozeduren wird das Object, auf welches sich die Operation bezieht (`Self`), implizit als letzter Parameter übergeben. Demzufolge haben die beiden Aufrufe

```
Object.doIt(p1, p2, p3);
DoIt( p1, p2, p3, TypeDesc, ObjectRef );
```

dieselben Parameter auf dem Stack.

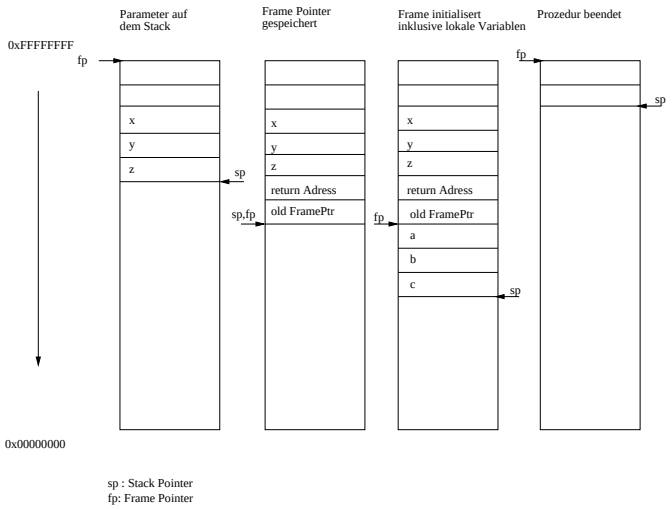

Abbildung 2.4: Darstellung des Stackzustandes während einem Aufruf von `Foo`.

2.3.3 Module

In Aos ist ein Modul eine abgeschlossene Programm-Einheit. Mit Hilfe des Modulkonzeptes wird ein System strukturiert. Aos lädt ein Modul bei Bedarf und linkt es mit den bereits geladenen Modulen. Auf diese Weise wird sichergestellt, dass dieselben Module nicht mehrfach in den Speicher geladen werden.
Ein geladenes Modul wird in Aos durch den Datentyp `ModuleDesc` beschrieben.

In einer Modulinstanz werden die Einsprungadressen von Prozeduren, die Typdescriptoren der deklarierten Typen, der Maschinen-Code und globale Daten festgehalten. Beim Laden eines Moduls wird eine Objekt-Datei gelesen und eine Modul-Instanz angelegt.
Im Feld `Modul.refs` sind ausserdem die Namen, Adressen, Parameter und Variablen für jede Prozedur enthalten (*Metadaten*). Diese Metadaten können beispielsweise für die Behandlung von Ausnahmen verwendet werden.
Der Garbage-Collector braucht die geladenen Module und die Stack-Frames als Startpunkt für die Markierung der erreichbaren Speicherobjekte.

2.3.4 Traps

Werden vom Prozessor während der Programmausführung anormale Ereignisse festgestellt (Beispielsweise eine Division durch 0, eine Referenzierung eines undefinierten Speicherobjektes,...), so wird vom Prozessor ein Interrupt ausgelöst. Innerhalb eines Programmes kann explizit mit den Prozeduren `HALT(nr)` oder `ASSERT(condition)` ein Software-Interrupt ausgelöst werden.
Im Aos werden diese Interrupts als sogenannte *Traps* behandelt. Eine Liste dieser Traps und ihrer Bedeutung kann aus [32] entnommen werden. Im System kann für jeden Thread eine Prozedur definiert werden, welche diese Traps behandelt. Als Parameter erhält diese Trapbehandlungs-Routine den aktuellen Zustand des Prozessors, auf welchem der Thread läuft (`AosInterrupts.State`) und den Interrupt-Status (`AosInterrupts.ExceptionState`). Zudem wird ein Flag mitgegeben, mit welchem angezeigt werden kann, ob die Trapbehandlung erfolgreich durchgeführt werden konnte. Der Typ dieser Prozedur ist so definiert:

```
TYPE State = RECORD (* Prozessor Zustand *)
               ..   (* alle Register *)
               EBP, ESP, EIP, .. :LONGINT;
             END;
TYPE ExceptionHandler =
     PROCEDURE( VAR int    :AosInterrupts.State;
                VAR exc    :AosInterrupts.ExceptionState;
                VAR resume :BOOLEAN);
```

Im Typ State sind alle Registerwerte zugreifbar. Für die Behandlung des Traps in einem Traphandler sind `EBP`, `ESP` und `EIP` die wichtigsten Register. Dabei ist EBP der Frame-Pointer, ESP der Stack-Pointer und EIP ist der Programm-Counter der Stelle, an welcher die Ausnahme aufgetreten ist.
Falls kein Exception-Handler installiert wurde oder das Flag `resume` den Wert `FALSE` hat, wird ein Stackdump ausgegeben und der Thread wird entweder abgebrochen oder neu gestartet. Für die Ausgabe des Stackdumps werden die Daten auf dem Stack mit Hilfe der Metadaten der Module ausgewertet.
Falls `resume` nach dem Aufruf des Exception-Handlers den Wert `TRUE` hat, so wird die Ausführung des Programmes an der Adresse in `State.EIP` fortgesetzt. Es ist also für einen Exception-Handler möglich, den Programm-Ablauf bei einer Ausnahme zu steuern!

2.3.5 Threads

Aos ermöglicht es, nebenläufige Aktivitäten zu definieren und ausführen zu lassen. Für nebenläufige Prozesse wurde die Abstraktion des *Active Object*'s eingeführt. Ein Active Object läuft in einem eigenen Thread. Ein Thread besitzt einen eigenen Stack. Während der Thread aktiv ist, monopolisiert er einen Prozessor. Der Heap wird von allen Threads gleichzeitig benutzt. Das Betriebssystem verwaltet die Threads und lässt sie auf den Prozessoren der Maschine laufen.
Daten, welche von mehreren Threads gleichzeitig verwendet werden (auf dem Heap), können als *Protected Object* definiert werden. Protected Object's sind Monitore im Sinne von [15]. Mit diesen Monitoren können die Threads synchronisiert werden.
In der Sprache *Active Oberon* wurden neue Sprachelemente eingeführt, um aktive Objekte und Monitore zu implementieren und um die Active Object's miteinander zu synchronisieren. In Figur 2.5 ist gezeigt, wie das Schema des Producer Consumers (vergleiche [1]) in Active-Oberon definiert werden kann.
In diesem Beispiel werden die neuen Sprachkonstrukte {`EXCLUSIVE`}, `AWAIT()` und {`ACTIVE`} benutzt. `OBJECT` ist eine Aliasbezeichnung für `RECORD TO OBJECT`. Es werden spezielle Initialisierungsprozeduren verwendet. Diese werden durch ein vorangestelltes `&` gekennzeichnet.
{`EXCLUSIVE`} markiert eine *Critical Section* und garantiert den gegenseitigen Ausschluss von verschiedenen Active Objects. `AWAIT()` hält ein Active Object an, bis die entsprechende Bedingung erfüllt ist. Mit {`ACTIVE`} wird die Aktivität des Active Object's (Thread) definiert.
Der Compiler übersetzt diese Sprachkonstrukte in System-Aufrufe.
Mit `CreateActivity(..)` kann ein Thread erzeugt werden ({`ACTIVE`}). Mit einem Aufruf `Passivate(..)` wird der aktuelle Thread in einen Wartezustand versetzt (`AWAIT()`), bis die Bedingung, welche in `Condition()` spezifiziert wurde, erfüllt ist. Dabei müssen die vom Thread gesperrten Objekte freigegeben werden, damit andere Threads weiterarbeiten können. Mit `lock( p )` wird ein Objekt vom laufenden Thread gesperrt, und mit `Unlock(p)` wieder frei gegeben ({`EXCLUSIVE`}).
Ein bereits gelocktes Objekt kann nicht ein zweites Mal vom selben Thread gelockt werden. Bei einer Exception (Trap) werden die Locks auf die Objekte nicht freigegeben.

Abbildung 2.5: Producer Consumer implemented with active objects

```
MODULE ProducerConsumer;

TYPE   Buffer* = OBJECT
           data : ARRAY BufSize OF LONGINT;
           in, out : LONGINT;

           PROCEDURE put( i : LONGINT );
           BEGIN{EXCLUSIVE}
             AWAIT( (in+1) MOD BufSize # out );
             data[in] := i;
             in := in + 1 MOD BufSize
           END put;

           PROCEDURE get(): LONGINT;
           VAR i : LONGINT;
           BEGIN{EXCLUSIVE}
             AWAIT( out # in ); i := out;
             out := out+1 MOD BufSize;
             RETURN i
           END get;
         END Buffer;

       Producer* = OBJECT
         buffer : Buffer; item : LONGINT;

         PROCEDURE &init( b : BUFFER );
         BEGIN buffer := b
         END init;

       BEGIN{ACTIVE}
         WHILE TRUE DO b.put( item ); INC( item ) END
       END Producer;

       Consumer* = OBJECT
         buffer : Buffer; item : LONGINT;

         PROCEDURE &init( b : Buffer );
         BEGIN  buffer := b
         END init;

         BEGIN{ACTIVE}
           WHILE TRUE DO item := b.get()END
         END Consumer;

   PROCEDURE Run*;
   VAR b : Buffer; p : Producer; c : Conusumer;
   BEGIN
     NEW( b ); NEW( p, b ); NEW( c, b );
   END Run;

END ProducerConsumer.
```

2.4 Gegenüberstellung von Java und Aos

An sich verfügen die zwei Systeme über manche Gemeinsamkeiten. Beide Systeme besitzen einen Garbage-Collector. Die Sprachen erlauben Typerweiterungen (sind *Objektorientiert*) und besitzen ein striktes Typsystem. Beide Systeme verfügen über Unterstützung für nebenläufiges Programmieren.
In Java existieren einige Sprachelemente, die in Active-Oberon nicht vorhanden sind (Interfaces, Exception-Handling), die jedoch in ähnlicher Art und Weise in Active-Oberon aufgenommen werden sollen [26].
Bei genauerer Betrachtung werden aber auch grundlegende Unterschiede deutlich. Den zwei Systemen liegen zwei verschiedene Philosophien zugrunde. Die Architekten des Oberon-Systems liessen sich von Einsteins Motto leiten: *"make it as simple as possible but not simpler"*. Aos versucht diesen Grundsatz weiter zu führen.
Das entstandene System ist kompakt. Die Programmiersprache ist klein und effizient kompilierbar. Daten können sowohl auf dem Heap als auch auf dem Stack alloziert werden. Die Programmiersprache an sich erlaubt keinen direkten Zugriff auf Systemresourcen. Mit dem `MODULE SYSTEM` werden aber auch Operationen ermöglicht, die in der Sprache verboten sind. Mit Hilfe dieses Moduls lässt sich in Active-Oberon grundsätzlich alles realisieren, was auf der Hardware möglich ist (low-level Prozeduren, Anbindung der Hardware).
Das System ist aus verschiedenen Modulen aufgebaut. Die exportierten Prozeduren und Datentypen können in anderen Modulen wiederverwendet werden. Die Schnittstellen der Module sind die API des System. Somit ist das System sehr flexibel und alles wäre vorhanden, um ein Maximum an Code-Reuse zu erreichen.
Leider wurde nach Meinung des Autors in vielen Fällen diesen Schnittstellen zu wenig Aufmerksamkeit geschenkt. Als Beispiel sei die Behandlung von Zeichenketten erwähnt. Es existiert ein Modul `Strings` mit einigen grundlegenden Funktionen für die Behandlung von Zeichenketten. Allerdings sind die Schnittstellen für einige dieser Funktionen so ungeschickt, dass deren Funktion nur eingeschränkt genutzt werden kann. Zudem werden einige Funktionen dieser String-Bibliothek bereits beim Start-Up des Systems gebraucht. Diese Funktionen sind in den entsprechenden Modulen ein weiteres Mal implementiert.
Dieses Beispiel zeigt, dass der Code-Reuse nicht dadurch erreicht wird, dass allgemeine Design-Pattern (vergleiche [11]) eingesetzt wurden, sondern Code-Blöcke kopiert werden.
Beim Design des Java-Systems wurde dem Komfort für den Benutzer des Systems grosse Bedeutung zugemessen. Wie sich die Konzepte einfach und effizient umsetzen lassen, schien eher von untergeordneter Bedeutung. Verschiedene Konzepte der Sprache Java sind inhärent ineffizient [2].
In [30] wird darauf hingewiesen, dass die Bytecode-Verifikation für die aktuelle Sprache Java unmöglich ist. Das heisst, dass sich dieses Konzept überhaupt nicht umsetzen lässt!
Die zur Verfügung stehenden Bibliotheken spielen in Java eine zentrale Rolle. Diese Bibliotheken bieten eine Unmenge von Funktionalitäten. Einige Funktionalitäten scheinen mehr Kosten als Nutzen zu verursachen (Beispiel: `String.intern()`). Mit ihrer Fülle an Funktionalität ist die API jedoch sehr

[2] Beispiele: Nur primitive Datentypen können auf dem Stack alloziert werden. Mehrdimensionale Arrays werden als Arrays von Arrays angelegt.

komfortabel. Sicher tragen diese API's von Java auch viel zur Popularität dieser Sprache bei.
Die Programmiersprache Java kann nicht eindeutig von der API getrennt werden. Teile dieser API sind Bestandteil der Sprache (Object, Thread). Die Implementierungen dieser Klassen benutzen viele andere Klassen der API [3]. Durch diesen Einbezug der Library in die Sprache, wird die Sprache äusserst umfangreich.
Für die Realisierung der API's ist Java auf sogenannte `native` Methoden angewiesen. Das Java-System kann nicht vollständig in Java geschrieben werden.
Im Unterschied zu Active-Oberonprogrammen werden Javaprogramme in Class-Files übersetzt. In [9] [10] wird darauf hingewiesen, dass dieses Codeformat ungeeignet ist, um effizient verifiziert werden zu können. Auch für eine On-the-Fly Übersetzung des Bytecodes in Maschinecode gibt es Formate, die bessere Optimierungen erlaubten.

[3] Die Klasse `java.lang.String` referenziert mehr als 70 weitere Klassen, wenn die API von [6] verwendet wird.

Kapitel 3

Die Aos-JVM

3.1 Einleitung

In diesem Kapitel wird ein Überblick über die Aos-JVM gegeben. Die Konzepte und die Struktur der JVM werden erklärt. Detaillierte Informationen zur Implementierung sind im Kapitel 4 vorhanden.
Grundsätzlich wurde versucht, die Java-Strukturen auf das Aos-Laufzeitsystem abzubilden. In gewissen Fällen war eine solche Abbildung ohne weiteres möglich. In anderen Fällen mussten zusätzliche Mechanismen gefunden werden.
Die vorliegende JVM ist in der Lage, einfache Javaprogramme auszuführen. Die Beschränkungen liegen nicht in der Ausführung von Bytecode-Instruktionen sondern, bei den zur Verfügung stehenden Bibliotheken!

3.2 Darstellung von Daten

3.2.1 Basistypen

Die Basistypen von Java können zum grössten Teil auf einen Active-Oberon-Datentyp abgebildet werden. Tabelle 3.1 zeigt, wie die einzelnen Java-Typen

Java-Type	Oberon-Type
`boolean`	`BOOLEAN`
`byte`	`SHORTINT`
`char`	`INTEGER`
`short`	`INTEGER`
`int`	`LONGINT`
`long`	`HUGEINT`
`float`	`REAL`
`double`	`LONGREAL`

Tabelle 3.1: Mapping zwischen Java-und Oberon-Basistypen

nach Active-Oberon abgebildet werden. Ausnahmen existieren bei den Typen

`HUGEINT` sowie bei `LONGREAL`. `HUGEINT` wurde zum Zeitpunkt dieser Arbeit nicht vollständig vom Oberon-Compiler unterstützt. Dieser Typ musste mit entsprechenden low-level Prozeduren in Oberon emuliert werden. `LONGREAL` wird vom Active-Oberon-Compiler unterstützt. Es gibt jedoch Unterschiede in der Behandlung von Ausnahmen zwischen Aos und Java. In Java gibt es keine Floatingpoint Exceptions! Zudem kennt Active-Oberon keine Modulus-Operation für Floatingpoint-Daten.

3.2.2 Klassen, Instanzen, Interfaces

Beim Laden einer Java-Objektklasse wird für diese Klasse ein Aos-Modul generiert. Dieses Modul enthält die Typbeschreibung der Klasse sowie die statischen Variablen dieser Klasse. Ein so geladenes Modul wird direkt in den Speicher gestellt.
Java-Objektinstanzen werden in Protected Objects, wie sie in 2.3 erklärt sind, abgebildet. Sie werden gleich behandelt, wie übliche Aos-Rekords, welche auf dem Heap alloziert wurden. Der Garbage-Collector kann diese Objekte einsammeln, wenn sie nicht mehr gebraucht werden. Die Klasse `java.lang.Object` wird durch den Active-Oberon-Typ `Object` implementiert. Die Implementierung liegt vollständig in Aos vor!
Mit dem Aos-Modul und dem Aos-Typ kann jedoch nicht die gesamte Information einer Klasse festgehalten werden. Deshalb wird ein Klassen-Objekt benötigt, welches die zusätzliche Information für den Betrieb der JVM bereitstellt.
Interfaces werden im jetzigen Aos-System nicht unterstützt. Für die Implementierung von Interfaces mussten separate Strukturen in Form von Methoden-Rekords geschaffen werden.
Die Verifikation des Class-Files ist noch unvollständig. Im jetzigen Zeitpunkt werden nur static Constraints geprüft, insofern das für die Generierung von Maschinencode notwendig ist.

3.3 Abbildung des VM-Instruktionssatzes

Aos ist zum jetzigen Zeitpunkt nur auf Intel-PC's verfügbar. Der Bytecode wird deshalb ebenfalls in Maschineninstruktionen eines Intel 80836 abgebildet (siehe [7]). Die Aos-JVM macht also eine *Just in Time Compilation*. Der jetzige Codegenerator übersetzt jede Instruktion mit einem geeigneten Code-Muster. Es wird keine Intermediate-Repräsentation (vergleiche [14]) gebildet und keine Optimierungen werden vorgenommen (vergleiche [14],[5]). Bei der Just in Time Compilation wird immer eine ganze Klasse auf's mal übersetzt.
Für die Java-Methoden wird die Aos-Calling-Convention erzwungen.
In 2.2 wurde gezeigt, dass einige Bytecode-Instruktionen komplizierte Abläufe beschreiben. In einigen Fällen wurden solche Instruktionen durch Active-Oberonprozeduren ausgeführt. Das entsprechende Assembler-Codemuster schiebt die Parameter auf den Stack, und springt in die entsprechende Prozedur. Anschliessend werden die Return-Werte wieder auf den Stack geschrieben.
Für die Implementierung der ca. 200 VM-Instruktionen wurden zwölf sogenannte System-Calls eingeführt.

3.4 Nebenläufigkeit (*Concurrency*)

Die Java-Threads wurden auf die Aos-Threads abgebildet. Die API von Aos wurde geringfügig angepasst, um diese Abbildung zu ermöglichen.
Java-Threads sind eigene Objekte, welche einen Aos-Thread benutzen, um ihre Funktion zu erfüllen.
Ein explizites Thread-Objekt wird benötigt, um die Threads in der Klassenhierarchie von Java einzugliedern. Dieses Java-Threadobjekt stellt einige Dienste zur Verfügung, welche von Aos nicht direkt angeboten werden.

3.5 Exception-Handling

Im Aos wurde die Möglichkeit geschaffen, für jeden Thread einen eigenen Exception-Handler zu installieren. Zudem kann der Exception-Handler dem Betriebssystem mitteilen, wo die Ausführung des Programmes fortgeführt werden soll und wie der Stack-Pointer und der Frame-Pointer gesetzt werden sollen.
Für die JVM musste eine Struktur definiert werden, um die verschiedenen Handler zu registrieren. Zudem musste eine Handlerprozedur definiert werden, welche bei einer Exception vom Betriebssystem aufgerufen werden kann, die Datenbank mit den registrierten Handlern durchsucht und dem Betriebsystem die entsprechende Information zurückliefert.
Im Normalfall ergibt sich zur Laufzeit kein Overhead durch das Exception-Handling (Zero Overhead Exception Handling [8],[22]) [1]).

3.6 Interopabilität

Aus den obigen Beschreibungen ist klar, dass ein Java-Objekt in der Aos-JVM benutzt werden kann, wie irgend ein anderes Objekt auf dem Heap. Einerseits könnten Java-Programme mit einigen Einschränkungen auf Aos-Funktionalitäten zugreifen (Java hat seine eigene Typhierarchie). Andererseits könnten Oberon-Programme Funktionalitäten von Java nutzen. Code, der in Oberon geschrieben wurde, wird in demselben Thread ausgeführt wie der Java-Code. Das Problem bei der Benutzung von Funktionen und Objekten aus Aos beziehungsweise Java besteht darin, dass die Compiler von Java und von Aos nichts von der Welt des anderen Systems wissen.
Eine Möglichkeit, die gewählt wurde, um Active-Oberoncode für die Implementierung von Java-Funktionalität zu verwenden, besteht darin, die entsprechenden Deklarationen sowohl in Aos als auch in Java zur Verfügung zu stellen. Diese Möglichkeit wurde für die Implementierung einiger grundlegender Java-Klassen (`Object`,`Throwable`) gewählt.
Als Alternative zu diesem Mechanismus kann in Aos über die Java-Reflection auf die Java-Objekte zugegriffen werden. Das ist nur möglich, weil der `native` Oberon-Code in demselben Thread ausgeführt wird wie der Java-Code. Diese Schnittstelle ist nicht effizient, da viel zur Laufzeit des Programmes gemacht werden muss, was bereits zur Übersetzungszeit gemacht werden könnte. Im Gegensatz zum oben beschriebenen Mechanismus bietet diese Methode aber den grossen Vorteil, dass die Klassenhierarchie von Java nicht bekannt sein muss.

[1] Eine Ausnahme bildet das Locking und Unlocking von Objekten (vergleiche 4.3

Möchte man ausschliesslich über die vordeklarierten Schnittstellen auf die Java-Objekte zugreifen, so ergäben sich zwei Probleme:
In der Java-API ist alles von allem abhängig. Aus diesem Grund müsste fast die gesamte Java-API-Schnittstelle in Oberon verfügbar gemacht werden. Das würde viel Speicher benötigen. Andererseits erlaubt Oberon keine zyklischen Imports. In Java sind zyklische Imports jedoch erlaubt und in grosser Zahl vorhanden. Um die gesamte Java-API-Schnittstelle in Oberon verfügbar zu machen, müssten alle zyklischen Abhängigkeiten aufgelöst werden. Das wäre mit grossem Aufwand verbunden.
Aus diesen Gründen rechtfertigt sich der Einsatz einer weniger effizienten Schnittstelle für gewisse Fälle.

3.7 Die Kernmodule der Aos-JVM

In Abbildung 3.1 sind die wichtigsten Module der JVM aufgeführt. Neben diesen Modulen existieren noch einige weitere Module, welche zusätzliche Dienste anbieten, jedoch nicht grundlegend für das System sind. In den folgenden Abschnitten werden die Kernmodule und ihre wichtigsten Funktionen erklärt.

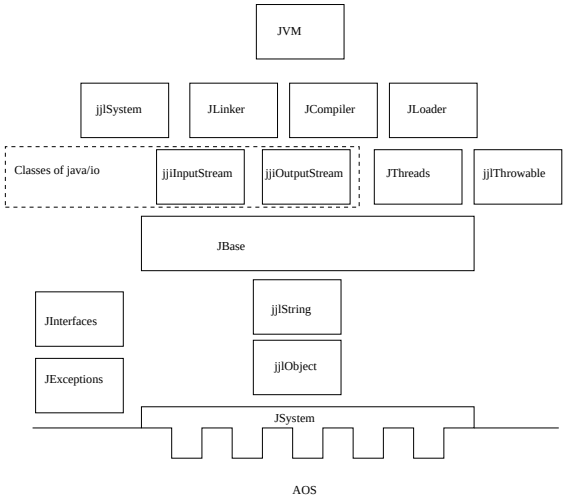

Abbildung 3.1: Kernmodule der Aos-JVM

3.7.1 Modul JSystem

Das Modul JSystem stellt die Schnittstelle zu Aos und zur Hardware dar. Hier werden System-Calls exportiert und low-level Dienste angeboten. Diese Dienste umfassen Operationen auf dem Typdescriptor, Operation auf Modulen, die normalerweise vom Modul-Lader übernommen werden, mathematische Operationen für den Typ `HUGEINT`, Operationen für das Stack-Unwinding sowie Funktionen für das explizite Aufrufen von Funktionen und den Rücksprung aus solchen. Dieses Modul sollte eine Migration der Aos-JVM auf andere Oberon-Plattfor-

men erleichtern. Allerdings sind Threads und das Exception-Handling auf das Aos-System abgestützt.

3.7.2 Module JInterfaces und JExceptions

Diese zwei Module sind Grundlage für die Funktionen der Java-Interfaces sowie der Exceptions. Sie werden im gesamten JVM-Subsystem verwendet. Die zwei Module wurden jedoch so konzipiert, dass ihre Funktionalität auch in anderen Aos-Programmen verwendet werden kann. Sie benutzen keine Klassen des Java-Frameworks (`Object`, `Class`). Ihre Implementierung ist ausführlich in Abschnitt 4.3 beschrieben.

3.7.3 Modul jjlObject

In diesem Modul ist die Java-Klasse `java.lang.Object` als Active-Oberon-Typ definiert. Die gesamte Implementierung liegt in Active-Oberon vor. Die Funktionalität der Java-API 1.2 [17] wird unterstützt.
Die wichtigsten Funktionen des Typen `Object` sind das Locking, Unlocking sowie die Funktionen `wait()` und `notify()` für die Synchronisation von Threads (vergleiche 4.4).
Neben dem Typ `Object` sind auch alle Java-Arraytypen in diesem Module definiert.

3.7.4 Modul jjlString

Die Klasse `java.lang.String` definiert die Behandlung von Zeichenketten. Im Gegensatz zu Aos wird diese Aufgabe zentral im System verankert.
Das Modul jjlString stellt die Schnittstellen zur entsprechenden Java-Klasse zur Verfügung. Alle Funktionen ausser `String.intern()` sind in Java implementiert. Die Implementierung wurde von [6] übernommen. Zusätzlich zu den Funktionen der String API mussten noch einige Funktionen in Oberon implementiert werden, um die JVM überhaupt laden zu können [2]. Gewisse Funktionen von `java.lang.String` beruhen auf der Behandlung verschiedener Zeichensätze. Für die Umwandlung von Gross- in Kleinbuchstaben wird die Unicode-Datenbank (vergleiche [33]) von [6] verwendet (Dateien: oberon.jvm.*.uni). Um das zu ermöglichen, wurde die Klasse `java.lang.Character` angepasst.

3.7.5 Die Module JBase, JLoader, JCompiler, JLinker

Die Module JBase, JLoader, JCompiler und JLinker sind sehr eng miteinander verflochten.
In JBase sind die Datenstrukturen für das Laden der Class-Files definiert. Dazu gehören die Strukturen für das Speichern der Information aus dem Constant-Pool. Beim Laden von Java-Klassen werden zudem Class-Objects, Field- und Method-Objects alloziert. Diese werden einerseits für die Übersetzung des Codes benötigt, andererseits stellen diese Typen gerade die Java-Reflection-API dar. In Abbildung 3.2 ist eine Übersicht über die definierten Typen gegeben.

[2] Beim Laden der Klasse `java.lang.String` wird die Klasse `java.lang.String` bereits benutzt.

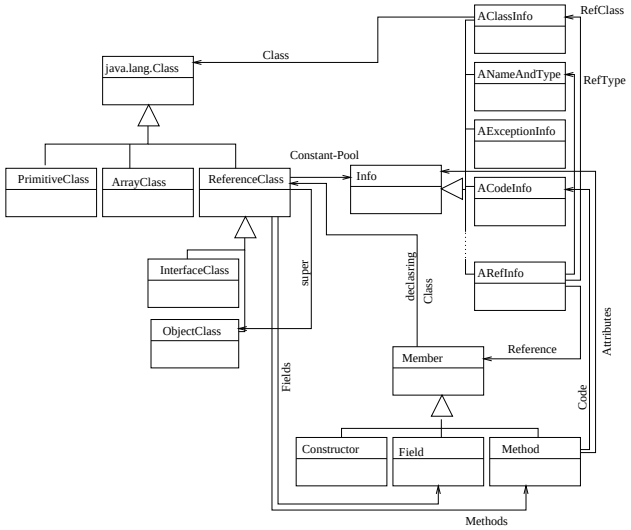

Abbildung 3.2: UML-Klassendiagramm der wichtigsten Datentypen im Modul JBase

Die Klasse `java.lang.Class` stellt in dieser Modellierung eine abstrakte Klasse dar. Sie besitzt die konkreten Erweiterungen `PrimitiveClass`, `ArrayClass` und `ReferenceClass` (für Interfaces und Objekt-Klassen). Primitive-Klassen und Array-Klassen haben wenige zusätzliche Eigenschaften. Sie implementieren das Interface von `java.lang.Class` auf sehr einfache Art und Weise.
Referenz-Klassen werden aus einem Class-File geladen. Sie haben einen eigenen Constant-Pool. Im Constant-Pool haben alle Einträge den Typ `AInfo`. Die Einträge im Constant-Pool referenzieren wiederum Klassen, Methoden, Felder sowie andere `AInfo` Elemente.
Alle Referenz-Klassen können Methoden und Felder besitzen. Alle Elemente des Typs `Member` haben einen Verweis auf die Klasse, deren Teil sie sind.
Im Modul JLoader wird eine Klasse von einem Input-Stream geladen. Für jede Objekt-Klasse und Interface-Klasse wird ein Oberon-Typ alloziert. JCompiler generiert den Maschinencode sowie die Meta-Information zu den Methoden (Reference-Section der Module). JLinker ersetzt die symbolischen Links durch absolute Speicheradressen, JVM-Instruktionsadressen durch Intel-Instruktionsadressen, initialisiert Interfaces und registriert Information für das Exception-Handling.

3.7.6 Modul JThreads

JThreads implementiert die Funktionalitäten von `java.lang.Thread` (vergleiche 4.4). Dazu gehört die Anbindung an die Threads von Aos. Zudem wird ein Typ für das Interface `java.lang.Runnable` implementiert. Die Registrierung der Threads im Java-System ist ebenfalls vorgesehen. Da zwischen den Klassen `java.lang.Thread` und `java.lang.ThreadGroup` eine enge Kooperation stattfindet, wird vorgeschlagen, `java.lang.ThreadGroup` ebenfalls in diesem Modul zu implementieren. Im jetzigen Zeitpunkt ist `java.lang.ThreadGroup` jedoch nicht implementiert.

3.7.7 Modul jjlThrowable

In jjlThrowable liegt die Implementierung von `java.lang.Throwable` vor. Dieser Typ ist die Basisklasse aller Java-Errors- und Exceptions. In diesem Modul werden auch die Up-Calls definiert, welche der Exception-Handler benötigt, um mit dem aktuellen System zusammenzuarbeiten (Mapping von System-Traps auf Java-Exceptions und Sichern des Stacks für die Ausgabe eines Stackdumps zu einem späteren Zeitpunkt).

3.7.8 Klassen aus java.io

Für die Ein- und Ausgabe von Daten sind in Java zahlreiche Klassen vorhanden. Die Schnittstellen der Basisklassen wurden in Aos als eigene Module importiert. Somit kann die Funktionalität dieser Klassen effizient genutzt werden. In Abbildung 3.3 sind die Klassen des IO-Frameworks der Aos-JVM als UML-Klassendiagramm aufgezeichnet.

Für die Ausgabe von Daten wird in Java die Abstraktion des Output-Stre-

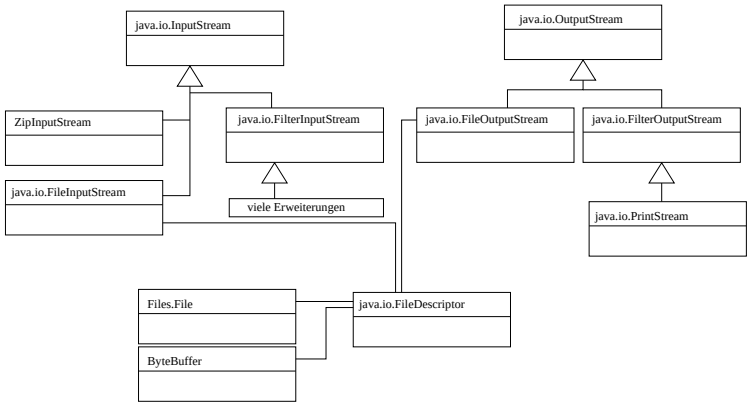

Abbildung 3.3: UML-Klassendiagramm des IO-Framework der Aos-JVM

ams verwendet. Als konkrete Erweiterung der abstrakten Klasse `java.lang.OutputStream` ist die konkrete Klasse `java.io.FileOutputStream` implementiert. Diese Klasse schreibt die Daten entweder in einen zyklischen Puffer, welcher von einem beliebigen Konsumenten geleert werden kann, oder in eine Datei. Mit der Ausgabe der Daten in einen Puffer wird das Konzept von *Standard-Output* simuliert.

Eine weiterer konkreter Output-Stream ist `java.io.PrinStream`. Das Modul jjlPrintStream, welches diese Klasse implementiert, spielte eine wichtige Rolle beim Lösen des Bootstrapping-Problems. Häufig wird diese Klasse verwendet um über `java.lang.System.out` Daten auf den Bildschirm zu bringen. Zu Beginn dieser Arbeit war es wichtig, möglichst bald einfache Java-Programme ausführen zu können. Die Java-Implementierung von `java.io.PrintStream` benutzt jedoch viele andere Klassen. Um zu verhindern, dass all diese Klassen geladen werden, wurde diese Klasse direkt in Active-Oberon implementiert.

Für das Laden der Klassen wird die Abstraktion des Input-Stream verwendet. In Oberon liegen zwei konkrete Erweiterungen der abstrakten Klasse `java.io.-`

`InputStream` vor, (`java.io.FileInputStream` und `ZipInputStream`) [3] . Man beachte, dass die Active-Oberon-Implementierung von `InputStream` benötigt wird, bevor diese Klasse vom Class-File geladen worden ist!

3.7.9 Module joOberonEnv, joJLang

joOberonEnv und joLang sind reine Schnittstellen-Module um Funktionalität von Oberon in Java verfügbar zu machen. Für jedes dieser Module existiert eine Java-Klasse mit derselben Schnittstelle.
In ioOberonEnv werden Funktionalitäten von `Files` nach Java exportiert. `arraycopy()` ist ebenfalls in diesem Modul implementiert. Ursprünglich war vorgesehen, alle `native` Funktionalität über dieses Modul zu exportieren. Es stellte sich jedoch heraus, dass diese Strategie zu einem riesigen Modul führen würde. Für das weitere Vorgehen wird vorgeschlagen, jeweils ein Oberon-Modul pro Java-Package vorzusehen. Als einziges solches Modul ist zum jetzigen Zeitpunkt joJLang vorhanden. In joJLang werden die `native`-Calls von `java.lang.Math` implementiert.

3.7.10 Modul JVM

Das Modul JVM repräsentiert die eigentliche JVM. Dieses Modul steht zuoberst in der Modul-Hierarchie. Von diesem Modul aus werden Java-Programme gestartet (`Execute()`), und die geladenen Klassen können abgefragt werden (`ShowClasses()`).
In Java wird die JVM durch die Klasse `java.lang.Runtime` repräsentiert. Im Modul JVM ist diese Klasse implementiert. Die JVM läuft in einem eigenen Thread. Zu einem gegebenen Zeitpunkt kann jeweils nur eine Instanz einer JVM laufen (Singleton [11]) [4] .
Die JVM wird vom Native-Oberon Thread von Aos gestartet. Das Java-Programm läuft anschliessend in einem separaten Thread. Von diesem Thread können mehrere Java-Threads abgespalten werden. Der Thread der JVM wartet, bis alle Java-Threads terminiert wurden. Das UML-Aktivitätsdiagramm 3.4 beschreibt, was bei der Ausführung des Kommandos `JVM.Execute()` abläuft.

3.8 Ergebnisse

Die vorliegende JVM ist in der Lage, Java-Programme auszuführen, welche nur beschränkt Gebrauch der Libraries machen. Die Performanz ist in der jetzigen Version noch bescheiden. Im Kapitel 5 sind die Tests, mit welchen Zeitmessungen durchgeführt wurden, mit ihren Ergebnissen beschrieben.
Die Abbildung 3.5 zeigt einen Screenshot der JVM-Console nach der erfolgreichen Ausführung des Spec-Benchmarktests `spec.benchmarks._200_ceck.Main`.

[3] `ZipInputStream` hat nicht dieselben Schnittstellen wie `java.util.zip.ZipInputStream`
[4] Beim jetzigen Stand existiert nur ein Namensraum für alle Klassen.

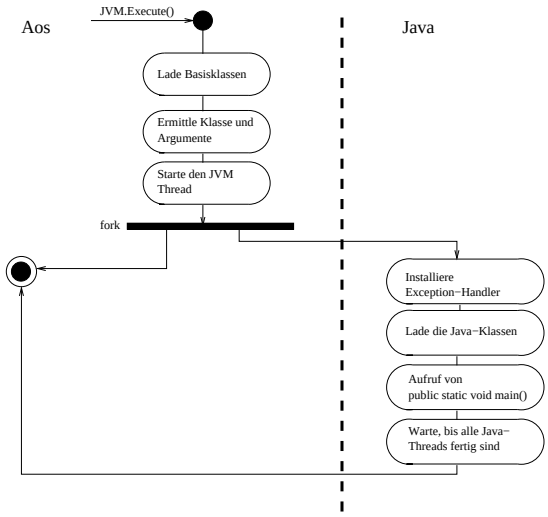

Abbildung 3.4: UML-Aktivitätsdiagramm für das Kommando JVM.Execute()

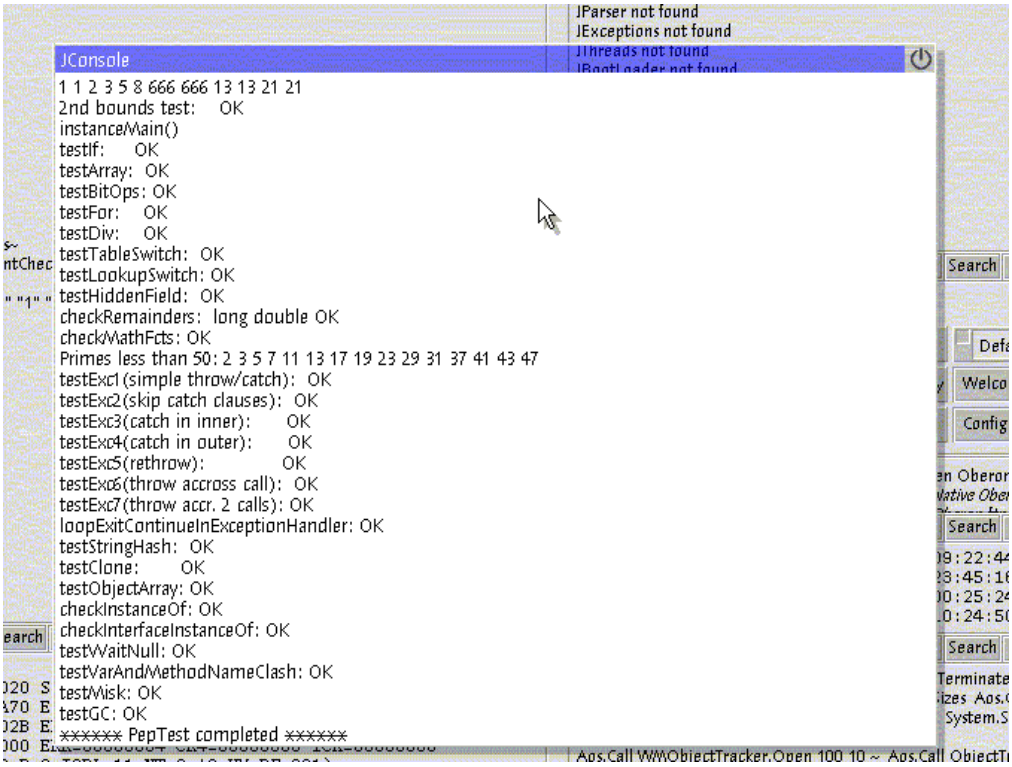

Abbildung 3.5: Screenshot der JVM-Console

Kapitel 4

Implementierung

4.1 Einleitung

Die Hauptthemen dieser Arbeit waren der Design und die Implementierung von Interfaces, des Exception-Handlings und der Concurrency-Unterstützung. Darüber hinaus, sollten die vorhandenen Strukturen an realen Java-Beispielsprogrammen getestet werden.
Bald stellte sich heraus, dass in den vorhandenen Programmteilen noch so viele Fehler enthalten waren, dass die Fehlersuche und Korrektur des bestehenden Codes zu einem wichtigen Bestandteil dieser Arbeit wurde. Insbesondere nahm auch die Entwicklung von Intel-386-Codemustern, sowie Korrekturen am Lader und am Compiler viel mehr Raum ein, als anfänglich angenommen.
In diesem Kapitel wird eine detaillierte Beschreibung der gewählten Implementierung von Interfaces, Exception-Handling und der Concurrency-Unterstützung gegeben. In einem weiteren Abschnitt werden summarisch weitere Aspekte beschrieben, wie eine Übersicht über den Ladevorgang, die zusätzlichen Code-Muster sowie über die System-Aufrufe.

4.2 Implementierung von Interfaces

4.2.1 Was sind Interfaces

In diesem Abschnitt soll die Semantik von Interfaces anhand von zwei unterschiedlichen Interface-Definitionen kurz diskutiert werden.

Java-Interfaces

Java-Interfaces sind Schnittstellen zu einem konkreten Objekt. Wenn eine Klasse ein Interface implementiert, überprüft der Compiler, ob die entsprechenden Methoden auch wirklich implementiert werden. Mit anderen Worten, Interfaces definieren eine Sicht eines Objektes oder einen Vertrag, welcher von diesem Objekt erfüllt wird.
Neben Methoden, können in Java-Interfaces auch Konstanten deklariert werden. Die Werte der Konstanten werden bei der Übersetzung eines Java-Programmes zu Bytecode direkt in den Code eingeführt. Im übersetzten Bytecode gibt es keinen Hinweis mehr auf das verwendete Interface (im Class-File sind die Felder

der Interfaces aufgeführt).
Aus dem oben Gesagten wird klar, dass Java-Interfaces etwas Abstraktes, ohne eigene Implementierung sind. Wird in einem Java-Programm mit einer Variablen `x` des Typs `InterfaceA` gearbeitet, so wird erwartet, dass `x` auf ein konkretes Java-Objekt zeigt, das die vereinbarten Eigenschaften hat. Das konkrete Objekt `x` besitzt aber durchaus noch andere Eigenschaften, die in diesem Zeitpunkt nicht interessieren.

Interfaces im *Component-Object-Model*

Interfaces, wie sie im Component-Object-Model von Microsoft (COM) ([4],[31]) definiert sind, unterscheiden sich in einigen Punkten ganz wesentlich von Java-Interfaces. In Abbildung 4.1 ist die Struktur eines COM-Objektes mit einem Interface dargestellt.

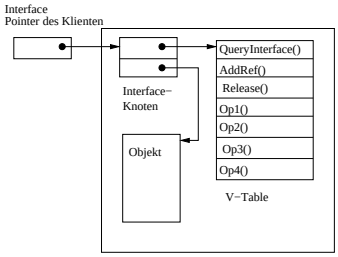

Abbildung 4.1: Struktur einer COM-Komponente, welche ein einziges Interface implementiert

Auch im COM sind Interfaces Schnittstellen zu einem Objekt beziehungsweise einer Komponente. Diese Schnittstellen sind jedoch das Einzige, was von der Komponente sichtbar ist. Jede Kommunikation mit einer COM-Komponente geschieht über Schnittstellen. Eine Schnittstelle besitzt auch eine eigene physische Repräsentation. Diese Repräsentation besteht aus einem Zeiger auf einen Interface-Knoten. Ein Interface-Knoten besitzt wiederum einen Zeiger auf eine deklarierte Methodentabelle. Innerhalb dieser Methodentabelle besitzt jede Methode eine eindeutige Nummer. Somit kann eine Abbildung einer Methodennummer auf eine effektive Methodenadresse eines Objektes vorgenommen werden. Da die Methodennummer bereits zur Übersetzungszeit bekannt ist, kann ein solcher Methodenaufruf ohne Laufzeitmechanismen übersetzt werden. Die ersten drei Einträge in dieser Methodentabelle sind Methoden für die Abfrage von Interfaces der entsprechenden Komponente (`Query Interface()`, `AddRef()`, `Release()`). Mit der Methode `QueryInterface()` kann ein Interface einer Komponente verlangt werden. Der Klient erhält dabei einen Zeiger der oben beschriebenen Form. Benötigt ein Klient ein anderes Interface, so erhält er auch einen anderen Zeiger.

Vergleich der beiden Interface-Definitionen

Bei COM-Objekten sieht ein Klient einer Komponente nur gerade einen Zeiger auf ein verlangtes Interface. Das Interface hat eine konkrete binäre Struktur,

die zur Übersetzungszeit bekannt ist.
Im Gegensatz dazu sieht ein Klient bei Java-Interfaces immer eine Referenz auf das gesamte Objekt. Ein Typecast oder eine Zuweisung auf eine Variable des Typs Interface verändert eine Referenz auf ein Objekt nicht. Die Benutzung des Interfaces garantiert lediglich, dass das entsprechende Objekt wirklich über die im Interface deklarierten Eigenschaften verfügt. Der Klient darf sicher sein, dass der Vertrag erfüllt wird. Es wird jedoch keine binäre Struktur für ein Interface vorgeschrieben.
Diese unterschiedlichen Definitionen haben ihre Auswirkungen auf das Laufzeitverhalten.
Bei einem COM-Interface muss während der Ausführung eines Programmes ein Interface abgefragt werden. Der zurückgegebene Zeiger besitzt eine definierte Struktur. Somit können die effektiven Methodenadressen ohne zusätzliche Unterstützung des Laufzeitsystems ermittelt werden.
Mit der Java-Interface-Definition ist beim Programm-Start lediglich bekannt, dass ein Objekt an einer bestimmten Stelle bestimmte Methoden implementieren muss. Zur Laufzeit muss geprüft werden, ob das entsprechende Objekt wirklich ein Interface implementiert und wo die spezifizierte Methode ist.
Wird die Java-Interface-Definition direkt umgesetzt, ergibt sich daraus ein grosser Aufwand zur Laufzeit eines Programmes.

4.2.2 Design von Interfaces für die Aos-JVM

Konzepte

Ein Java-Interface wird in der Aos-JVM durch einen Aos-Typdescriptor und ein `Class`-Objekt dargestellt. Der Typdescriptor definiert einen Methoden-Rekord. In diesem Methoden-Rekord ist für jede Interface-Methode eine Prozedurvariable vorgesehen. Ein Methoden-Rekord besitzt den Basistypen `JInterfaces.VTbabel`. Für Methoden-Rekord eines Interfaces wird nicht von anderen Rekord-Typen erweitert. Die Information bezüglich der Super-Interfaces ist im Class-Objekt festgehalten.
Eine Java-Klasse, welche ein Interface implementiert, alloziert den Methoden-Rekord des Interfaces und initialisiert die Prozedurvariablen mit den Adressen der entsprechenden Methoden dieser Klasse. Die eigentlichen Methodenimplementierungen sind in der Klasse, welche das Interface implementiert. Sie sind an den Objekttyp gebunden.
Falls das implementierte Interface von einem anderen Interface abgeleitet wurde, wird der Methoden-Rekord der Basisklasse implizit alloziert und initialisiert. Die Subklassen einer Java-Klasse, welche ein Interface implementiert, implementieren alle Interfaces der Superklasse auf die oben beschriebene Art.
Bei einem Aufruf von `invokeinterface` muss zur Laufzeit der Methoden-Rekord des referenzierten Interfaces gesucht werden. Zu diesem Zweck wurde ein System-Call eingeführt (`ILookup( ObjectClass, InterfaceClass)`). Abbildung 4.2 zeigt die Formulierung der Instruktion `invokeinterface` in einem Pseudocode.

In Abbildung 4.3 ist dargestellt, wie ein Rekord R, welcher ein Interface A und B implementiert, mit den erklärten Strukturen im Speicher dargestellt wird.

```
INSTRUCTION invokeinterface(ObjectRef, MethodRef)
  ObjectClass := ObjectRef.Class.descriptor.tag;
  InterfaceClass := MethodRef.Class.descriptor.tag;
  FunktionAdress := MEM[ ILookup(ObjectClass, InterfaceClass),
                    + MethodRef.MethodNr*4 ];
  Call FunktionAdress;
END invokeinterface;
```

Abbildung 4.2: Instruktion Invokeinterface

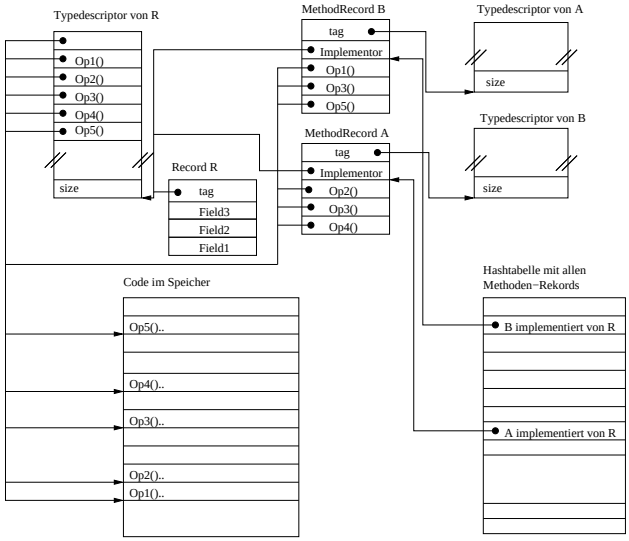

Abbildung 4.3: Struktur eines Objektes in der Oberon VM, das zwei Interfaces A und B implementiert

Lookup von Interfaces

Für den System-Call `InterfaceLookup()` wurden zwei Varianten diskutiert. Die erste, dieser zwei Varianten wurde implementiert.
In dieser ersten Variante werden die Methoden-Rekords aller implementierten Interfaces in einer globalen Hashtabelle gespeichert. Als Schlüssel für das Hashing werden das Typtag der Interface-Klasse sowie das Typtag der Objekt-Klasse verwendet. Es wurde ein doppeltes Hashing wie in [25] beschrieben verwendet.
Die Geschwindigkeitsmessungen zeigten, dass mit der Implementierung des Lookups viel Zeit verschenkt oder gewonnen werden kann. Es wurde darauf geachtet, dass die Instruktion `idiv` nicht verwendet werden muss. Die `MOD`-Operation kann durch eine Reihe von Bit-Operationen, Additionen und Subtraktionen berechnet werden. Die Wahl der Grösse der Hashtabelle kann diese Berechnung vereinfachen. In Abbildung 4.4 ist beschrieben, wie eine Modulo-Operation ausgedrückt werden kann.
Mit solchen Massnahmen konnte die Geschwindigkeit des Interface-Lookups verdreifacht werden!
Die andere Variante benutzt eine lineare Liste, um die implementierten Interfa-

$0 \leq x < 2^{32}$ und $m = 8191 = 2^{13} - 1$
$x = r1 + (r2 + n * 2^{13}) * 2^{13}$
$y = (x\&\texttt{1FFFH}) + ((x \gg 13)\&\texttt{1FFFH}) + (x \gg 26)$
$x \bmod m = y - z$ für $z = 0 \vee z = m \vee z = 2 * m$

Abbildung 4.4: x mod 8191

ces einer Klasse zu speichern. Durch eine *move to front* [25] Strategie wird das zuletzt abgefragte Interface an den Beginn der Liste gehängt. Auf diese Weise sollte es möglich sein, die Suchwege kurz zu halten.
Auf den ersten Blick erscheint die Variante mit der Hashtabelle effizienter zu sein. Allerdings muss darauf hingewiesen werden, dass die Hashtabelle genügend gross sein muss. Wie gross ist genügend gross? Für kleine Programme wird viel Speicher verschwendet. Für grosse Programme kann die Performanz leiden.
Bei der zweiten Variante muss man sich bewusst sein, dass einerseits die Listen kurz sind. Zudem wird ein Interface meistens wiederholt verwendet. In der zweiten Variante wird nicht mehr Speicher verwendet, als nötig. Somit ist es fraglich, ob eine Move to Front Liste nicht eine geeignete Wahl für die Implementierung des InterfaceLookups() wäre.

Erklärungen

Wie in 4.2.1 erwähnt, resultiert eine naive Umsetzung der Java-Interface-Semantik in einem grossen Aufwand zur Laufzeit eines Programmes. Anhand von Methodentabellen, wie sie im COM eingesetzt werden, lässt sich dieser Laufzeitaufwand reduzieren. Jede Methode hat bezüglich eines Interfaces eine eindeutige Nummer, welche einmal beim Übersetzen ermittelt werden muss. Die Methoden-Rekords, welche für jedes Interface gebildet werden, entsprechen genau den Methodentabellen beziehungsweise *vTables*.

Die Methoden-Rekords verwenden grundsätzlich keine Vererbungsmechanismen. Die Information über die Superinterfaces ist in den Class-Objekten festgehalten. Eine Klasse, die ein Interface mit Super-Interfaces implementiert, alloziert einen Methoden-Rekord für das entsprechende Interface, sowie je einen für jedes Basis-Interface. Man betrachte dazu folgende zwei Beispiele:

a) Einsatz von mehrfacher Interface-Vererbung

```
interface IA{
    public void f1();
}
interface IB{
    public void f2();
}
interface IC extends IA, IB{
   public void f3();
}
class C implements IC {
   public void f1(){..}
   public void f2(){..}
```

```
    public void f3(){..}
}
```

b) Keine Interface-Vererbung

```
interface IA{
    public void f1();
}
interface IB{
    public void f2();
}
interface IC{
   public void f3();
}
class C implements IA,IB,IC {
   public void f1(){..}
   public void f2(){..}
   public void f3(){..}
}
```

Sowohl im Fall a) als auch im Fall b) muss die Klasse C die Funktionen `f1()`, `f2()`, sowie `f3()` implementieren, um die Verträge zu erfüllen. In beiden Fällen kann eine Variable des Typs C einer Variablen des Typs `IA`, `IB` oder `IC` zugewiesen werden. Aus der Sicht des Objektes sind demzufolge beide Varianten identisch.
Wäre es nicht möglich, die Information der Basis-Interfaces im Methoden-Rekord des Subinterfaces einzuschliessen? Anschliessend könnte man nur mit dem Methoden-Rekord von `IC` arbeiten um die effektiven Methodenadressen zu finden! In einer Klasse müssten weniger Methodentabellen verwaltet werden.

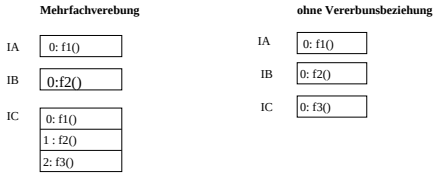

Abbildung 4.5: Methodentabellen für Mehrfachvererbung

Wie die Methoden-Rekords für beide Varianten aussehen könnten, ist in Abbildung 4.5 dargestellt. Man betrachte nun folgendes Codebeispiel:

```
C x = new C();
IC y = x;
y.f1(); y.f2();
```

Der Compiler[1] übersetzt diese Methodenaufrufe in

```
invokeinterface IA.f1()
invokeinterface IB.f2()
```

[1] Javac von Sun

Versucht nun das Laufzeitsystem diese Aufrufe anhand der Methodentabelle von `IC` aufzulösen, so geschieht folgendes: `f1()` hat die Methodennummer Null. Somit wird die Methode `f1()` des Objektes x aufgerufen. `f2()` hat in der Methodentabelle `IB` ebenfalls die Nummer Null und fälschlicherweise wird wiederum die Methode `f1()` des Objektes x aufgerufen.
Wird die Methodentabelle `IC` anders aufgebaut, so funktionieren die Aufrufe für die Methoden von `IB`, hingegen schlagen die Aufrufe der Methoden von `IA` fehl.
Die Idee mit der Kumulierung der Methodentabellen der Superinterfaces funktioniert bei Einfachvererbung. Bei Mehrfachvererbung ist die Situation jedoch komplizierter. Innerhalb einer Methodentablle müsste zusätzliche Information über die Methodentabellen der Superinterfaces abgelegt werden.
Aus diesem Grund wurde grundsätzlich auf eine Vererbungsbeziehung verzichtet.

Schlussbemerkungen

Der vorgestellte Ansatz verwendet Ideen des COM, insbesondere Methodentabellen und einen allgemeinen Mechanismus zur Abfrage der implementierten Interfaces. In der vorgestellten Lösung wird das eigentliche Objekt nicht durch die Interfaces abgeschirmt. Sie stellen lediglich einen zusätzlichen Weg dar, um auf die Funktionen eines Objektes zuzugreifen. Der vorgestellte Mechanismus ist vom Laufzeitverhalten analog zum COM-Interfacemechanismus. Durch den Aufruf einer Interface-Methode oder den Typecast wird die Objekt-Referenz nicht verändert und alle Instruktionen können einfach übersetzt werden.
Bei der Überprüfung von Zuweisungen und der Überprüfung von `is a` Beziehungen muss die Abfrage des Interfaces zusätzlich zum eigentlichen Typtest gemacht werden. Eine Umsetzung dieser Interface-Implementierung in Active-Oberon kein Problem.
Welche Datenstruktur am geeignetesten ist, müsste in konkreten Tests verifiziert werden. Neben der Geschwindigkeit für den Interface-Lookup sollte auch die Speichereffizienz berücksichtigt werden!
Aber auch die Implementierung der gewählten Variante spielt eine entscheidende Rolle. Die beste Strategie wird keine guten Ergebnisse zeigen, wenn sie nicht sorgfältig umgesetzt wird!

4.3 Implementierung von Exceptions

4.3.1 Java-Exceptions

Ausnahmen, die während der Ausführung eines Programmes auftreten, werden in Java mit sogenannten Exceptions behandelt.
Dort, wo es zu einer Ausnahme kommt, wird eine Exception geworfen. Dadurch wird die Ausführung des Programmes unterbrochen und an einer anderen Stelle weitergeführt.
Der Code, bei welchem die Ausführung fortgesetzt wird, heisst Exception-Handler. Dieser Exception-Handler kann innerhalb derselben Prozedur definiert sein, in welcher die Ausnahme aufgetreten ist. Es ist auch möglich, dass der entsprechende Exception-Handler in einer anderen Prozedur gesucht werden muss. Es ist Aufgabe des Laufzeitsystems, den Stack zurückzuverfolgen und einen passenden Exception-Handler zu suchen. Wenn kein Exception-Handler gefunden

werden kann, so wird der laufende Thread beendet. Abbildung 4.6 zeigt eine Situation auf dem Stack, in welcher viele Frames vom Stack entfernt werden müssen um die Exception zu behandeln. Objekte, die auf dem Heap alloziert wurden, bevor die Exception geworfen wurde und die nach der Behandlung der Exception nicht mehr referenziert werden, werden vom Garbage-Collector abgeräumt. Falls eine Exception innerhalb eines `synchronized` Blocks auftritt, muss das entsprechende, gelockte Objekt wieder frei gegeben werden.
Die beschriebene Art von Ausnahmebehandlung erlaubt es, das eigentliche Programm sauber von der Ausnahmebehandlung zu trennen.

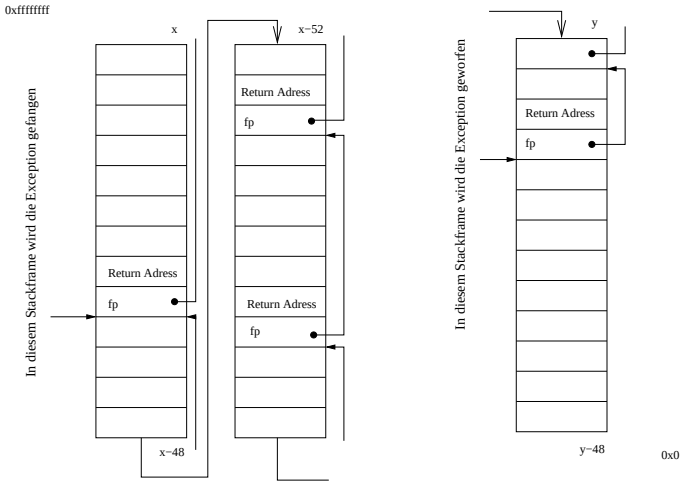

Abbildung 4.6: Möglicher Stackzustand beim Werfen einer Exception

4.3.2 Exceptions in der Aos-JVM

Ausnahmen während der Programmausführung erzeugen in Aos einen Trap (Software- oder Hardware-Exception). Im Abschnitt 2.3 wurden die Mechanismen, wie diese Traps behandelt werden, kurz erklärt. Im Normalfall wird in Aos beim Auftreten eines Traps die Ausführung des entsprechenden Kommandos abgebrochen. Die ausgegebene Information kann benutzt werden, um den Fehler im Programm zu beheben.
In [22] werden Mechanismen für das Exception-Handling in Oberon beschrieben, die es erlaubten, das Programm an einer anderen Stelle weiter zu führen. Eine solche Möglichkeit ist in den Fällen notwendig, wo die Exception nicht durch einen Fehler im Programm, sondern durch ein externes Ereignis hervorgerufen wurde. In verteilten Systemen ist es beispielsweise durchaus möglich, dass einmal ein Rechner nicht ansprechbar ist. Ein solches Ereignis muss separat behandelt werden, darf aber nicht zur Beendigung des Programms führen.
In der Aos-JVM wurde die Möglichkeit vorgesehen, Programmteile direkt in Active-Oberon zu schreiben. Deshalb war es nötig, einen Exception-Mechanismus zu definieren, der es erlaubt, Exceptions, welche in Java-Programmteilen geworfen wurden, in Oberon-Programmteilen zu behandeln und umgekehrt. Der Mechanismus für die Behandlung von Exceptions in der Aos-JVM benutzt die Elemente des Aos-Systems zur Behandlung von Traps und ergänzt diese. In den

folgenden Abschnitten werden die Einzelheiten dieses Exception-Handlings erklärt. Beim Design des Exception-Handlings standen Ideen des Zero Overhead Exception Handling im Vordergrund (vergleiche [8], [22]).

4.3.3 Design des Exception-Handlings in der Aos-JVM

Um einen Codeblock vor Exceptions zu schützen, muss ein Handler für diesen Codeblock registriert werden. Ein geschützter Codeblock kann eine ganze Prozedur aber auch nur eine kurze Sequenz von Instruktionen sein. Der geschützte Codeblock kann mehrere Exception-Handlers haben (`ExceptionHandler.catch`). Dies erlaubt es, den Block in kleinere geschützte Teilblöcke aufzuteilen, oder aber eine Bedingung an die Ausführung eines bestimmten Handlers zu knüpfen (`CatchDesc.handlesIt()`. Die genaue Struktur eines solchen Registrierungseintrags ist in 4.7 dargestellt.

```
TYPE
  ExceptionHandler = RECORD TO ExceptionHandlerDesc;
  CatchDesc = RECORD
    start, end : LONGINT;       (* for this range                  *)
    handler : LONGINT;          (* handler is invoked              *)
    ex      : PTR;              (* some pointer to hold additional
                                   info for evaluate               *)
    handlesIt: PROCEDURE( ex: PTR; adr :LONGINT )
                                (* must be TRUE, to invoke handler *)
  END;
  CatchTable : POINTER TO ARRAY OF CatchDesc;
  ExceptionHandlerDesc = RECORD
    left, right : ExceptionHandler;
    balance : SHORTINT;         (* for the tree only               *)
    start, end : LONGINT;       (* beginning and end of procedure  *)
    inner : BOOLEAN;            (* handler is an local procedure   *)
    mod : MODULE;               (* module that defines handler     *)
   catch : CatchTable;          (* handler information             *)
  END;
```

Abbildung 4.7: Datenstrukturen des Exception-Handling-Systems

In Java und in Active-Oberon werden geschützte Codeblöcke auf verschiedene Weise definiert. In Java sind Exceptions Bestandteil der Sprache. In Abbildung 4.8 ist beschrieben, wie geschützte Codeblöcke definiert werden und Exceptions geworfen werden können. Der Compiler und das Laufzeitsystem erzeugen die entsprechenden Exception-Handler-Registrierungen.
In Active-Oberon existiert zur Zeit keine Sprachunterstützung. Ein Oberon-Exceptionhandler wird als innere Prozedur deklariert. Er hat als expliziten Parameter das Exception-Objekt und muss denselben Rückgabewert wie die umschliessende Prozedur aufweisen. In [22] wird eine Variante beschrieben, in welcher das Laufzeitsystem den Handler ohne vorherige Registrierung anhand seiner Parameter und Rückgabewerte findet. Im implementierten System muss ein Oberon-Exceptionhandler explizit registriert werden. Zu diesem Zweck

```
try{
   Exception e = new IOExceptoin(‘‘message’’);
   throw e;
}catch( Exception e ){
   System.out.println( e );
}
```

Abbildung 4.8: try...catch() in Java

wurde die Funktion `InstallOberonExHandler(Module,Procedure,Handler,-ExceptionType,HandlesIt)` vorgesehen. Für das Auslösen einer Exception kann die Funktion `Throw(ExceptionObject)` verwendet werden. In dieser Funktion wird ein `HALT(0CAFEH)` aufgerufen. Der Parameter bleibt auf dem Stack und kann vom System dort abgeholt werden. In Abbildung 4.9 sind die unterschiedlichen Calling-Conventions für diese Handler aufgezeigt.

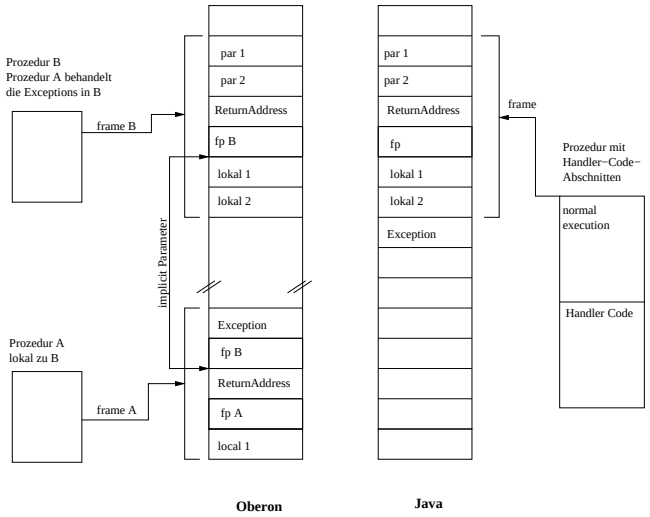

Abbildung 4.9: Calling-Convention für einen Handler, der als lokale Prozedur, respektive innerhalb einer anderen Prozedur definiert ist.

Für die Registrierung aller vorkommenden Exception-Handler verwendet das System einen balancierten binären Baum (*HandlerRegistry*). Das Ordnungskriterium in diesem Baum stellt der Programm-Bereich dar, für welchen der Handler definiert wurde (vergleiche 4.7).
Eine Exception wird durch einen Software-oder Hardware-Interrupt ausgelöst. Dadurch wird das Aos-Interrupthandling aufgerufen. Von dort wird die Kontrolle dem JVM-Exceptionhandler übergeben. In 4.10 sind die Aktionen dieser Exception-Handling-Routine beschrieben. In einem ersten Schritt wird geprüft, ob es sich um ein explizites `Throw()` einer Exception handelt (Software-Interrupt `0CAFEH`). In diesem Fall befindet sich das Exception-Objekt auf dem Userstack und kann von dort abgeholt werden. Anderenfalls muss der aufgetretene Interrupt übersetzt und ein geeignetes Exception-Objekt erzeugt werden. Falls kein Mapping vorgenommen werden konnte, wird das standardmässige Traphandling

```
PROCEDURE HandleException( VAR int: AosInterrupts.State;
                           VAR exc :AosInterrupts.ExceptionState;
                           VAR resume : BOOLEAN);
  IF exc.halt = 0CAFEH THEN top := TopOfStack();
  ELSE top := TranslateTrap( int.halt, resume );
       IF ~resume THEN RETURN END;
  END;
  REPEAT
    handler := FindHandler( HandlerRegistry, pc);
    IF handler # NIL THEN
       i := 0; found := FALSE;
       WHILE ~found & i < CatchTableSize DO
         found:=handler.catch[i].handlesIt(top)
               & (handler.catch[i].start <= pc)
               & (handler.catch[i].end > pc )
         INC(i)
       END
       IF found THEN
         PrepareResumption( int, top );
         resume := TRUE;
         RETURN
       END
    END
    pc := GetReturnOfFrame( fp );
    fp := GetPreviousFrame( fp )
  UNTIL StackInspected;
  resume := FALSE;
END HandleException;
```

Abbildung 4.10: Stack-Unwinding bei der Suche nach einem Handler

aufgerufen. Das heisst, der laufende Thread wird terminiert.
Konnte die Trapbehandlungsroutine ein gültiges Exception-Objekt in Empfang nehmen, wird der Stack zurückgespult. Dabei wird der Frame-Pointer (`fp`) als Rückwärtszeiger benutzt, um das vorangehende Stack-Frame zu finden. An der Adresse `fp-4` steht jeweils die Adresse an welcher das Programm nach Beendigung einer Prozedur fortgesetzt werden soll (`pc`). Nach jedem Unwinding Schritt (`fp := MEM[fp]`) wird in der HandlerRegistry nach einem Handler-Eintrag für den nächsten `pc` gesucht. Falls ein Handlereintrag vorhanden ist, muss noch geprüft werden, ob dieser Eintrag auch wirklich den entsprechenden Bereich abdeckt und ob die Bedingung für die Ausführung dieses Handlers erfüllt ist.
Dieser Vorgang wird solange durchgeführt, bis ein passender Handler gefunden wurde, oder der gesamte Stack zurückgespult worden ist. Wurde kein passender Handler gefunden, wird der laufende Thread terminiert. Anderenfalls muss alles für die Fortsetzung des Programmes im Exception-Handler vorbereitet werden. Dies ist mit der Funktion `PrepareResumption()` angetönt. Was sich hinter dieser Funktion verbirgt ist in den Abbildung 4.11 und 4.12 aufgeführt. Die Fortsetzung in einem Java-Exceptionhandler ist der einfachere Fall.

```
ref := Refs.FindRefForAdr( handler.module, pc );
locals := Refs.GetLocalSize( ref );
int.ESP := fp-locals-4;
MEM[int.ESP] := top;
int.EBP := fp;
int.EIP := handler.catch[i].handler
```

Abbildung 4.11: Code für die Fortsetzung des Programmes in einem Java-Exception-Handler

Es müssen lediglich die Werte `int.ESP` (Stack-Pointer), `int.EBP` (Frame-Pointer) und `int.EIP` korrekt gesetzt werden. Aschliessend muss das Exception-Objekt an der richtigen Stelle auf den Stack geschoben werden. Bei der Fortsetzung in einem Oberon-Exception-Handler ist die Situation komplizierter.

```
ref := Refs.FindRefForAdr( handler.module, pc );
locals := Refs.GetLocalSize( ref );
int.ESP := fp - locals;

(* push parameters for InvokeInner() *)
DEC( int.ESP, 4 ); MEM[int.ESP] := handler;
DEC( int.ESP, 4 ); MEM[int.ESP] := top;
DEC( int.ESP, 4 ); MEM[int.ESP] := fp;
DEC( int.ESP, 4 ); MEM[int.ESP] := i;

(* return adress of InvokeInner(); should never get there *)
DEC( int.ESP, 4 ); MEM[int.ESP] := 0; (* return adress of
int.EIP := InvokeInner();
int.EBP := int.ESP;
```

Abbildung 4.12: Code für die Vorbereitung des Aufrufs einer lokalen Prozedur

Die Oberon-Exception-Handlers sind als lokale Prozeduren definiert. Sie haben ein eigenes Stack-Frame und den Frame-Pointer der umschliessenden Prozedur als impliziten Parameter. Für die Fortsetzung des Programmes in einem Active-Oberon Exception-Handler wird zuerst vom Privileged-Level in den User-Level gewechselt. Im User-Level wird die innere Prozedur aufgerufen (`InvokeInner()`). Anschliessend wird der Stack inklusive das Stackframe der umschliessenden Prozedur abgeräumt. Bei diesem Vorgehen muss `InvokeInner()` als Fortsetzung der Exception-Handlingroutine im User-Level betrachtet werden. Es ist wichtig, dass dieser Wechsel in den User-Level vorgenommen wird. Im Privileged-Level sollten keine Traps vorkommen [2] und somit kein User-Code ausgeführt werden!

4.3.4 Diskussion

Die gewählte Implementierung benutzt Ideen aus [22] und bringt diese mit dem Java-Exception-Handling zusammen. Für das Java-Exception-Handling ist eine Registrierung der einzelnen Handler notwendig. Die Registrierung erfolgt jedoch nicht für jede Klasse oder jede Methode einzeln, sondern zentral im System. Für die Information über den Stackzustand werden die Metadaten der Aos-Module verwendet (`Module.refs`). Ein Exception-Handling ohne Registrierung scheint auch im Zusammenhang mit der Organisation der Aos-Module als fragwürdig. Beim Starten des Aos-Systems werden über hundert Module geladen und in einer linearen Liste verknüpft. Die JVM ihrerseits lädt wiederum Dutzende von Modulen. Für die Assoziation von `pc` zu einer Prozedur müsste diese Liste sequentiell durchsucht werden und das Behandeln einer Exception würde noch teurer. Das Argument, dass die Behandlung einer Exception beliebig teuer sein darf, verliert seine Berechtigung, wenn der Exception-Mechanismus mehr und mehr benutzt wird, um eine Applikation zu strukturieren und Exceptions bewusst in Kauf genommen werden.
Der Overhead für die Behandlung einer Exception könnte weiter reduziert werden, wenn mit einer speziellen Funktion der Exception-Handling-Mechanismus aufgerufen werden könnte, ohne einen Software-Interrupt auszulösen. Eine solche Funktion zur Verfügung zu stellen, böte keine grösseren Schwierigkeiten.
In [8] und [22] wird auf die Bedeutung von Zero Overhead beim Behandeln von Exceptions hingewiesen. Im Zusammenhang mit der Freigabe von Locks wurde diskutiert, ob das überhaupt möglich ist. Einerseits könnte der Garbage-Collector die gelockten Objekte wieder freigeben. Dadurch wird der Garbage-Collector langsamer. Es entsteht ein zusätzlicher Aufwand für alle Programme. Dieser Aufwand ist jedoch nicht direkt mit dem Exception-Handling assoziiert. Eine andere Möglichkeit bestünde darin, dass beim Locking und Unlocking die entsprechenden Objekte in das Stack-Frame der aktuellen Prozedur geschrieben würden. Beim Aufräumen des Stacks wäre es die Aufgabe des Exception-Handling Systems, diese Objekte wieder freizugeben. In dieser Variante müssen beim Locking und Unlocking zusätzliche Instruktionen im Zusammenhang mit dem Exception-Handling ausgeführt werden. Der Garbage-Collector muss jedoch nicht angepasst werden und alle anderen Programme sind nicht betroffen. Beide Varianten können nicht als Zero-Overhead Exception Handling bezeichnet werden. Im vorgestellten System ist diese Funktionalität noch nicht implemen-

[2] Bei Traps während der Behandlung eines Traps wird das Aos-System neu gebootet.

tiert.Im Augenblick wird unter Aos nicht genügend Information zur Verfügung gestellt, um diese Aufgabe in allen Fällen korrekt durchführen zu können.

4.4 Nebenläufigkeit in der Aos-JVM

In Aos gibt es die Abstraktionen *Active Object* und *Protected Object* (vergleiche 2.3). Bei der Implementierung der Nebenläufigkeit ging es darum, mit diesen Mitteln Java-Threads zu realisieren. Die Java-Threads müssen erzeugt und miteinander über Monitore synchronisiert werden können.
Die wichtigsten Unterschiede bezüglich des Verhaltens zwischen Java-Threads und Aos Active-Objects liegen beim Sperren (*locking*) von Objekten und beim Passivieren von Threads.
Ein Java-Thread kann ein Objekt mehrmals locken. Wenn ein Objekt von einem Thread n-mal gelockt wurde, muss auch die Unlock-Operation n-mal aufgerufen werden, um das Objekt wieder frei zu geben. Im Gegensatz dazu sind bei Aos mehrfache Locks nicht erlaubt.
In Java werden Threads dadurch passiviert, dass ein gelocktes Objekt ein `wait()` aufruft. Der Thread gibt den Lock auf dieses Objekt vorübergehend frei und wartet, bis er wieder explizit aktiviert wird. Die Aktivierung erfolgt durch den Aufruf von `notify()` desselben Objektes oder durch das Ablaufen eines Timers.
In Aos wird bei der Passivierung ein logischer Ausdruck mitgegeben. Das gelockte Objekt wird vorübergehend freigegeben. Bei jeder Unlock-Operation auf diesem Objekt sucht das System nach einem Active-Object, welches zuvor ein `AWAIT()` mit einem Lock auf dieses Objekt aufgerufen hatte. Falls ein solches Active-Object gefunden wird, wird die angegebene Bedingung reevaluiert. Falls das Ergebnis `TRUE` zurückgibt, wird der wartende Thread wieder aktiviert und erhält den Lock auf das Objekt zurück.

4.4.1 Java-Threads

Die Java-API sieht eine eigene Klasse für Threads vor (`java.lang.Threads`). Alle Threads implementieren das Interface `java.lang.Runnable`. Ein neuer Thread wird dadurch erzeugt, dass entweder eine Erweiterung von Threads gebildet wird und eine Instanz dieser Klasse erzeugt wird. Alternativ dazu kann eine Thread-Instanz mit einer Instanz eines `Runnable-Objektes` erzeugt werden. Das Starten des Threads erfolgt durch den Aufruf der Instanz-Methode `start()`.
Das Oberon-Framework, welches verwendet wird, um dieses Verhalten nachzubilden ist in Abbildung 4.13 aufgezeigt. Dabei liefert die Funktion `Td(obj)` das Typtag zu einem Objekt und die Funktion `Tag(Type)` das Typetag eines Typs. Es sei darauf hingewiesen, dass die Initialisierung von Runnable vom System beim Laden der Klasse automatisch vorgenommen wird.

4.4.2 Locks und Waits

In Aos können Protected-Objects gelockt werden. Jedes Protected-Object hat vordefinierte Felder, welche das System benötigt, um das Locking und Unlocking vornehmen zu können.

```
MODULE JThreads;
IMPORT AosActive, JI := JInterfaces;

TYPE Runnable = POINTER TO RECORD(VTable) (*java.lang.Runnable*)
         run : PROCEDURE(td:LONGINT;obj:Object);
     END;

TYPE Thread = OBJECT(Object)
        ro :Object;                      (*Pointer to object    *)
        ri :Runnable;                    (*Pointer to interface *)
        thread :AosActive.Thread;        (*used for services    *)
                                         (*of Java\trenn Threads*)
        PROCEDURE initA();               (*default constructor  *)
        BEGIN
          ro := SELF
          ri := JI.Lookup( Td(ro), Tag(Runnable) )
        END initA;
        PROCEDURE initB(ro : Object); (*constructor with runnable*)
        BEGIN
          SELF.ro := ro;
          ri := JI.Lookup(Td(ro),Tag(Runnable))
        END initB;
        PROCEDURE Start();
        BEGIN
          AosActive.CreateActivity(RunBody,priority,{},SELF)
        END Start;
        ..
    END Thread;

    PROCEDURE RunBody(td:LONGINT;obj:Object);
    VAR t:Thread
    BEGIN
      t:=obj(Thread);
      AosActive.SetExceptionHandler(
                  HandleException);(* install java Exception  *)
                                    (* Handler for this Thread *)
      BEGIN{EXCLUSIVE}
        t.thread:=AosActive.CurrentThread();
      END
      t.ri.run(Td(t.ro),t.ro)      (*invoke the runnable method*)
    END RunBody;

END JThreads.
```

Abbildung 4.13: Oberon-Implementierung von Java-Threads

Jedes Java-Objekt kann gelockt werden und muss demzufolge als Protected-Object alloziert werden. Da die Semantik von verschachtelten Locks in Aos nicht unterstützt wird, mussten zusätzliche Prozeduren definiert werden (`Object.lock()`, `Object.unlock()`). Ein Objekt, welches im Java-Subsystem gelockt wurde, erscheint im Aos-System ebenfalls als gelockt. Jedoch sind mehrfache Locks möglich.
Um einen aktiven Thread zu passivieren sind in Java die Methoden `wait()` und `wait(delay)` vorhanden. Wartende Threads werden mit `notify()` beziehungsweise `notifyAll()` wieder geweckt. Mit `notify()` wird ein beliebiger Thread, welcher auf dieses Object wartet, geweckt. Mit `notifyAll()` werden alle Threads, welche auf dieses Objekt warten, geweckt. Dieses Verhalten wurde mit dem Ticket-Algorithmus (siehe [1]) implementiert. Ein `wait()` zieht ein Ticket. Ein `notify()` löst genau ein Ticket ein und `notifyAll()` löst alle Tickets ein. Ein Thread wartet solange, bis sein Ticket an die Reihe kommt.
In Abbildung 4.14 sind die Prozeduren für das Locking und Unlocking von Java-Objekten sowie für die Umsetzung des Ticket-Algorithmus aufgeführt.
Es muss darauf hingewiesen werden, dass die Bedingung `(ticket-out)<0)` natürlich nur korrekt ist, wenn nicht mehr als 2^{15} Threads gleichzeitig dieses Objekt locken. Zudem muss das System sicherstellen, dass die wartenden Threads in First-In-First-Out-Ordnung bearbeitet werden. Anderenfalls wäre es durchaus möglich, dass ein wartender Thread nie mehr an die Reihe kommt. Für die Implementierung von `wait( delay )` ist ein Timer-Hilfsobjekt notwendig.

```
MODULE jjlObject;
TYPE Timer= OBJECT(AosActive.EventHanlder)
       PROCEDURE HandleTimeout();
       BEGIN{EXCLUSIVE} wakeup := TRUE; obj.signal();
       END HandleTimeout;

       PROCEDURE &start( obj : Object; delay : LONGINT );
       BEGIN{EXCLUSIVE} wakeup := FALSE;
         SELF.obj := obj; AosActive.SetTimeout( SELF, delay )
       END start;

       PROCEDURE stop();
       BEGIN AosActive.CancelTimeout()
       END stop;
     END Timer;

TYPE Object=OBJECT
     depth : LONGINT; in,out:INTEGER;
     PROCEDURE waitA();
     VAR ticket, depth : LONGINT;
     BEGIN ASSERT( AosActive.LockedByCurrent( SELF ) );
       ticket := in; INC(in); depth := SELF.depth;
       AWAIT( ticket - out < 0 );
       SELF.depth := depth
     END waitA;

     PROCEDURE waitB( ms : LONGINT );
     VAR ticket, depth : LONGINT; timer : Timer;
     BEGIN ASSERT( AosActive.LockedByCurrent( SELF ) );
       ticket:=in;INC(in); depth:=SELF.depth; NEW(timer,SELF,ms);
       AWAIT( timer.wakeup OR (ticket-out<0) );
       timer.stop(); SELF.depth := depth
     END waitB;

     PROCEDURE notify();
     BEGIN IF in#out THEN INC( out ) END
     END notify;

     PROCEDURE notifyAll();
     BEGIN out := in;
     END notifyAll;

     PROCEDURE lock();
     BEGIN
       IF AosActive.LockedByCurrent( SELF ) THEN INC( depth )
       ELSE AosActive.Lock( SELF, EXCLUSIVE ) END
     END Object;

     PROCEDURE unlock();
     BEGIN DEC( depth );
       IF depth = 0 THEN AosActive.Unlock( SELF, dummy ) END
     END unlock();

     PROCEDURE signal();(*this notifies all threads waiting on this object*)
     BEGIN{EXCLUSIVE} END signal;
  END Object;
END jjlObject.
```

Abbildung 4.14: Implementierung der Java-Locking-Semantik in Aos

4.5 Zusätzliche Implementierungshinweise

In diesem Abschnitt wird die Implementierung von weiteren Teilen der JVM besprochen. Einzeln betrachtet, handelt es sich dabei um kleinere Problemstellungen. In ihrer Gesamtheit bilden diese Programmstücke jedoch einen wesentlichen Teil der JVM und nahmen in dieser Arbeit auch entsprechend viel Raum ein.

4.5.1 Modul Refs

Eine wichtige Rolle für den Ladevorgang sowie für das Exception-Handling spielen die Methadaten eines Oberon-Moduls (`Module.refs`). Diese Metadaten beschreiben die Prozeduren mit ihren Parametern und lokalen Variablen. Sie ermöglichen, jederzeit den Stackzustand auszugeben. Aber auch ein Linken von Prozeduren ist mit der vorhandenen Information möglich. Das genaue Format dieser Metadaten ist in [27] beschrieben. Das Modul `Refs` stellt den Datentyp `Ref` zur Verfügung. In einer Variablen dieses Typs wird die Information einer Referenz (Prozedur innerhalb dieses Modules) akkumuliert. Zudem werden Funktionen zur Verfügung gestellt um eine Bestimmte Referenz zu finden (nach Prozedurnamen oder Adresse) und um über die gesamte Metainformation zu iterieren.
Eine Referenz wird sequentiell gesucht. Es kann jedoch bestimmt werden, bei welcher Position die Suche gestartet wird.

4.5.2 Der Ladevorgang

Für jeden Klassentyp existiert eine eigene Factory-Methode (vergleiche [11]) für die Erstellung dieser Klasse. Die Klassen des Typs `PrimitiveClass` werden beim Laden des Moduls JVM initialisiert. Klassen des Typs `ArrayClass` werden von der JVM erzeugt, sofern der Elementtyp bekannt ist. Die Erzeugung von `ArrayClass`-Typen erfolgt rekursiv.
Klassen des Typs `InterfaceClass`-oder `ObjectClass` müssen zuerst vom Class-File geladen werden. Die Zustände, die eine Referenzklasse einnehmen kann, sind in Abbildung 4.15 aufgezeichnet.
Im Folgenden wirden die einzelnen Zustände des Ladeprozesses bei Referenzklassen beschrieben.

Laden

Im Modul JLoader wird in einem ersten Schritt das Class-File gesucht. Dieses kann in einem Zip-Archive oder unverpackt im Filesystem sein. In einer Konfigurationsdatei (`JVM.properties`) werden die Zip-Archive definiert, welche durchsucht werden sollen. Falls keine entsprechende Datei gefunden wurde, wird eine Exception geworfen. Anderenfalls liest der Loader das Class-File, bis bekannt ist, ob es sich um eine Objekt-Klasse oder eine Interface-Klasse handelt (Zugriffsflags der Klasse; vergleiche 2.2.1). An dieser Stelle wird das eigentliche Klassen-Objekt erzeugt (Zustand: loading). Der Ladevorgang wird dem Typ entsprechend fortgesetzt. Der Zustand loading unterbricht die Rekursionszyklen, die beim Laden von Klassen vorhanden sind.

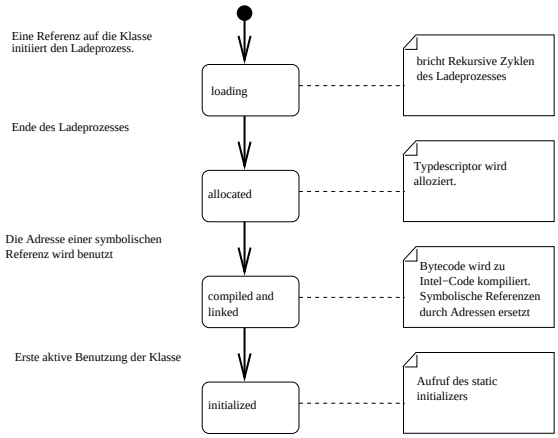

Abbildung 4.15: Zustandsdiagramm von Referenzklassen

Allozieren

Am Ende des Ladevorgangs (Zustand: allocated) ist ein Typdescriptor für die Klasse erstellt. Für Objekt-Klassen wird zudem ein eigenes Modul erzeugt. Sowohl für Objekt-Klassen als auch für Interface-Klassen muss die Verknüpfung zu einem allfällig vorhandenen Oberon-Modul und den entsprechenden Oberon-Typen hergestellt werden. Die Verknüpfung zu einem Modul geschieht über den Namen. Die Funktion `JTypes.MakeStubName()` bildet für einen Klassennamen einen Modulnamen. In einigen Fällen wird ein explizites Mapping von Klassenname zu Modulname durchgeführt. Für die meisten Fälle wird jedoch eine einfache Namenskonvention angewendet um aus dem Klassennamen einen Modulnamen zu bilden. Der Oberon-Typ in einem solchen Stub-Modul hat typischerweise denselben Namen wie die Java-Klasse, welche er beschreibt.
Natürich müssen auch die Funktionen und Variablen verknüpft werden. Die Objektvariablen werden in Active-Oberon wie auch vom Java-Class-Loader in derselben Reihenfolge alloziert, wie sie definiert werden. Bei statischen Variablen (globale Variablen in einem Modul) sortiert der Active-Oberon-Compiler die exportierten Variablen lexikographisch. Der Loader alloziert die Class-Variablen in der Reihenfolge, wie sie in Java deklariert sind. In beiden Fällen können sich Probleme ergeben. Werden in einem Oberon-Typ mehr Felder deklariert als im entsprechenden Java-Typ, so müssen diese Felder am Schluss der Felddeklaration stehen. Für die statischen Felder muss mit der Namensgebung dafür gesorgt werden, dass die Felder im Aos-Modul die richtige Reihenfolge haben! Natürlich ist das keine endgültige Lösung für das Problem.
Das Linken der Java-Methoden zu den Active-Oberon-Prozeduren erfolgt grundsätzlich über die Metadaten in der Referenz-Sektion der Module. In dieser Referenz-Sektion wird nach der Prozedur mit dem passenden Namen gesucht. Der Offset der gefundenen Referenz ist die Adresse der Prozedur relativ zum Codeabschnitt dieses Moduls.
Um nach einem Prozedurnamen suchen zu können, muss zuerst ein Mapping von Java-Prozedurnamen zu Acive-Oberon-Prozedurnamen vorgenommen werden. Einerseits können *Java-Identifier* andere Zeichen beinhalten als *Oberon-Identifier*. Zudem sind die Java-Methodennamen nicht eindeutig. Das Map-

ping von Java-Methodennamen zu Oberon-Prozedurnamen wird in der Prozedur `JTypes.MakeProcHashName()` vorgenommen. Dabei werden nicht erlaubte Zeichen innerhalb des Java-Namens ausgelassen. Aus der Java-Methodensignatur wird ein vierstelliger Hashcode gebildet, der an den Prozedurnamen angehängt wird und somit die Namen eindeutig macht. Falls es bei dieser Art der Namensbilung zu Kollisionen kommen sollte, müssten diese explizit aufgelöst werden. Solche Kollisionen sind jedoch unwahrscheinlich und werden mit Sicherheit vom Active-Oberon-Compiler entdeckt, bevor ein Stubmodul je benutzt werden kann. Beim Verknüpfen von Java-Methoden zu Active-Oberon-Prozeduren wird zwischen statischen und dynamischen Prozeduren unterschieden. Für die Anbindung der statischen Prozeduren muss lediglich die Adresse der Prozedur ermittelt werden. Für die dynamischen Methoden muss zusätzlich die Methodennummer innerhalb des Typdescriptors gesucht werden. Dazu wird die Methodentabelle sequentiell durchsucht und jeder Eintrag mit der Adresse der gesuchten Methode verglichen. Der Pseudocode in Abbildung 4.16 fasst diesen Vorgang zusammen.

```
method[] = {all Methods of Class A};
FOR i := 0 TO LEN(methods)-1 A DO
    method[i].duplicates := HasDuplicates( A, method[i] );
    name := MakeHashName( method[i] );
    ref :=  FindRefForName( StubModul, name );
    ASSERT( ref#NIL );
    method[i].stubadr := ref.offset + CodeBase( StubModul );
    IF ~method[i].isStatic THEN
        method[i].methodNumber := GetMethodNumber( A.TypeDescriptor,
                                                   method[i].stubadr );
    END
END;
```

Abbildung 4.16: Suche nach Adressen und Nummern der Prozeduren

Die Funktion `HasDuplicates()` ermittelt, ob andere Prozeduren mit demselben Namen innerhalb dieser Klassenhierarchie existieren. Die Funktion `MakeHashName` berechnet einen eindeutigen Prozedurnamen. `FindRefForName()` sucht im `StubModul` die Metadaten der Prozedur `name`. Die Funktion `CodeBase()` liefert die Speicheradresse des Beginns des Codeabschnittes von `StubModul` und die Funktion `GetMethodNumber()` durchsucht die Methodentabelle des Typdescriptors nach einem Eintrag mit dem Wert `stubadr`.

Kompilation

Eine Klasse muss dann kompiliert werden, wenn eine ihrer Funktionen referenziert wird, oder wenn die Klasse das erste Mal verwendet wird (vergleiche [23]). Bevor die einzelnen Methoden übersetzt werden, werden die symbolischen Referenzen aufgelöst. Anhand von Namen werden die Adressen von Methoden-, Feld- und Klassen-Objekten gesucht.
Gemäss der JVM-Spezifikation wird beim Aufruf einer Methode ein neues Frame aufgebaut. In diesem Frame sind die Parameter und lokalen Variablen diese Methode in einem Array festgehalten. Die Bytecode-Instruktionen greifen auf dieses Array zu. Dabei wird nicht zwischen dem Zugriff auf einen Parameter

oder eine lokale Variable unterschieden.
Um die Oberon-Calling-Convention zu erzwingen, muss eine Zuordung von lokalen Variablen der JVM zu Parametern und lokalen Variablen im Oberon-Stack-Frame vorgenommen werden. Anhand der Methodensignaturen wird berechnet, wieviele Prameter und lokale Variablen die entsprechende Methode hat. Mit dieser Information kann zu jedem Zeitpunkt bestimmt werden, welche Variable oder welcher Parameter gemeint ist, wenn auf eine JVM-Variable mit Index `i` zugegriffen werden soll. In Abbildung 4.17 ist diese Zuordnung beschrieben. Für

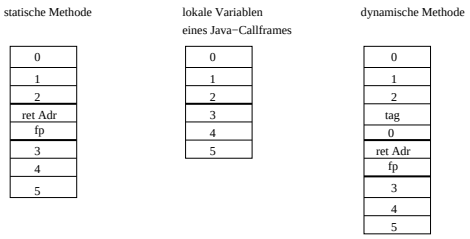

Abbildung 4.17: Zuordung von lokalen Variablen der JVM zu Parametern und lokalen Variablen des Stack-Frames

typgebundene Methoden muss der erste Prameter dupliziert werden. Zusätzlich wird in Active-Oberon der Typdesriptor als Parameter mitgegeben.
Wie der Compiler arbeitet ist in Abbildung 4.18 aufgezeichnet.

```
method[]:= {All Methods Of Class A}
FOR i := 0 TO LEN( method )-1 DO
    WriteRefSection( method[i] );
    WriteProlog( method[i].localVariables );
    pc := 0;
    WHILE pc < LEN(method[i].code) DO
       instruction := NextInstruction( pc, method[i].code );
       ApplyI386Pattern( instruction, A, method[i] )
    END;
    TranslateExceptionTable( method[i] );
    CreateModuleCodeSection()
END;
```

Abbildung 4.18: Übersetzungsschema des JVM-Compilers

Bei diesem Übersetzungsschema wird immer ein ganzes Modul aufs Mal übersetzt. Für jede Methode wird ein Eintrag in die Reference-Section des Moduls geschrieben (`WriteRefSection()`). Dabei werden Vereinfachungen vorgenommen. Die Parameter erscheinen als `P1...Pn`, die lokalen Variablen als `l1...ln`. Alle Einträge werden als `LONGINT` ausgewiesen. Für Datentypen, die grösser sind als vier Bytes werden zwei Einträge in den Referenz-Abschnitt geschrieben.
`WriteProlog()` erzeugt die Codesequenz für das Öffnen eines neuen Stack-Frames. Dabei handelt es sich mu ein Active-Oberon-Stackframe (vergleiche 2.3). Parameter müssen keine kopiert werden, da diese zu diesem Zeitpunkt bereits auf dem Stack sind.
Anschliessend wird der Bytecode Instruktion um Instruktion abgearbeitet und

Name	Funktion
invokeinterface	Aufruf von Interface-Methoden
multianewarray	Instanziierung eines mehrdimensionalen Arrays
fcmp?, dcmp?	Vergleiche auf Floatingpoint Daten
lshr, lshl, lushr	Shiftoperationen auf HUGEINT
lmul	Multiplikation von HUGEINT
ldiv, lrem	Division und Modulus-Operation auf HUGEINT
tableswitch	Analog zu CASE statement

Tabelle 4.1: Wichtive neue Codepatterns des JVM-Compilers

das entsprechende Code-Muster angewendet (`ApplyI386Pattern()`). Für die Generierung des Code-Musters ist Contextinformation notwendig. Es werden keine Optimierungen vorgenommen.
Am Ende jeder Übersetzung wird die Exception-Tabelle übersetzt. Die JVM-Instruktionsadressen werden durch die Intel-Instruktionsadressen ersetzt (`TranslateExceptionTable()`).
Der erzeugte Code wird nun in ein eigenes Code-Array copiert und dem Modul dieser Klasse angehängt (`CreateModuleCodeSection()`).
In Tabelle 4.1 sind die neuen Instruktionen aufgeführt, welche im Ramen dieser Arbeit erstellt worden sind. Für die Erstellung einiger Code-Muster wurde [12] zu Hilfe gezogen. Von den übrigen Instruktionen musste ein grosser Teil korrigiert werden!

Linking

Direkt anschliessend an die Übersetzung der Methoden, werden neuen Module gelinkt. Während der Übersetzung wurden sogenannte *Fixup-Ketten* aufgebaut

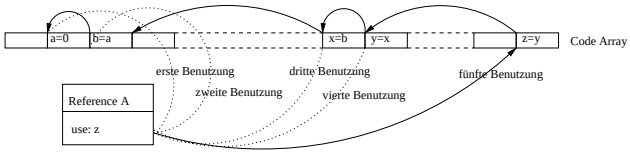

Abbildung 4.19: Fixup-Kette für Referenzen, die zur Kompilationszeit noch nicht bekannt sind

(vergleiche Abbildung 4.19). Wenn bei der Kompilation die Adresse einer Referenz noch nicht bekannt ist, so wird diese Stelle im Referenz-Objekt vermerkt und im Code die Adresse `NULL` eingefügt. Bei der zweiten Benutzung derselben Referenz wird an der neuen Stelle die Adresse der ersten Benutzung dieser Referenz eingetragen und das Referenz-Objekt speichert die Adresse der zweiten Benutzung. Auf diese Weise entsteht eine lineare Liste im Code-Array. In jedem Knoten dieser Liste muss die Adresse dieser Referenz eingetragen werden. Der Linker bearbeitet diese Fixup-Listen. Zudem werden Methodenadressen der typgebundenen Methoden in die Methodentabelle der Typdescriptoren geschrieben (nur falls kein Stub-Modul vorhanden). Statische Prozeduren in

den Stub-Modulen, welche die Implementierung des Java-Codes benutzen, werden mit der Instruktion `JMP JavaCodeAdr` überschrieben (`PatchMethods()`, `PatchStaticMethods()`).
Neben dem Linking von Methoden kopiert der Linker auch Daten in den Daten-Abschnitt der Module (`PatchConstants()`). Die Interfaces einer Klasse werden initialisiert und registriert (`InitInterfaces()`). Bei der Initialisierung der Interfaces wird für jedes Interface, welches von einer Klasse implementiert wird, ein Methoden-Rekord alloziert und die Felder werden mit den Adressen der entsprechenen Methode initialisiert (benutzt die Reflection-Eigenschaften der Klassen-Objekte).
Während des Linkens werden mit den Exception-Tabellen der Methoden Einträge für das Exception-Handling-System erzeugt und registriert (`Install-ExceptionTable()`).

4.5.3 Die Java-Console

Für die Ausgabe über *Standard-Out* wurde ein eigenes Fenster verwendet. Ein spezieller Thread (`Modul JConsole`) liest Daten aus einem zyklischen Buffer (`Modul ByteBuffer`) und schreibt diese auf die Ausgabe.
Eine eigene JVM-Console wurde notwendig, da das Log des Native-Oberon-Threads nicht von mehreren Threads gleichzeitig beschrieben werden kann. Es kommt unweigerlich zu einem Trap.

4.5.4 Die JVM-Systemaufrufe

In der Aos-JVM wurden einige Bytecode-Instruktionen als Prozeduraufrufe implementiert, da sie viele Laufzeitinformationen benötigen oder weil sie selber auf Systemaufrufen aufsetzen. In Tabelle 4.2 sind diese sogenannte *System-Calls* aufgelistet.

System-Call	**Funktion**
`New()`	Erzeugung einer neuen Objektinstanz
`NewArray()`	Erzeugung einer neuen Arrayinstanz; eindimesional; für Primitive Typen
`NewArrayA()`	Erzeugung einer neuen Arrayinstanz; eindimensional; für Referenz Typen
`MultianewArray()`	Erzeugung einer neuen Arrayinstanz; mehrdimensional
`Lookup()`	Interface-Lookup
`Lock()`	Locking eines Objektes
`Unlock()`	Unlocking eines Objektes
`CheckInitialized()`	Aufruf der `static initializers`
`ThrowException()`	Werfen einer Java-Exception
`CheckCast()`	Überprüfen eines Type-Casts
`InstanceOf()`	Analog zu `IS`

Tabelle 4.2: System-Calls der Aos-JVM

Der System-Call `New()` erzeugt eine neue Objektinstanz. Er benutzt direkt den Aos-System-Call `NewRec()`. Das neu erzeugte Objekt erhält zusätlich einen Zei-

ger auf das Klassen-Objekt.
Bei der Erzeugung einer neuen, eindimensionalen Arrayinstanz wird ein entsprechendes Arrayobjekt erzeugt. Jedes Arrayobjekt besitzt einen Zeiger auf ein eindimensionales Oberon-Array. Dieses Array wird in der entsprechenden Grösse alloziert. Mehrdimensionale Arrays sind als Arrays von Arrays definiert. Ihre Erzeugung ist rekursiv unter Verwendung der Prozeduren für die Erzeugung eindimensionaler Arrays implementiert.
Die Prozedur `Lookup` ist in Abschnitt 4.3 und die Prozeduren `Lock()` und `Unlock()` sind in Abschnitt 4.4 beschrieben.
`ThrowException()` löst lediglich einen Software-Trap mit der Nummer `0CAFEH` aus. Der Parameter, die Exception, welche geworfen werden soll befindet sich bereits auf dem Stack. Dieser System-Call könnte entfernt werden und direkt als Codemuster definiert werden.

```
PUSH 0CAFEH
INT 3
```

Allerdings wäre dann ein Ausführen des Exception-Handlings ohne Software-Interrupt nicht mehr möglich (vergleiche Abschnitt 4.3).
`CheckCast()` und `InstanceOf()` beinhalten nahezu dieselben Aktionen. `CheckCast()` verwendet `InstanceOf()`. Einerseits wird ein Typtest wie für Oberon-Typen durchgeführt. A ist genau dann eine Instanz von B wenn Folgendes gilt:

```
A IsInstanceOf B <=> (A.tagtable[B.ext] = B.tag)
```

Dabei ist `A.tagtable[]` die Tabelle im Typdesriptor von A, welche die `tags` der Basistypen beinhaltet. Ist B ein Interface so muss gelten:

```
A IsInstanceOf B <=> (JInterfaces.Lookup(B.tag, A.tag)#NIL)
```

Falls A und B Arraytypen sind, so werden ihre Elementtypen geprüft. Jedes Array implementiert implizit das Clonable Interface.

4.5.5 Anbindung der JVM an die Classpath-Java-API

In [21] werden die Schnittstellen zwischen der API und der JVM beschrieben. Gemäss dieser Schnittstelle muss die JVM einige Klassen der Java-API vollständig implementieren (vergleiche Tabelle 4.3) und einige Hilfsklassen zur Verfügung stellen (vergleiche 4.3).

JVM-Klassen	**Hilfsklassen**
java.lang.Class	java.lang.VMObject
java.lang.Runtime	java.lang.VMClassLoader
java.lang.Thread	java.lang.VMSystem
java.lang.Throwable	java.lang.VMSecurityManager
java.lang.reflect.Constructor	
java.lang.reflect.Method	
java.lang.reflect.Field	

Tabelle 4.3: Von der JVM zu implementierende Klassen

Für alle JVM-Klassen wurden Oberon-Stubs erzeugt. Eine Untersuchung der Funktionen der Hilfsklassen zeigte, dass diese lediglich die nativen Methoden für

die entsprechenden API-Klassen zur Verfügung stellen. Es wurde beschlossen, direkt die entsprechenden API-Klassen in Oberon zu implementieren und die Hilfsklassen nicht zu verwenden. Mit diesem Vorgehen konnte erreicht werden, dass eine eigene Library erzeugt werden konnte, welche kompatible Schnittstellen zur Classpath-Library hat.
Leider genügt es nicht die in Tabelle 4.3 aufgeführten Klassen zu implementieren. Innerhalb der übrigen Klassen wird viel nativer Code verwendet. In dieser Arbeit wurden aus diesem Grund die Hilfsklassen `oberon.OberonEnv` und `oberon.JLang` eingeführt, welche über statische Methoden eine Schnittstelle zum Oberon-Environment anbieten. Weitere solche Hilfsklassen werden nötig sein, um die Anbindung an die Classpath-Library weiter zu führen.
Die Entwicklung der Library wird dadurch erschwehrt, dass keinerlei Schichtstruktur innerhalb der Class-Path-Library vorhanden ist. Entweder läuft alles oder fast nichts. Durch die Anpassung einzelner Klassen aus der Library wurde versucht, die Situation etwas zu entschärfen.

Kapitel 5

Resultate

5.1 Erreichte Ziele

In dieser Arbeit wurde das vorhandene Gerüst einer Oberon-JVM weiterentwickelt und in Aos integriert. Im jetzigen Zeitpunkt ist diese JVM in der Lage, Javaprogramme auszuführen, welche nur beschränkten Gebrauch der Java-API machen. Die wesentlichen Sprachkonstrukte von Java werden vollständig unterstützt (Typerweiterung, Interfaces, Exceptions, Threads, sämtliche JVM-Instruktionen). Einfache Operationen auf Strings sowie einfache Ein- und Ausgabe über Streams sind implementiert.
Die vorgestellte JVM setzt direkt auf den Laufzeitstrukturen des Betriebssystems auf. Die Objekte der JVM besitzen dieselbe Struktur wie die Protected Objects in Aos. Die Speicherverwaltung und der Garbage-Collector von Aos können vollständig benutzt werden. Die Prozeduraufrufe für Java-Methoden sind kompatibel mit den Aos-Calling-Conventions. Die Aos-Threads werden benutzt, um die Java-Threads zu realisieren. Das Exception-Handling-System ist in die Aos-Interrupt-Behandlung integriert. In einigen Fällen wurden die Schnittstellen zum Betriebssystem geringfügig angepasst. Es wurde jedoch darauf geachtet, keine Konstrukte einzuführen, welche nur für diese JVM benötigt werden.
Um diese Ziele zu erreichen, wurde der Java-Interface-Mechanismus und ein Exception-Handling-Framework entworfen und implementiert. Zudem wurden die Java-Threads auf die Aos-Threads abgebildet und die Java-Semantik für die Synchronisation von Threads implementiert. Die Übersetzung des JVM-Instruktionssatzes in Intel-386-Code wurde vervollständigt und korrigiert. Die Schnittstellen zwischen Java-Klassen und Aos-Modulen mussten angepasst und korrigiert werden.
Um Funktionalitäten der JVM zu testen und zu demonstrieren musste eine Java-API vorhanden sein. Dazu wurde die Library [6] verwendet und angepasst. Teile der im Augenblick verfügbaren Funktionalitäten sind mit Active-Oberon entwickelt worden. Einige Funktionalitäten können von der Library unverändert übernommen und benutzt werden.
Es wurden zahlreiche Testbeispiele entwickelt, um die JVM zu testen. Aber auch Tests der Spec-Benchmark SPECjvm98 [28] (_200_check) konnten durchgeführt werden.

5.1.1 Performanz

Es wurden einige Tests bezüglich des Laufzeitverhaltens der Aos-JVM durchgeführt. Diese Tests erlauben keine abschliessende Beurteilung des Laufzeitverhaltens der Aos-JVM. Es ging in dieser Arbeit auch vorrangig darum, Java-Programme korrekt ausführen zu können.
Die vorliegenden Tests geben jedoch einen Hinweis auf die möglichen Stärken und Schwächen dieser JVM. Die vorliegenden Zahlen vergleichen die Ausführungsgeschwindigkeiten der Aos-JVM (**A**), der JVM von SUN (JDK1.2) [1] mit Just in Time Compiler (**B**) und ohne Just in Time Compiler (**C**). Die Tests wurden auf einem Compaq Armada 7730MT (166MHz, 32 MB RAM) ausgeführt. Die Werte in Klammern bezeichnen Messungen auf einer Linux-Java-Plattform mit einem anderen Rechner (166MHz, 48 MB RAM). Auf dieser Plattform konnte der Just in Time Compiler leere Prozeduraufrufe nicht wegoptimieren.

Interface-Performance

In diesem Test werden einfache Methoden via `invokeinterface` und `invokevirtual` aufgerufen. Die entsprechende Objektklasse implementiert lediglich ein einziges Interface! Die Aufrufe stehen in einer Schlaufe, welche 100'000-Mal ausgeführt wird.

Funktionalität	**A**	**B**	**C**
`invokevirtual` X()	36 ms	0/(11) ms	60/(77) ms
`invokeinterface` X()	57 ms	0/(47) ms	110/(107) ms
`invokevirtual` Y(X())	187 ms	60 ms	220 ms
`invokeinterface` Y(X())	348 ms	110 ms	330 ms

Ein analoges Testprogramm, welches in Active-Oberon ausgeführt wurde, benötigte 12 ms. Das ist die gleiche Grössenordnung wie bei der SUN-JVM mit JIT-Compiler (11 ms). Die Zeit von 36 ms für 100'000 Ausführungen von `invokevirtual` ist ein Hinweis darauf, dass der erzeugte Code für die Schleife wirklich schlecht ist. Im Fall, wo die Prozeduren eine einfache Rechnung ausführen, ist die SUN-JVM sogar ohne JIT-Compiler schneller!
Wenn man davon ausgeht, dass die Schleife langsam arbeitet, kann man annehmen, dass die Zeit für einen einfachen Interface-Lookup in der Aos-JVM nicht schlechter ist als bei der SUN-JVM mit JIT-Compiler.

Exception-Performance

Es werden Exceptions geworfen und wieder gefangen. Dabei müssen jeweils unterschiedlich viele Stackframes abgeräumt werden. Die Aufrufe stehen in einer Schlaufe, welche 10'000-Mal ausgeführt wird!

Funktionalität	**A**	**B**	**C**
Ein Stackframe entfernt	4103 ms	380 ms	440 ms
Zwei Stackframes entfernt	4141 ms	390 ms	500 ms

[1] Bei einem Test wurden vom JIT-Compiler die leeren Prozeduraufrufe wegoptimiert.

Das Exception-Handling ist fast 10-Mal langsamer als bei der SUN-JVM. Bei der Ausführung dieses Tests auf Aos gab es Anzeichen dafür, dass sich das System selber blockierte. Das Verhalten der Aos-JVM und des Aos-Kerns muss noch genau analysiert werden.

Producer-Consumer

Zwei Threads sind über einen zyklischen Puffer der Kapazität 256 miteinander synchronisiert. Der Consumer empfängt vom Producer 100'000 Items (ganze Zahlen). Die Wechsel zwischen den Threads scheinen in Aos effizient implementiert zu sein. Dieser Test konnte von der Aos-JVM schneller ausgeführt werden als bei den anderen JVMs.

Funktionalität	**A**	**B**	**C**
100'000 Zahlen übergeben	3297 ms	4560 ms	7080 ms

New-Object

In einer Schlaufe werden 500'000 neue Objekte auf dem Heap alloziert. Dabei muss der Garbage-Collector mehrmals aktiv werden.

Funktionalität	**A**	**B**	**C**
500'000 Objekte erzeugt	5760 ms	5930 ms	7530 ms

Bei den Zahlen für die Aos-JVM ist nicht sichergestellt, dass sie ganz genau sind. Während der Arbeit des Gargage-Collector wird der Timer-Interrupt ausgeschaltet und somit auch der Aos-Timer. Somit dürften die tatsächlichen Zeiten etwas länger sein. Die Ausführungszeit wäre demnach vergleichbar mit der Zeit von **C**. Es ist unwahrscheinlich, dass die Aos-JVM viel langsamer ist.

Zusammenfassung

Die Geschwindigkeit, mit welcher Java-Programme ausgeführt werden, war ein bisschen enttäuschend. Zwar lassen sich einige Unterschiede gut mit dem sehr einfachen Übersetzungsschema erklären. Die Unterschiede beim Methodenaufruf der leeren Methode zwischen dem Just in Time Compiler von SUN und der Aos-JVM sind sehr wahrscheinlich in der ineffizienten Implementierung der Schleife zu suchen. Ein identischer Methodenaufruf in Oberon benötigt gerade etwa gleich viel Zeit, wie beim Just in Time Compiler. Der Methodenaufruf von Oberon und der Aos-JVM ist jedoch identisch.
Die Zeiten für das Exception-Handling und für die Allozierung von neuen Objekten deuten darauf hin, dass Aos noch verbesserungswürdig ist! Allem Anschein nach blockiert sich das System zeitweise selber.
Producer-Consumer konnte als einziger Test in der Aos-JVM schneller ausgeführt werden als bei allen anderen JVM.
Man sollte sich jedoch von diesen Werten nicht abschrecken lassen. Der erzeugte Code kann deutlich besser gemacht werden. Aos ist ein neues System, welches

sicher auch noch Verbesserungen erfahren wird. Schliesslich wird auch die effiziente Implementierung einer API die Performanz der gesamten Java-Plattform entscheidend mitbeeinflussen.

5.1.2 Speicher

Die JVM selber benötigt ungefähr drei MB Speicher, um alle Module, Datenstrukturen sowie Java-Klassen zu laden. Das System Aos benötigt ca. 11 MB. Somit kann die Aos-JVM auf einem Rechner mit 16 MB ausgeführt werden.

5.2 Weitere Schritte

In dieser Arbeit wurde erreicht, dass *echte* Java-Programme von der Aos-JVM ausgeführt werden können. Zu einer vollständigen Java-Plattform ist jedoch nach wie vor ein weiter Weg!
Einerseits muss eine Java-API entwickelt werden. Will man den grossen Pool an Java-Software nutzen, so müssen die grössten Teile des JDK1.2 vorhanden sein. An sich kann die verwendete Library benutzt werden. Gewiss sind jedoch viele Anpassungen notwendig. In einigen Fällen wird man sich sogar die Frage stellen, ob man diesen Code nicht doch lieber selber schreiben will [2]!
Ein weiterer wesentlicher Teil der JVM, welcher bislang nicht berücksichtigt wurde, ist die Bytecode-Verifikation. Die Bytecode-Verifikation könnte dazu benutzt werden, eine Intermediate-Repräsentation aufzubauen. Anhand einer solchen Intermediate-Repräsentation könnte der erzeugte Code optimiert werden [14]. Spätestens wenn Class-Files über das Netz heruntergeladen werden, muss eine Bytecode-Verifikation gemacht werden!
Eine weitere Schwachstelle des aktuellen Systems sind die Schnittstellen zwischen Aos-Modulen und Class-Files. Zwar ist es möglich (und äusserst praktisch) Active-Oberon-Code und Java-Code nebeneinander in einer Klasse ausführen zu lassen. Die Art und Weise, wie das erreicht wird, ist jedoch umständlich und fehleranfällig (vergleiche 4.5.2). Es ist unschön, dass Stub-Module erzeugt werden müssen, nur um die Information über Klassenhierarchie zur Verfügung zu stellen. Unter Umständen könnte für ein Class-File ein Aos-Symbol-File erstellt werden, welches die Schnittstellen dieser Klasse wiederspiegelt.
Vermutlich möchte man irgendeinmal nicht nur Class-Files ausführen können, sondern auch solche erstellen. Sei es in der Sprache Active-Oberon oder Java.

5.3 Diskussion

Die in Abschnitt 1.3 formulierten Ziele konnten erreicht werden. Es wäre wünschenswert gewesen, mehr Tests mit der JVM98 Benchmark durchführen zu können. Die dazu benötigten Libraries stehen jedoch noch nicht zur Verfügung. Die Hauptursachen, dass nicht mehr Funktionalitäten implementiert werden konnten liegen darin, dass der bereits vorhandene Code nicht den Erwartungen entsprach und dass die verwendete Bibliothek kaum eine Schichtung erkennen

[2] Eine direkte Implementierung der Klasse `java.lang.String` in Oberon würde **viel** Speicher sparen und die Garbage-Collection beschleunigen!

lässt. Es können kaum einzelne Teile isoliert werden. Entweder läuft alles oder fast nichts.
Die Schnittstellen von Aos sind mächtig und flexibel. Im Verlauf dieser Arbeit wurden geringfügige Änderungen vorgenommen. Somit stellt Aos eine Art *Betriebssystem-Komponente* dar. Mit dieser Komponente lassen sich weitere Systeme realisieren. Im Laufe dieser Arbeit tauchte die Frage nach der organisatorischen Einheit eines *Subsystems* auf. Im Augenblick funktioniert die JVM als eine Gruppe von Threads, welche von einem *Main-Thread* kontrolliert werden. Alle diese Threads besitzen dieselbe Exception-Handling-Semantik und dieselbe Semantik der Instruktionen (Maskierungen der FPU-Exceptions, Rundungen). Jeder Aos-Thread besitzt jedoch sein eigenes Exception-Handling und initialisiert den Prozessor auf seine Art und Weise. Unter Umständen könnte ein Subsystem-Konstrukt zusätzliche Dienste für solche Situationen zur Verfügung stellen. Die Frage nach Schutzmechanismen für das System wäre in diesem Zusammenhang sicher auch interessant.
Die grösste Hürde auf dem Weg zu einer Java-Plattform für Aos stellt das Fehlen der Bibliotheken und deren immenser Umfang dar. Der Weg zu einer solchen Java-Plattform ist zwar noch weit. Das Ziel wäre jedoch attraktiv!

5.4 Dank

An dieser Stelle möchte ich mich bei allen bedanken, die mich bei dieser Arbeit unterstützt haben. Ganz spezieller Dank gehört P. Reali für seine vorbildliche Betreuung sowie P. Muller für seine spontanen Anpassungen am System und die geduldigen Antworten auf viele Fragen zu Aos. Nicht vergessen werden sollen auch F. Fiore und S. Antifakos, die viele Rechtschreibefehler korrigiert haben. Nicht zuletzt möchte ich mich auch bei meiner Frau und meinen Kindern bedanken, dass sie in dieser Zeit so viel Geduld und Verständnis aufgebracht haben. Sie werden froh sein, dass diese Arbeit jetzt ein Ende hat!

Literaturverzeichnis

[1] Andrews, G.R. *Concurrent Programming Principles and Practice.* Addison-Wesley, 1991.

[2] Banfi, G. Another way to run java programs. Master's thesis, ETH-Zürich, Institut für Computersysteme, 1997.

[3] Blackbox component builder. http://www.oberon.ch/prod/BlackBox/index.html.

[4] Brockschmidt, K. *Inside OLE.* Microsoft Press, second edition, 1995.

[5] Cierniak, M., Lueh, G-Y., and Stichnoth, J.M. Practicing judo: Java under dynamic optimizations. *Comm. ACM*, 2000.

[6] Gnu classpath. http://www.classpath.org/.

[7] Crawford, J.H. and Gelsinger, P. *Programming the 803086.* SYBEX, first edition, 1987.

[8] Drew, S., Gough, K.J., and Ledermann, J. Implementing zero-overhead exception handling. Technical report, Queensland University of Technology's School of Computing Science, 1995.

[9] Franz, M. The java virtual machine - a passing fad? *IEEE Software*, November 1998.

[10] Franz, M. and Kistler, T. Slim binaries. *Comm. ACM*, December 1997.

[11] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. *Design-Patterns Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1999.

[12] gcc, version 2.95.2, 24.10.1999.

[13] Gosling, J., Joy,B., and Steel,G. *The Java Language Specification.* Addison-Wesley, first edition, 1996.

[14] Griesemer, R. and Mitrovic, R.S. A compiler for the Java HotSpot virtual machine. In László Böszörményi, Jürg Gutknecht, and Gustav Pomberger, editors, *The School of Niklaus Wirth: The Art of Simplicity.* Morgan Kaufmann Publishers, 2000.

[15] Hoare, C.A.R. Monitors: an operating system structuring concept. *Comm. ACM*, 1974.

[16] Jbed, the next generation embedded and real-time operating system. http://www.esmertec.ch/p_jbed_jbed.html.

[17] Java 2 sdk, standard edition 1.2. http://www.javasoft.com/products/jdk/1.2/.

[18] Java 2 sdk, standard edition. http://www.javasoft.com/j2se/1.3/.

[19] Ibm research jikes compiler project. http://www.research.ibm.com/jikes/.

[20] Java native interface specification. http://java.sun.com/products//jdk/1.2/docs/guide/jni/index.html.

[21] Keiser, J. and Jones, C.B. Gnu classpath vm integration guide. http://www.gnu.org/software/classpath/doc/vmintegration.html.

[22] Kepler, J., Mössenböck, H., and Pirkelbauer, P. Zero-overhead exception handling using metaprogramming. Technical report, Johannes Kepler University Linz, Austria, 1997.

[23] Lindholm, T. and Yellin, F. *The Java Virtual Machine Specification.* Addison-Wesley, second edition, 1999.

[24] Muller, P. A Multiprocessor Kernel for Active Object-Based Systems. In *Proceedings of the Joint Modular Language Conference.* ETH Zürich, 2000.

[25] Ottmann, T. and Widmayer, P. *Algorithmen und Datenstrukturen.* Spektrum Akademischer Verlag, third edition, 1996.

[26] Reali, P. Persönliche Mitteilungen.

[27] Reali, P. *Native Oberon: Symbol and Object File Format*, 2000. http://www.oberon.ethz.ch/native/compiler/.

[28] Spec jvm98 benchmarks. http://www.spec.org/osg/jvm98/.

[29] Stärk, R.F. et al. Java and the the java virtual machine definition, verification, validation, 2000.

[30] Stärk, R.F. and Schmid, J. Java bytecode verification is not possible. Technical report, Computer Science Department ETH Zürich, Siemens Corporate Technology, February 2001.

[31] Szyperski, C. *Component Software.* Addison-Wesley, second edition, 1998.

[32] Trap information. http://www.oberon.ch/native/WebTraps.html.

[33] Unicode. http://www.unicode.org.

[34] Wirth, N. and Gutknecht, J. *Project Oberon The Design of an Operating System and Compiler.* Addison-Wesley, acmpress edition, 1992.