Universität Hamburg
Fachbereich Informatik
Datenbanken und Informationssysteme

STUDIENARBEIT

Portierung des Tycoon-Systems auf das Macintosh Betriebssystem

23. November 1995

Markus Breilmann
Matrikel-Nummer:
4346494

Betreuer:
Prof. Dr. Joachim W. Schmidt

Inhaltsverzeichnis

1. Einleitung und Zielsetzung **1**
1.1 Einleitung 1
1.2 Zielsetzung 2
1.3 Aufbau 3

2. Aufbau des Tycoon-Systems **5**
2.1 Das Tycoon Schichtenmodell 5
2.2 Portabilität 7

3. Die Funktionsweise der Tycoon-Maschine **11**
3.1 Namenskonventionen der Tycoon-Maschine 11
3.2 Die Tycoon-Maschine 12
3.2.1 Runtime-System 12
3.2.2 Tycoon Virtual Machine 12
3.2.3 Unterstützung von TL-Befehlen 13
3.3 Funktionelle Einheiten 13
3.3.1 Ein-/Ausgabe 15
3.3.2 C-Calls 17
3.3.3 Das abstrakte Objektspeicherprotokoll TSP 18
3.3.4 Threads und Coroutinen 18
3.3.5 Speicherverwaltung “Tycoon Virtual Machine” 19
3.3.6 Ausnahmebehandlung 20
3.4 Die Startphase 20

4. Portierung auf Apple Macintosh **23**
4.1 Das Apple Macintosh Betriebssystem: MacOS 23
4.2 Das Vorgehen bei der Portierung 24
4.3 Tycoon als Macintosh-Applikation : Eine Konsole für Textein-/-ausgabe . . . 25
4.4 Speicherverwaltung 29
4.5 Coroutinen-Implementation 31
4.5.1 Coroutinen: 68k-Macintosh 32
4.5.2 Coroutinen: PowerPC-Macintosh 34

4.6 Umgebungsvariablen 34
4.7 Socket-Implementation 36
4.8 Zugriffe auf das Dateisystem 36
4.9 Datei-Namen 37
4.10 Dynamisches Binden auf Apple Macintosh 37

5. Zusammenfassung und Ausblick **39**
5.1 Zusammenfassung und Bewertung der Portabilität 39
5.2 Ausblick 40

Literaturverzeichnis **41**

1. Einleitung und Zielsetzung

1.1 Einleitung

In allen Bereichen der Wirtschaft und somit auch in der Software-Branche ist eine Zunahme des Kostendruckes zu beobachten. Er bedingt die Forderung nach geringen Entwicklungszeiten und einfacher Wartbarkeit von Software-Produkten.

Dieser Forderung kann durch den Einsatz von Mehrzweckprogrammiersprachen begegnet werden. Sie machen den Einsatz mehrerer spezialisierter Sprachen in einer Softwarelösung überflüssig und reduzieren damit die Komplexität eines Systems. Die Folge sind leichtere Erweiterbarkeit und Wartung der Produkte und einfachere Anforderungsprofile an die beteiligten Entwickler. Die Wiederverwendbarkeit von Code ist von ebenso großer Bedeutung: viele Programmteile z.B. für Benutzerschnittstellen, Dateiverwaltung und ähnliches können in Folgeprojekten erneut eingesetzt werden und senken dadurch die Kosten.

Ein bedeutender Teil heutiger Software operiert mit großen Datenmengen. Dies führt zu einem steigenden Bedarf an Entwicklungswerkzeugen mit Datenbankausrichtung, die eine effiziente Software-Entwicklung vor allem für heterogene Systemumgebungen erlauben und den Programmierer in größerem Maße als bisher unterstützen. Diese Tatsache und die vorher genannten Punkte motivierten die Entwicklung sogenannter polymorpher persistenter Programmiersprachen. Sie integrieren verschiedene Aspekte bekannter Mehrzweckprogrammiersprachen mit solchen zur Datenverwaltung in Datenbanken. Sie lösen dadurch typische Probleme wie den "Impedance Mismatch" und schaffen eine für den Programmierer einheitliche semantische und syntaktische Programmierumgebung für die Erstellung datenintensiver Anwendungen [MMueS94]. Das Konzept der Polymorphie [CaWe85] ermöglicht generische Funktionsbibliotheken und damit die flexible Wiederverwendung von Programmcode. Die strikte Typisierung von Daten in solchen Systemen ist geeignet, die Korrektheit und Sicherheit von Software zu erhöhen.

Die Sprachumgebung Tycoon [MS92], [Matt93], [MMM93] ist ein Beispiel für eine polymorphe persistente Programmiersprache. Sie ist das Ergebnis von Forschungsprojekten des Arbeitsbereiches Datenbanken und Informationssysteme (DBIS) am FB Informatik der Universität Hamburg . In Programmiersystemen wie Tycoon hat sich eine Schichtenarchitektur als sinnvoll erwiesen. Augenfälliges Merkmal ist die Ausführung von Anweisungen eines kleinen syntaktischen Sprachkernes durch einen Interpreter. Neben dem ausführbaren Programmcode und dem eigentlichen Quelltext existiert eine Zwischenrepräsentation für Refle-

xion und damit Optimierungen, wie sie aus Datenbanksystemen bekannt sind. Ein wichtiger Aspekt des Tycoon-Systems ist die Eigenschaft der Selbstimplementation der Hochsprache: die Sprachprozessoren, Type-Checker usw. sind in Tycoon selbst realisiert.
Investitionen für Anschaffung und Wartung von Software und die Schulung von Personal stellen einen beachtlichen Kostenfaktor dar. Ein wirksamer Investitionsschutz für Softwaresysteme wird zum einen durch Verträglichkeit mit anderen Produkten, zum anderen durch Einsetzbarkeit solcher Systeme in unterschiedlichen Hardware- und Betriebssystemumgebungen ermöglicht. Die Verträglichkeit mit anderen Software-Produkten kann durch offene Programmiersysteme mit wohldefinierten Schnittstellen erreicht werden. Anpassungsfähigkeit ist vornehmlich bei portabler Software gegeben, deren Struktur eventuelle Änderungen antizipiert.
Die Forderung nach portabler Software ist unter Kostengesichtspunkten offensichtlich. Neben dem Argument sich schnell weiterentwickelnder Hard- und Betriebssystem-Software gewinnt darüber hinaus ein weiteres zunehmende Bedeutung: das Interesse an der computergestützten Verwaltung von Geschäftsprozessen. Der sinnvolle Einsatz von sogenannten "Workflow-Management-Systemen" erfordert nicht nur den Austausch von Daten über die Grenzen von Betriebssystemen, Netzwerken und Hardware hinweg, sondern auch den Transfer ganzer Prozesse, also von Programmen in Ausführung. Damit wird Portabiltität zu einem zentralen Bewertungskriterium von Software.
Das Tycoon-System stellt die Basistechnologie für "Workflow-Managment-Systeme" zur Verfügung und kommt der Forderung nach Portabilität durch seine interne Struktur entgegen: der weitaus größte Teil des Systems ist in Tycoon selbst implementiert, Gegenstand einer Portierung ist lediglich der Interpreter. Der Anwendungsprogrammierer wird dadurch völlig von Fragen der Portabilität entlastet. Bei der Entwicklung des Interpreters kamen zahlreiche Konzepte zum Einsatz, die dessen Portabilität ermöglichen oder verbessern.

1.2 Zielsetzung

Hauptziel der vorliegenden Arbeit ist die Portierung des Tycoon-Systems auf das Apple Macintosh Betriebssystem. Gegenstand der Portierung sind zum einen die Tycoon-Maschine (Interpreter) [1], zum anderen der DBIS-eigene Einzelbenutzerobjektspeicher "Tycoon Memory" (TYMEM) [2].
Diese Arbeit soll die Eignung der verwendeten Konzepte für Portabilität untersuchen. Dabei stellt die Portierung einer Unix-Applikation dieser Komplexität auf das Macintosh Betriebssystem eine besodere Herausforderung dar, da in zentralen Bereichen starke Unterschiede in den Betriebssystemen vorliegen. Es wird zum einen die Schichtenarchitektur des Tycoon-Systems, zum anderen der modulare Aufbau der Tycoon-Maschine dargestellt. Die Ausführ-

[1] Das Tycoon Laufzeitsystem wurde zum überwiegenden Teil von Bernd Mathiske, DBIS, FB Informatik, entwickelt und implementiert. Der Byte-Code-Interpreter (Modul tmexec) wurde hauptsächlich durch Andreas Gawecki, DBIS, konzipiert. Die Entwicklung fand auf Sun Sparc Rechnern unter den Betriebssystemen SunOS und Solaris statt.
[2] Entwickelt von Bernd Mathiske, DBIS.

lichkeit soll dabei einen Einblick in das Zusammenspiel der Komponenten, den Daten- und Kontrollfluß ermöglichen. Auf dieser Grundlage kann ein Vergleich mit dem Verhalten des Apple Betriebssystems angestellt werden, aus dem sich notwendige Anpassungen direkt ergeben.
Neben der Funktionalität des "Tycoon-Top-Level", der Terme der "Tycoon Language" (TL) evaluiert, sollen auch die Funktionen der Tycoon-Standardbibliotheken zur Verfügung gestellt werden. Einige Tycoon-Funktionen sind durch Einbinden von C-Funktionsaufrufen realisiert. Diese C-Funktionen befinden sich in speziellen C-Modulen der jeweiligen Standardbibliothek; sie müssen auf ihre Lauffähigkeit überprüft und gegebenenfalls angepaßt werden.
Die Weiterentwicklung der Apple Hardware hat neben den bekannten Geräten mit Motorola 68xxx-Prozessoren zu einer neuen Generation von Rechnern geführt. Sie verwenden die sogenannten PowerPC-Prozessoren auf RISC-Basis (Reduced Instruction Set Computer). Für diese neuen Geräte wurden Änderungen des Apple Betriebssystemes und der Entwicklungsumgebungen notwendig, die sich auf die Portierung auswirken. So existieren unterschiedliche Compiler und es gibt eine geänderte Runtime-Architektur für PowerPC-basierte Apple-Modelle. Die neuen Compiler orientieren sich stärker am ANSI-C Standard, was in Einzelfällen zu unterschiedlichen Implementationen für die beiden Apple-Plattformen führen kann.
Die Kommunikation zwischen Tycoon und Benutzer wird auf Unix-Systemen über zeichenorientierte Terminals abgewickelt. Das Apple Betriebssystem kennt keine solchen Konsolen und basiert auf einer rein graphischen Benutzeroberfläche. Dieser Umstand erfordert bei der Apple-Portierung eine spezielle Lösung, die die benötigte Funktionalität nachbildet. Weitere gravierende Unterschiede bestehen im Dateisystem, so daß auch hier Anpassungen notwendig werden.
Die Kommunikation mit anderen Systemen ist in der Unix-Welt unter anderem über RPC (Remote Procedure Call) oder Sockets (zeichenorientierte Datenströme) möglich. Beide Konzepte stehen unter zahlreichen Unix-Derivaten zur Verfügung und sind dann nahtlos in das Betriebssystem integriert. Sie werden im Gegensatz dazu durch das MacOS nicht unterstützt. Eine Public-Domain-Lösung für Socket-Kommunikation bietet für Apple-Rechner einen Großteil der bekannten Funktionen. Sie wird in dieser Arbeit auf ihre Eignung geprüft und für den Datenaustausch mit anderen Rechnersystemen eingesetzt.

1.3 Aufbau

Der modulare Aufbau des Tycoon-Systems bedingt, daß lediglich an wenigen Teilen Änderungen notwendig wurden. Zum besseren Verständnis der Funktionsweise und des Zusammenspiels aller Komponenten werden aber auch solche vorgestellt, die nicht direkt von Anpassungen betroffen waren.
Das folgende Kapitel gibt einen Überblick über den Aufbau des Tycoon-Systems.
Im Abschnitt 2.2 werden mögliche Problembereiche bei der Portierung und adäquate Programmiertechniken vorgestellt.

Der Aufbau der "Tycoon Virtual Machine" und das Zusammenspiel ihrer Komponenten ist Gegenstand des Kapitels 3.
Schließlich werden im Kapitel 4 die Besonderheiten des Apple Macintosh Betriebssystems und notwendige Anpassungen and der "Tycoon Virtual Machine" dargelegt. Dieses Kapitel beschreibt den implementatorischen Beitrag dieser Arbeit zum Tycoon-System.

2. Aufbau des Tycoon-Systems

Die Entwicklung des Tycoon-Systems verfolgt eine Reihe von Zielsetzungen. Die Sprachumgebung soll die effiziente Programmierung datenintensiver Anwendungen, die Korrektheit von Programmen und geringen Wartungs- und Erweiterungsaufwand gewährleisten. Darüber hinaus sollen vorhandene Software-Ressourcen wie Bibliotheken und externe Dienste genutzt werden können. Die Sprache soll dennoch leicht zu implementieren und portabel sein.
Tycoon verbindet Konzepte verschiedener Programmierparadigmen: es werden imperative, funktionale und objektorientierte Programmiertechniken unterstützt. Das System ist modular aufgebaut, verschiedene Arten der Polymorphie erlauben die Erstellung generischer Bibliotheken. Die Sprache ist strikt typisiert; es können sowohl statische als auch dynamische Typüberprüfungen durchgeführt werden. Eine quasiparallele Ausführung von Programmen innerhalb eines Betriebssystemprozesses wird durch Multithreading ermöglicht. Funktionen, Typen und Threads sind Sprachobjekte erster Klasse und können daher z.B. selbst als Funktionsargument auftreten. Alle Sprachobjekte wie Werte (Daten), Programmcode und Threads werden persistent in einem Objektspeicher gehalten.
Die zum Teil konkurrierenden Zielsetzungen und Lösungsansätze erzwingen eine Systemarchitektur, die geeignet ist, die Komplexität nachhaltig zu reduzieren. Eine solche Architektur unterteilt das Gesamtsystem in mehrere funktionale Schichten. Jede dieser Schichten ist für bestimmte Aufgabenbereiche optimiert. Das Tycoon Schichtenmodell wird in diesem Kapitel erläutert und die Aufgaben der Komponenten dargestellt.

2.1 Das Tycoon Schichtenmodell

Die oberste Schicht des Tycoon-Systems stellt die eigentliche Programmiersprache zur Verfügung und bildet somit die Schnittstelle zum Benutzer. Der grundlegende Ansatz der Schichtenmodellierung besteht darin, eine prinzipiell beliebige Hochsprache in den darunterliegenen Schichten auf Zwischenrepräsentationen abzubilden, die für bestimmte Aufgaben besonders geeignet sind. Derzeit sind zwei Hochsprachen implementiert: “Tycoon Language” (TL) und “Tycoon object oriented Language” (TooL). Sie können sowohl zur Applikationsentwicklung als auch zur Systemprogrammierung eingesetzt werden. Die TL-Sprachprozessoren selbst sind in TL realisiert. Der kleine Sprachkern kann dynamisch erweitert und durch Funktionen in Bibliotheken ergänzt werden. Es handet sich bei TL um einen

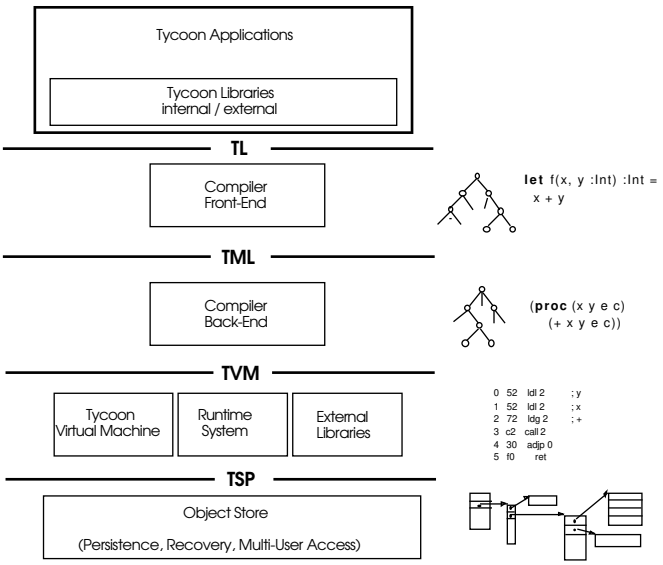

Abbildung 2.1: Das Tycoon Schichtenmodell

typisierten Lambdakalkül. In dieser Arbeit wird aus Gründen der Einfachheit lediglich auf die Sprache "Tycoon Language" Bezug genommen.

Das Compiler Front-End wandelt Terme der Sprache TL in die Zwischenrepräsentation "Tycoon Machine Language" (TML) um. TML ist eine nichtlineare Codierung im "Continuation Passing Style" (CPS). Ihre baumartige Struktur erlaubt Analysen und Optimierungen, wie sie aus Datenbanksystemen und anderen optimierenden Compilern bekannt sind. Der Sprachkern umfaßt lediglich wenige Instruktionen für Funktionsdefinition und -applikation, destruktive Zuweisungen, bedingte oder wiederholte Auswertungen, Ausnahmebehandlungen und den Aufruf von Funktionsimplementationen externer Programmiersprachen.

Der TML-Code wird durch das Compiler Back-End in einen persistenten, portablen Byte-Code (Tycoon Virtual Machine, TVM-Code) umgewandelt, dessen Instruktionen an den Befehlssatz von Transputern angelehnt sind. Der Code ist von konkreten Hardware-Plattformen entkoppelt und umfaßt Operationen auf elementaren Tycoon-Datenstrukturen.

Der Tycoon Byte-Code wird durch die "Tycoon Virtual Machine" (TVM) ausgeführt. Dieser virtuelle Computer bildet Operationen in Tycoon auf die jeweilige Hardware-Plattform ab.

Eine wichtige Rolle spielt das "Tycoon Store Protokoll" (TSP). Alle Datenstrukturen von atomaren Werten bis hin zu Programmcode und Threads werden in einem Objektspeicher gehalten. Dieser Objektspeicher kann annähernd frei gewählt werden; derzeit werden zwei Tycoon-eigene Stores, NapierStore und ObjectStore unterstützt. Das TSP bildet die Schnittstelle zwischen den Tycoon-Sprachen und diesen Objektpeichern: Operationen in höheren Schichten werden auf Objektkonstruktoren, -modifikatoren oder -destruktoren abgebildet. Dies entkoppelt die Sprache vollständig von konkreten Speichermodellen der zugrundelie-

genden Hardware-Plattform. Das Problem der Freispeicherverwaltung, wie es z.B. durch Fragmentierung des Heap in klassischen Sprachmodellen entsteht, wird damit in Tycoon vermieden. Der jeweilige Objektspeicher ist für eine effiziente Speicherverwaltung verantwortlich. Der Einsatz eines Objektspeichers ermöglicht die Persistenz aller Tycoon-Objekte. Mit Hilfe von Sicherungspunkten (commit) und Rücksetzmechanismen (rollback) kann für das gesamte System Datenbankfunktionalität realisiert werden.

2.2 Portabilität

Das Tycoon-Schichtenmodell reduziert deutlich die Komplexität des Gesamtsystems. Ein Großteil der Funktionalität ist in den oberen Schichten angesiedelt und Sprachprozessoren, Type-Checker und weitere Module sind in TL selbst implementiert. Abhängig von der zugrundeliegenden Hardware und des Betriebssystems ist lediglich die Tycoon Maschine, die den persistenten Byte-Code ausführt. Ihre Aufgaben sind der Datenaustausch mit dem Benutzer, Modifikation des Objektspeichers, Aufruf externer Funktionen und einiges mehr. Der Anpassungsaufwand bei einer Portierung des Systems beschränkt sich also auf Funktionen in diesem Laufzeitsystem.
In der Literatur werden mögliche Problembereiche bei der Portierung aufgeführt und Empfehlungen für den Entwurf portabler Applikationen gegeben. Die folgende Aufstellung nennt die Punkte, die für das Tycoon-System relevant sind:

- Programmiersprache:

  Sprachendialekt: Häufig implementieren Compiler eine Sprache in Bereichen unterschiedlich, in denen die Sprachdefinition Lücken oder Unklarheiten läßt. In einigen Fällen gibt es sogar Abweichungen von Sprachstandard. Bietet eine Sprachimplementation gegenüber dem Standard eine erweiterte Funktionalität, so sollte aus Gründen der Portabilität auf die Nutzung dieser zusätzlichen Funktionen verzichtet werden [Hen88], [TKB85].

  Abbildung von Datentypen auf Speichereinheiten: Datentypen können unabhängig von der zugrundeliegenden Hardware unterschiedlich implementiert sein. Ein Integer kann z.B. 16 oder 32 Bit umfassen [TKB85], [Wal82].

  Packen von Daten: Compiler können z.B. mehrere Bool'sche Variablen in einem Speicherwort ablegen. Der Datenaustausch mit anderen Plattformen wird dadurch unmöglich [Sch80], [TKB85].

- Fließkommazahlen:

  Zulässiger Wertebereich: Der Wertebereich für die Darstellung von Fließkommazahlen ist in hohem Maße von der zugrundeliegenden Hardware abhängig [TKB85].

  Genauigkeit der Darstellung: Die kleinste darstellbare positive Fließkommazahl (häufig als Epsilon bezeichnet) ist ebenfalls maschinenabhängig. Dies führt z.B. bei Iterationen zu unterschiedlichen oder unbrauchbaren Ergebnissen, indem eine

Reihe bis zu einem bestimmten Grenzwert konvergiert und danach divergiert. Eine unterschiedliche Genauigkeit bei der Berechnung des Grenzwertes führt dann zu abweichenden Ergebnissen [Sch80], [TKB85], [Wal82].

- Dateien:

  Dateinamen: Die zulässigen Zeichen, die maximale Länge und der Aufbau von Dateinamen ist auf verschiedenen Betriebssystemen sehr unterschiedlich. PC's unter DOS und Windows limitieren Dateinamen z.B. auf 8 Zeichen plus 3 Zeichen Namenserweiterung [Hen88], [TKB85].

  Verzeichnisse und zugehörige Operationen: Für den Aufbau von Dateisystemen existieren zahlreiche Ansätze und es werden jeweils spezialisierte Operationen für den Zugriff auf Dateien und Verzeichnisse benötigt [Bro77], [Hen88], [TKB85].

- interaktive Ein-/Ausgabe:

  Behandlung der Ein-/Ausgabeeinheit: In einigen Umgebungen wird die Ein- und Ausgabeeinheit wie eine Datei behandelt. Die zulässigen Operationen sind in solchen Fällen häufig eingeschränkt [Hen88], [TKB85].

  Zeilenende: Für die Markierung des Zeilenendes existieren vier Möglichkeiten, CR, LF, CR+LF und LF+CR. Alle Kombinationen sind auf verschiedenen Systemen im Einsatz [TKB85].

- Betriebssystem:

  Speicherverwaltung und -zuteilung: Manche Betriebssysteme weisen einem Programm bestimmte Speicherbereiche zu. In einigen Fällen werden Funktionen für die Allokation von Speicher durch das Betriebssystem zu Verfügung gestellt [TKB85].

  Umgebungsvariablen, Systemzeit und -datum: Die Verfügbarkeit von Systemzeit und -datum ist nicht auf allen Systemen sichergestellt und folgt zumindest unterschiedlichen Konzepten. Umgebungsvariablen existieren nur auf einigen Plattformen [Sch80], [TKB85].

- Hardwareplattform:

  Wortlänge: Die Wortlänge ist maschinenabhängig und beeinflußt u.a. die Darstellung von Integerwerten [Bro77], [Sch80], [TKB85].

  Addressierung: Die Speicheraddressierung kann byte- oder wortweise erfolgen. Die Reihenfolge von Low- und High-Byte ist ebenfalls maschinenabängig [Sch80], [TKB85].

Die 'Tycoon Virtual Machine' wurde in ANSI-C implementiert. Die Verwendung einer derart verbreiteten Standardsprache trägt in hohem Maße zur Portabilität des Systemes bei.

Ein weiterer entscheidender Punkt ist die konsequente Modularisierung auf unterschiedlichen Ebenen, auf die später im Detail eingegangen wird. Sind mögliche Problembereiche bekannt, vereinfacht dieses Vorgehen die Kapselung aller kritischen Funktionen und damit Anpassungen. Der Einsatz von C-Präprozessorkommandos gestattet das selektive Übersetzen von maschinenabhängigen Programmteilen.

3. Die Funktionsweise der Tycoon-Maschine

Tycoon-Programme werden in Form eines Byte-Codes (TVM-Code) persistent im Objektspeicher gehalten und durch die "Tycoon Virtual Machine" ausgeführt. Die virtuelle Maschine stellt zur Programmausführung im wesentlichen folgende Funktionen zur Verfügung:

- Die Datenein- und -ausgabe in Form von Datenströmen (streams) in Verbindung mit Bildschirm/Tastatur (Konsole), Dateien und Sockets.
- Einbindung von Funktionen externer Programmiersprachen (z.B. C oder C++).
- Kommunikation mit dem jeweiligen Objektspeicher.
- Die Unterstützung von Multithreading durch die Implementation von Coroutinen.
- Speicherzugriffsmechanismen, die sich aus dem Speicherbedarf der vorher genannten Funktionen ergeben.
- Eine rudimentäre Ausnahmebehandlung im Fehlerfall.

3.1 Namenskonventionen der Tycoon-Maschine

Die Tycoon-Maschine unterscheidet Funktionen auf drei Abstraktionsniveaus. Die Funktionen eines Niveaus bilden dabei eine logische Komponente: das Runtime-System, die eigentliche "Tycoon Virtual Machine" und eine Unterstützung systemnaher TL-Befehle.
Die TVM ist in der Sprache C realisiert. Der Source-Code ist in mehreren Modulen (Dateien) organisiert, wobei Funktionen eines Aufgabenbereiches in einem Modul zusammengefaßt werden (z.B. Funktionen zur Speicherverwaltung zum Modul "rtmemory"). Die ersten beiden Buchstaben des Dateinamens zeigen die Zuordnung der Datei zu einer der drei Komponenten der Tycoon-Maschine an: RT für Runtime-System, TM für "Tycoon Virtual Machine" und TL für Tycoon-nahe Funktionen (Tycoon Language). Abbildung 3.1 gibt einen Überblick über die genannten Komponenten.
Die Namen exportierter Entitäten eines Moduls wie Funktionen oder globale Variablen werden durch Kombination des Modulnamens (Dateinamens) und des eigentlichen Namens

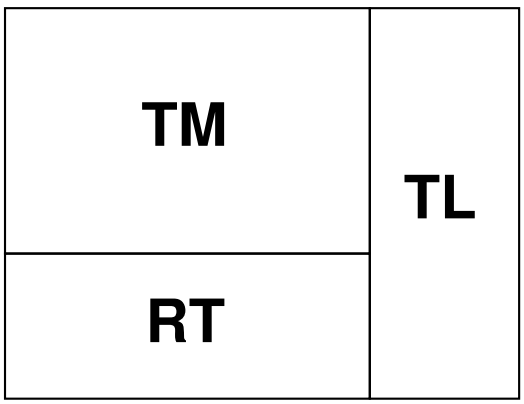

Abbildung 3.1: Die Komponenten der "Tycoon Virtual Machine"

der Entität gebildet (z.B. "rtmac_sample"). Funktionsnamen werden dabei klein ("tsp_openStore"), Typnamen groß geschrieben ("rtstring_T"). Makros werden gänzlich in Großbuchstaben notiert ("rtstring_LENGTH(s)").

3.2 Die Tycoon-Maschine

Die Tycoon-Maschine besteht aus drei Komponenten, dem Laufzeitsystem (RT, Runtime-System), der "Tycoon Virtual Machine" (TM) und der Unterstützung von Tycoon-Language-Funktionen (TL, siehe Abbildung 3.1).

3.2.1 Runtime-System

Das Runtime-System bildet die unterste Ebene des Systems und es stellt u.a. Funktionen für Textein- und -ausgabe, Dateizugriffe und Speicherverwaltung bereit (siehe Tabelle 3.1. Die in dieser Schicht zusammengefaßten Funktionen werden von annähernd jeder Applikation benötigt und sind am wahrscheinlichsten von Betriebssystemabhängigkeiten betroffen. Die Schnittstelle besteht daher aus abstrakten Funktionen, die im einfachsten Fall direkt auf Betriebssystemaufrufe oder Standardfunktionen abgebildet werden. Auf einigen Plattformen muß das gewünschte Verhalten nachgebildet werden. Derartige Änderungen bleiben durch die Kapselung auf eine Stelle im Gesamtsystem beschränkt und werden so vor übergeordneten Funktionen verborgen. Diese Komponente kann prinzipiell in jeder Art von Applikation verwendet werden, die unter dem Gesichtspunkt der Portabilität entwickelt wird.

3.2.2 Tycoon Virtual Machine

Zentrales Element der Tycoon-Maschine ist der Byte-Code-Interpreter. Darüber hinaus gibt es weitere Module, welche die Funktionalität der Byte-Codes implementieren. Das sind z.B.

rtconsol	Funktionen für Textausgabe auf der Konsole
rtcorout	Verwaltung von Coroutinen
rtdebug	Grundlegende Debugging-Unterstützung
rterror	Ausgabe von Fehlermeldungen auf STDERR
rtexcept	Ausnahmebehandlung
rtfile	Datei-Funktionen für Öffnen, Schließen, Ein- und Ausgabe
rtformat	Umwandlung von numerischen Werten in Zeichenketten für die Ausgabe
rthash	Unterstützung von Hash-Tabellen
rtmac	Macintosh-spezifische Funktionen und Ein- /Ausgabefenster
rtmemory	Speicherverwaltung
rtos	Betriebssystemabhängige Funktionen
rtring	Funktionen für Ringdatenstrukturen
rtscan	Umwandlung von Zeichenketten in numerische Werte
rtsess	Initialisierungen für eine Tycoon-Sitzung
rtsl	Sonderfunktionen für Zeichenkettenliterale
rtstd	Basisdatentypen- und Konstantendefinition
rtstream	Initialisierung von abstrakten Zeichenströmen
rtstring	Funktionen für das Kopieren, Vergleichen und Verketten von Zeichenketten

Tabelle 3.1: Die Dateien des Runtime-Systems

solche zur Transaktionsverwaltung, Multi-Threading, Store-Zugriffe und der Aufruf externer C-Funktionen (siehe Tabelle 3.2).

3.2.3 Unterstützung von TL-Befehlen

Die Sprache TL bietet einige Funktionen, die nicht direkt durch den TVM-Byte-Code abgedeckt werden können. Es handelt sich dabei z.B. um erweiterte Thread-Funktionalität, C-Callbacks, dynamisches Binden von TL-Funktionen usw. Die entsprechenden Funktionen sind daher in C realisiert und werden in TL eingebunden. Die Module für diese Erweiterung der Basisfunktionalität bilden die dritte Komponente der Tycoon Maschine (siehe Tabelle 3.3).

3.3 Funktionelle Einheiten

Häufig bilden Module aus mehreren Komponenten eine funktionelle Einheit. Die wichtigsten dieser Einheiten werden im folgenden vorgestellt und das Zusammenspiel der Module untereinander erläutert.

tm	Hauptroutine des Tycoon-Systems
tmaccess	Funktionen zum Registrieren von C-Funktionen
tmbyte	Definition der Tycoon Byte-Codes
tmccall	Funktionen für den Aufruf von C-Funktionen aus Tycoon
tmdata	Abbildung von Tycoon- auf C-Datentypen
tmdebug	Unterstützung für Tycoon-Debbuging auf C-Ebene
tmdiss	Dissassembler
tmexcept	Ausnahmebehandlung
tmexec	Hauptschleife des Byte-Code-Interpreters
tmgc	Hilfsfunktionen für die Freispeicherverwaltung (Garbage Collection)
tmhash	Funktionen für Tycoon Hash-Tabellen
tmlink	Funktionen für das Binden von C-Funktionen aus Tycoon
tmmeta	Werte für das dynamische Binden von Tycoon-Funktionen
tmprint	Formatierte Ausgabe von Tycoon Speicherobjekten
tmrecord	Implementation von Tycoon Tupeln (Records)
tmset	Implementation von Tycoon Mengen (Sets)
tmstatic	Registrierung statisch gebundener C-Funktionen
tmstore	Hilfsfunktionen für die Objektspeicherverwaltung
tmthread	Unterstützung von Tycoon-Threads
tmtrans	Transaktionsverwaltung

Tabelle 3.2: Die Dateien der Tycoon Virtual Machine

tlatom	Symbole für dynamisches Linken
tlcback	Aufruf von Tycoon aus C (C-Callbacks)
tlreader	Eingabe über Dateien (nicht blockierend)
tlreal	Fließkommaoperationen
tlsymbol	Unterstützung für TooL-Symbole
tlsunrpc	Funktionen für Sun-RPC
tlthread	Komplexere Thread-Funktionen
tlunsafe	Typunsichere Funktionen

Tabelle 3.3: Die Dateien für Tycoon-nahe Funktionen

3.3.1 Ein-/Ausgabe

Die Ein- und Ausgabe von Daten über Tastatur, Bildschirm, Dateien oder andere Rechner erfolgt einheitlich über sogenannte Streams. Das Konzept derartiger Daten- bzw. Zeichenströme wird bereits durch die Sprache C bereitgestellt: das Öffnen einer Datei z.B. liefert einen Stream, der anstelle des Dateinamens in weiteren Operationen auf dieser Datei verwendet wird. Ein Stream hat bestimmte Eigenschaften, die zur Zeit seiner Erzeugung festgelegt werden. Für eine Datei kann z.B. ein Datenstrom geöffnet werden, der nur Lesezugriffe zuläßt.

Aus technischer Sicht wird ein Stream in der Sprache C durch eine Datenstruktur realisiert, die unter anderem aus einem Datenpuffer und einem Zeiger in diesen Puffer besteht. Eine Leseoperation z.B. füllt diesen Puffer mit Daten aus der zugeordneten Datei und setzt den Zeiger auf die resultierende neue Position. In Funktionsaufrufen wird der Stream-Parameter in Form eines Zeigers auf die entsprechende Datenstruktur übergeben. Es existieren in C drei Standard-Streams, die nicht explizit geöffnet bzw. angelegt werden müssen [KR77]. Sie werden durch drei benannte Konstanten repräsentiert, die anstelle eines Zeigers übergeben werden können. Es sind dies "STDIN" oder der Wert 0 für Eingaben von der Konsole (Tastatur), "STDOUT" (1) für Ausgaben auf die Konsole (Bildschirm) und "STDERR" (2) für Fehlerausgaben.

In Tycoon wurde das Konzept der Streams verallgemeinert, um die Verwendung selbstentwickelter Stream-Typen zu ermöglichen. Ein Stream wird ebenfalls durch eine Datenstruktur realisiert (siehe Abbildung 3.2). Diese enthält neben dem Zeiger auf die Datenstruktur des C-Streams weitere Angaben, aus denen dessen Eigenschaften ersichtlich und somit über den Zeitpunkt seiner Generierung hinaus bekannt sind. Zusätzlich werden in diese Struktur Funktionszeiger auf Operationen eingetragen, die für den jeweiligen Stream zulässig sind.

Ein-/Ausgaben über die Konsole, Dateien und Fehlerausgaben können jeweils als Spezialfälle von Stream-Operationen angesehen werden. Dies führt zu einer Aufteilung der Funktionen auf vier Module: "rtstream" (Stream-Operationen), "rtfile" (Dateioperationen), "rtconsole" (Konsole) und "rterror" (Fehlerausgaben). Das Modul "rtstream" enthält die grundlegenden Funktionen für das Anlegen und Löschen eines Streams. Die übrigen Module stellen die jeweils zulässigen Funktionen für Ein-/Ausgabeoperationen zur Verfügung, die in der Tycoon-Datenstruktur eingetragen werden. Für die Konsole wird nur ein Tycoon-Stream geöffnet, der "STDIN" und "STDOUT" zusammenfaßt. Dies geschieht in der Startphase der Applikation, auf die später näher eingegangen wird.

Derzeit werden für alle Arten von Streams die zulässigen Ein-/Ausgabeoperationen über die Funktion "rtfile_initStream" in die Stream-Datenstruktur eingetragen, unabhängig davon, ob es sich um den Konsole-, Fehler- oder einen Datei-Stream handelt. Dies scheint für viele Betriebssysteme und insbesondere Unix vertretbar, da die Lese- und Schreiboperationen auf allen Stream-Typen gleiche Funktionalität bieten. Trifft dies für ein bestimmtes Betriebssystem nicht zu, können diese Initalisierungen über unterschiedliche Funktionen vorgenommen werden.

```
typedef struct rtstream_T {
  void *hDefault, *hOut, *hIn;
  void (*close)(struct rtstream_T *stream);
  void (*flush)(struct rtstream_T *stream);
  void (*clear)(struct rtstream_T *stream);
  Word (*write)(struct rtstream_T *stream, void *pBuffer, Word nBytes);
  void (*put)(struct rtstream_T *stream, Byte b);
  void (*putString)(struct rtstream_T *stream, String s);
  void (*putEndOfLine)(struct rtstream_T *stream);
  Word (*ready)(struct rtstream_T *stream);
  Word (*read)(struct rtstream_T *stream, void *pBuffer, Word nBytes);
  Bool (*get)(struct rtstream_T *stream, Byte *b);
  Bool (*getLine)(struct rtstream_T *stream, String sLine, Word nMaxChars);
  void (*setPosition)(struct rtstream_T *stream, Word iPosition);
  Word (*position)(struct rtstream_T *stream);
  Bool (*endOfStream)(struct rtstream_T *stream);
  Word (*nBytes)(struct rtstream_T *stream);
  Bool fAllocated, fWrite, fRead, fPosition, fLine;
} rtstream_TStruct, *rtstream_T;
```

Abbildung 3.2: Die Tycoon Stream-Datenstruktur

3.3.2 C-Calls

Ein beträchtlicher Anteil an Funktionalität in der Sprache Tycoon wird über den Aufruf externer C-Funktionen bereitgestellt [Matt93]. Der Aufruf solcher Funktionen bleibt für den Anwender völlig transparent: ein entsprechender Befehl in der Hochsprache TL bindet einen Tycoon-Bezeichner an eine C-Funktion. Der tatsächliche Aufruf einer C-Funktion wird über einen TVM-Code realisiert, der die notwendigen Umsetzungen auslöst.

Der Aufruf einer C-Funktion setzt sich aus drei Schritten zusammen: der Konvertierung der Parameter von Tycoon- in C-Format, dem eigentlichen Aufruf der Funktion und der Umwandlung des Ergebnisses in einen Tycoon-Datentyp. Die Funktionen für Typkonvertierungen sind im Modul "tmdata" zusammengefaßt. Die Liste der Tycoon-Parameter wird schrittweise abgearbeitet und das Ergebnis jeder Konvertierung in ein C-Array für maximal 20 Parameter eingetragen. Die C-Funktion wird dann über ihren Funktionszeiger aufgerufen. Die Parameterliste setzt sich aus den Feldern des genannten Arrays zusammen. Aufgrund der Aufrufkonventionen in C werden die Parameter derart auf dem Stack abgelegt, daß eventuell überflüssige Argumente die Ausführung nicht beeinträchtigen. Nach Rückkehr der Funktion wird das Ergebnis in ein Tycoon Datenformat gewandelt.

Eine besondere Technik wurde für die Ermittlung der Funktionszeiger gewählt, wobei statisch und dynamisch gebundene Funktionen unterstützt werden. In der Startphase der Tycoon-Applikation werden alle statisch gebundenen Funktionen in einer Hash-Tabelle erfaßt. Zum Zeitpunkt des C-Funktionsaufrufes wird diese Hash-Tabelle durchsucht und im Erfolgsfall der so ermittelte Zeiger verwendet. Scheitert die Suche, so wird über betriebssystemspezifische Operationen versucht, die Funktion dynamisch zu binden. Bleibt auch dies Vorgehen erfolglos, so wird die Ausführung des Tycoon-Programmes mit einer entsprechenden Fehlermeldung abgebrochen.

Die Funktionalität für C-Aufrufe ist auf die Module "tmccall", "tmaccess", "tmlink" und "tmstatic" verteilt. Die Datei "tmstatic" besteht lediglich aus der Funktion "tmaccess_begin". Sie enthält für jede statisch zu bindende Funktion eine Makro-Anweisung, die zu einer Registrierung in der Hash-Tabelle expandiert wird. Der Hash-Algorithmus verwendet als Schlüssel eine Kombination aus dem Funktionsnamen und dessen Länge. Der Aufbau der Tabelle läuft in zwei Schritten ab. Zunächst wird die Länge jedes Funktionsnamens ermittelt und so die Gesamtgröße des Speicherbereiches bestimmt, in dem die Schlüssel der Tabelle abgelegt werden. Im zweiten Durchlauf werden dann die Schlüssel und Funktionszeiger tatsächlich in die Hash-Tabelle eingetragen.

Die Datei "tmlink" enthält betriebssystemabhängige Implementationen für den Bindevorgang, da sich die Funktionsweise des dynamischen Bindens auf den verschiedenen Plattformen sehr unterschiedlich darstellt. Die Umsetzung eines Tycoon-Bezeichners in einen Funktionsnamen und gegebenenfalls einen Bibliotheksnamen wird durch Funktionen in dem Modul "tmaccess" gewährleistet.

3.3.3 Das abstrakte Objektspeicherprotokoll TSP

Das "Tycoon Store Protocol" (TSP) ist eine weitgehend modellunabhängige Objektspeicherschnittstelle [Matt93]. Es definiert abstrakte Funktionen für das Anlegen, Lesen und Modifizieren von Objekten. Das Tycoon-System kann auf unterschiedlichen Objektspeichern operieren, da es lediglich über Funktionen des TSP auf den Objektspeicher zugreift. Die Funktionen des TSP müssen dazu auf solche des Objektspeichers abgebildet werden.
Der DBIS-eigene Objektspeicher (TYcoon MEMory) implementiert alle notwendigen Funktionen, ein Mehrbenutzerobjektspeicher (TYSIN) ist ebenfalls verfügbar. Die Definitionen der abstrakten Funktionen befinden sich in der Datei "tsp.h", im Falle der Verwendung eines Objektspeichers eines Drittanbieters muß lediglich ein zusätzliches Modul eingefügt werden.
Das TSP umfaßt insbesondere Mechanismen für die Unterstützung datenintensiver Anwendungen. Dies sind:

Automatische Freispeicherverwaltung Objekte können nicht explizit gelöscht werden, sondern sie werden durch eine Garbage Collection beseitigt, sobald sie nicht mehr erreichbar sind.

Persistenz Alle von einem persistenten Wurzelobjekt erreichbaren Objekte sind ebenfalls persistent.

Atomarität, Fehlererholung Es gibt Sicherungspunkte (Commit) und Rücksetzoperationen (Rollback), die atomare Zustandsübergänge gewährleisten.

Portabilität Das TSP ermöglicht den Austausch von Daten zwischen verschiedenen Hard- und Softwarearchitekturen. Der Objektspeicher bleibt für Wertkonvertierungen verantwortlich.

Synchronisation Das TSP bietet Synchronisationmechanismen, um Mehrbenutzerbetrieb zu ermöglichen.

Das TSP verwaltet Objekte beliebiger Größe, die Objektgröße wird zum Zeitpunkt der Objektgenerierung festgelegt. Es wird eine markierte Datenrepräsentation (tagged format) verwendet, um die Freispeicherverwaltung und den Datenaustausch zu erleichtern.
Das TSP bietet Funktionen für unterschiedliche Aufgaben: Ermittlung der Eigenschaften des darunterliegenden Objektspeichers, Operationen auf Objekten, Operationen für Objektspeicher, Transaktionsoperationen, Freispeicherverwaltung und solche für den Datenim- und export.

3.3.4 Threads und Coroutinen

Der Einsatz von Multi-Threading erlaubt eine quasiparallele Ausführung mehrerer Programme auf einem Prozessor, im Falle von Tycoon auf der gleichen virtuellen Maschine [MMS95]. Threads werden als "leichtgewichtige Prozesse" (lightweight processes) bezeichnet. Es existieren mehrere Threads innerhalb eines Betriebssystemprozesses, wobei in jedem Thread

```
typedef struct rtring_T {
  struct rtring_T *ringPrevious;
  struct rtring_T *ringNext;
} rtring_TStruct, *rtring_T;
```

Abbildung 3.3: Ring-Datenstruktur für die Verwaltung von Coroutinen

eigenständiger Programmcode ausgeführt wird. Die Threads eines Prozesses teilen sich Resourcen wie offene Dateien, Kommunikationskanäle oder den Objektspeicher [DBBB95].
Ein Thread stellt einen Punkt im Kontrollfluß eines Programmes dar. In Multi-Threading-Systemen gibt es daher zu einem Zeitpunkt mehrere Programmcodeteile in Ausführung;
Ein Thread im Tycoon-System ist eine Instanz eines abstraktenTycoon-Datentyps. Dieser umfaßt den auszuführenden Programmcode, den Kontext der ausführenden virtuellen Maschine wie "stack", "stack pointer", "instruction pointer", "frame pointer" usw. und Referenzen auf andere Objekte. Ein Thread und somit ein Tycoon-Programm in Ausführung kann angehalten, auf ein anderes Tycoon-System übertragen und dort zu einem beliebigen Zeitpunkt fortgesetzt werden.
Die grundlegenden Funktionen für die Verwaltung von Threads wie deren Erzeugung, Anhalten, Fortsetzen etc. sind im Modul "tmthread" untergebracht. Es gibt weitere Funktionen, die für die Erweiterung der Tycoon Language benötigt werden. Sie befinden sich im Modul "tlthread" und es sind zumeist Operationen für die koordinierte Ausführung von Threads.
Die Kontextinformationen eines Threads werden zur Ausführung in eine C-Datenstruktur übertragen. Das Multi-Threading wird durch Coroutinen [Mar80] implementiert: für jeden Thread wird eine Coroutine instanziiert, die einen Byte-Code-Interpreter ausführt. Dieser arbeitet den jeweils zugeordneten Thread ab. Die Ausführungsreihenfolge der Coroutinen wird über eine ringförmige Zeigerdatenstruktur verwaltet: jede Coroutine zeigt auf ihren Vorgänger und Nachfolger (siehe Abbildung 3.3). Ist lediglich der Tycoon Top-Level aktiv, der interaktive Befehlseingaben verarbeitet, so befindet sich lediglich eine Coroutine im Ring. Die Funktionen für Coroutinen befinden sich im Modul "rtcorout", die zur Verwaltung von Ringstrukturen im Modul "rtring". Die Abbildung 3.4 zeigt die Datenstruktur einer Tycoon-Coroutine.

3.3.5 Speicherverwaltung "Tycoon Virtual Machine"

Die Speicherverwaltung stellt einen potentiellen Problembereich bei der Portierung einer Applikation dar. Entsprechende Funktionen sind daher in der Datei "rtmemory" zusammengefaßt. Es existieren neben solchen für das Allozieren und Freigeben von Speicher auf dem Heap noch weitere für den Vergleich, das Ausdehnen und Kopieren von Speicherbereichen. Insbesondere gibt es Funktionen für die Speicherallokation, die im Fehlerfall eine Ausnahmebehandlung auslösen.
Sämtliche Module der "Tycoon Virtual Machine" setzen auf den Speicheroperationen in

```
typedef struct {
  rtring_T ringPrevious;
  rtring_T ringNext;
  jmp_buf context;
  void *stack;
  void (*execute)(void);
  void *rtexcept_tag;
  void *rtexcept_parameter;
  void *rtexcept_pTarget;
  Bool rtexcept_fUnhandled;
} rtcorout_TStruct, *rtcorout_T;
```

Abbildung 3.4: Coroutinen-Datenstruktur

"rtmemory" auf. Diese Kapselung führt zu einer zentralen Speicherverwaltung. In anderen Funktionsbereichen wird die implizite Allokation von Speicher z.B. durch Funktionen aus Standardbibliotheken wie "printf" weitgehend vermieden.

3.3.6 Ausnahmebehandlung

Die Module "rtexcept" und "tmexcept" stellen eine Reihe von Makros und Funktionen zur Verfügung, die eine rudimentäre Ausnahmebehandlung ermöglichen. Dazu wird ein Anweisungsblock, dessen Ausführung zu Fehlern führen kann, durch die Makros "tmexcept_TRY" und "tmexcept_END" eingeschlossen. In diesem Block kann mit dem Makro "tmexcept_WHEN" auf einen bestimmten, mit "tmexcept_ELSE" auf beliebige Fehler geprüft werden.
Die Makros bedienen sich der C-Anweisungen "setjmp" und "longjmp", um vor einem Programmabschnitt, in dem Fehler auftreten können, den Prozessorkontex zu sichern. Dieser kann bei Mißerfolg rekonstruiert und eine entsprechende Fehlerbehandlung vorgenommen werden. Insbesondere können Ausnahmepakete propagiert werden, die die Art der Ausnahme anzeigen.

3.4 Die Startphase

Die Tycoon-Hauptroutine "tm_main" befindet sich im Modul "tm". Die Abbildung 3.5 zeigt die Schritte bei deren Bearbeitung. Sie eröffnet durch den Aufruf der Funktion "rtsess_new" (siehe Abbildung 3.6) eine Tycoon-Sitzung. Diese Funktion nimmt die Initialsierung einer Vielzahl von Modulen vor. Bei den Initialisierungen handelt es sich meist um das Löschen oder Vorbelegen von Datenstrukturen. Die entsprechenden Operationen befinden sich jeweils in der Funktion <Modul>_begin eines Moduls.

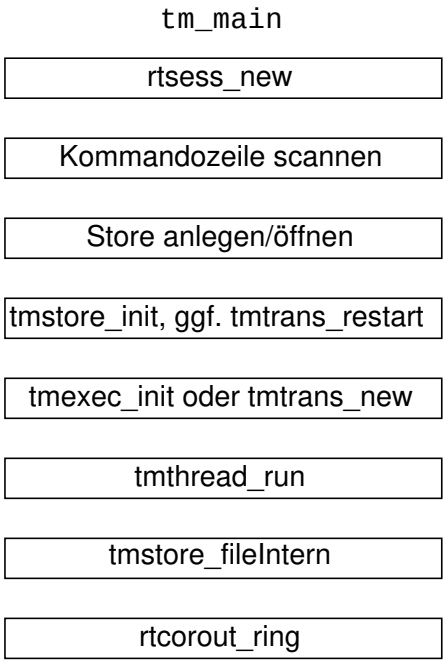

Abbildung 3.5: Die Startphase des Tycoon-Systems

Die Funktion "rtcorout_begin" erzeugt eine Coroutine "Super". Sie übernimmt die Freispeicherverwaltung (garbage collection) und die Kontrolle, sobald sich im späteren Verlauf des Programms keine Coroutinen mehr im Ring befinden. Sie löst ggf. die Beendigung der Tycoon-Sitzung aus.
Im Anschluß daran wird die Kommandozeile gescannt, um den Namen des zu verwendenden Objektspeichers zu bestimmen und festzustellen, ob dieser geöffnet oder neu angelegt werden soll. Nach dem Öffnen oder Anlegen des Objektspeichers wird die Store-Software initialisiert. Im Falle eines Wiederanlaufens eines bestehenden Stores wird der Zustand zum Zeitpunkt des letzten Kontrollpunktes durch die Funktion "tmtrans_restart" wiederhergestellt. Insbesondere werden alle aktiven Threads instanziiert. Für einen neuen Store wird neben dem Thread "Super" lediglich ein weiterer Thread gestartet und mit der Funktion "tmstore_fileIntern" das entsprechende Tycoon Boot-File in den Store geladen.
Schließlich wird die Funktion "rtcorout_ring" aufgerufen, damit die Kontrolle an die Threads im Ring und somit an Tycoon Byte-Code-Interpreter abgegeben; das Tycoon-System ist aktiv.

```
Bool rtsess_new(void)
{
  rtos_begin();
  if (!rtcorout_begin()) return FALSE;
  rtdebug_setAssertHandler(assertHandler);
  rtexcept_TRY
  {
    rterror_begin();
    rtconsol_begin();
    tsp_init();
    tmthread_begin();
    tmbyte_begin();
    tmaccess_begin();
#  ifdef tmstatic_STARVIEW
      svaccess_begin();
#  endif
    tmlink_begin();
    tmdebug_begin();
    tlcback_begin();
    tlsymbol_begin();
    toolrt_begin();
  }
  rtexcept_ELSE
    return FALSE;
  rtexcept_END;
  return TRUE;
}
```

Abbildung 3.6: Die Funktion rtsess_new

4. Portierung auf Apple Macintosh

Ziel der vorliegenden Arbeit ist die Portierung der Tycoon Maschine auf das Apple Betriebssystem. In diesem Kapitel werden die dafür notwendigen Anpassungen und Erweiterungen des bestehenden Systems und deren Ursachen erläutert; es stellt damit den eigentlichen Kern der Arbeit dar.

4.1 Das Apple Macintosh Betriebssystem: MacOS

Das Macintosh Betriebssystem kann als das erste kommerziell erfolgreiche Betriebssystem angesehen werden, das eine graphische Benutzeroberfläche einsetzt. Die Entwicklung reicht in die frühen 80er Jahre zurück. Damit verbunden sind einige veraltete und proprietäre Konzepte innerhalb der Systemarchitektur. Sie verursachen einen weit höheren Aufwand bei der Portierung als Punkte, die gewöhnlich in der Literatur als kritisch geschildert werden (wie z.B. Sprachdialekte oder Speicherwortgrößen). Diese Problemstellungen zeichneten sich bereits bei dem Entwurf des Tycoon-Systems ab, eine technisch zufriedenstellende Lösung konnte jedoch unter dem Sun Betriebssystem nicht vorweggenommen werden.
Die Apple Macintosh System-Software kann in drei Bereiche eingeteilt werden: das eigentliche Betriebssystem, die Macintosh "Toolbox" und zusätzliche System-Software. Entwicklern bietet diese Umgebung eine Fülle von Funktionen für die Programmierung der graphischen Benutzeroberfläche, den Einsatz Apple-eigener Werkzeuge aber auch zur Bewältigung systemnaher Aufgaben. Die Funktionen eines Aufgabenbereiches sind in sogenannten "Managern" zusammengefaßt. Die Funktionen zur Speicherwaltung ergeben z.B. den "Memory Manager". In einigen Ausnahmen wird synonym[1] auch der Ausdruck "Package" verwendet (z.B. "Standard File Package"). Es folgt ein Überblick über die wichtigsten Manager und deren Stellung in der System-Software:

- Betriebssystem

  Process Manager: Verwaltet das Starten, Koordinieren und Beenden von Applikationen.

  Memory Manager: Ermöglicht das dynamische Allozieren und Freigeben von Speicher innerhalb einer Applikationspartition.

[1] Die Implementation von "Packages" und "Managern" ist unterschiedlich; die Verwendung der Funktionen gestaltet sich jedoch für den Anwendungsprogrammierer in beiden Fällen gleich.

Virual Memory Manager: Verwaltet virtuellen Speicher (Auslagern von Daten aus dem RAM auf Festplatte).

File Manager: Erlaubt den Zugriff auf das Dateisystem wie Öffnen, Lesen, Schreiben und Schließen von Dateien.

Time Manager: Stellt Funktionen für die periodische Abarbeitung von Routinen zur Verfügung.

- Toolbox

QuickDraw: Übernimmt alle Bildschirmausgaben wie Graphik und Text.

Windows Manager: Erzeugung und Verwaltung von Fenstern

Scrap Manager: Ermöglicht das Ausschneiden und Einfügen von Daten unter Applikationen

Standard File Package: Stellt Standardauswahlfenster für die Dateiauswahl bereit.

Menu Manager: Erzeugt und verwaltet die Menüleiste einer Applikation.

Event Manager: Informiert eine Applikation über Benutzeraktivitäten und Programmausführung. Erlaubt die Kommunikation zwischen Applikationen.

TextEdit: Stellt einfache Text-Editor Funktionalität zur Verfügung.

- andere System-Software

QuickTime: Bietet Funktionen zum Abspielen von Ton- und Bildsequenzen.

Interapplication Communication: Regelt die Kommuniktion unter verschiedenen Applikationen.

Communication Toolbox: Funktionen für die Kommunikation über Modem etc.

World-Wide Support: Unterstützung für beliebige Schriftsysteme.

4.2 Das Vorgehen bei der Portierung

Für die Übersetzung der Tycoon Maschine und des DBIS-eigenen Objektspeichers wurde die Apple Entwicklungsumgebung "Macintosh Programmers Workshop" (MPW, Version ETO#16) eingesetzt. Sie unterstützt sowohl die Prozessorarchitektur Motorola 68k als auch die neue PowerPC Prozessorgeneration. Die Portierung erforderte eine Reihe von Anpassungen und zusätzlicher Funktionen. Der C-Präprozessor bietet die Möglichkeit, Programmcode in Abhängigkeit von Präprozessorvariablen zu übersetzen. Die Include-Datei "rtstd.h" enthält Anweisungen, um bei der Übersetzung der C-Quelltexte das Zielbetriebssystem zu ermitteln und entsprechende Schalter zu setzen. Im Falle des MacOS wird die Präprozessorvariable "rtstd_LIB_MPW" definiert. Zusätzlich wird bei der Übersetzung für die PowerPC-Architektur durch die MPW-Entwicklungsumgebung die Variable "powerc" definiert. Die Abbildung 4.1 zeigt ein Beispiel für den Einsatz des C-Präprozessors für bedingte Übersetzung.

```
#ifdef rtstd_LIB_MPW
# include <FCntl.h>
# include "rtmac.h"
#else
# include <sys/types.h>
# include <sys/time.h>
# include <sys/uio.h>
#endif
```

Abbildung 4.1: Der Einsatz von Präprozessorkommandos

Alle zusätzlichen Funktionen, die für die Macintosh-Applikation nötig wurden, sind im Modul "rtmac" zusammengefaßt.

4.3 Tycoon als Macintosh-Applikation : Eine Konsole für Textein-/-ausgabe

Die Ein-/Ausgabe von Text erfolgt unter X-Windows (Unix) über eine sogenannte Konsole: die Streams "STDIN", "STDOUT" und "STDERR" sind dazu mit einem Fenster bzw. der Tastatur verbunden. Eingaben über die Tastatur werden in das Fenster geleitet und können über den Stream "STDIN" gelesen werden. Ausgaben über die Streams "STDOUT" und "STDERR" werden ebenfalls in dem genannten Fenster dargestellt. Jedes Unix-Programm kann diese Funktionalität automatisch nutzen, ohne daß zusätzlicher Programmieraufwand nötig wird.
Bei der Portierung des Tycoon-Systems auf das Apple Macintosh Betriebssystem stößt man in diesem Zusammenhang auf mehrere Probleme. X-Windows ist eine Ergänzung des Unix-Betriebssystems, um dem Anwender eine graphische Benutzeroberfläche zu bieten. Sowohl X-Windows als auch MacOS verfolgen einen ereignisorientierten Ansatz bei der Kommunikation mit dem Benutzer: eine Aktion des Benutzers löst eine Reaktion der graphischen Oberfläche bzw. der Applikation aus. Ein großer Unterschied liegt jedoch in der Instanz, die derartige Ereignisse verarbeitet. X-Windows stellt nicht nur die Konsole-Funktionalität zur Verfügung, sondern übernimmt für die Applikation die Behandlung aller Ereignisse wie Tastatureingaben, Verschieben des Fensters oder des sichtbaren Textes innerhalb des Fensters. Damit bleibt der Einsatz einer graphischen Benutzeroberfläche für die Applikation und somit den Programmierer transparent, der Datenaustausch mit dem Benutzer erfolgt über die Standardschnittstelle Streams. Diese Schnittstelle wird von Tycoon intensiv genutzt.
Unter dem Apple Macintosh Betriebssystem muß das Anwendungsprogramm selbst mögliche Ereignisse erkennen und entsprechende Reaktionen auslösen. Da zu diesen Ereignissen auch Eingaben über die Tastatur zählen, fehlt die Funktionalität der Streams STDIN, STDOUT und STDERR, wie sie von Unix bekannt ist. Die Verarbeitung der Eingabe muß vollständig durch die jeweilige Applikation gehandhabt werden, wozu auch die Ausgabe auf dem Bild-

schirm bzw. in ein Fenster zählt.
Aus diesem Umstand ergibt sich für die Portierung, daß zum einen Funktionen in das Tycoon-System integriert werden müssen, die Ereignisse der Benutzeroberfläche verarbeiten, zum anderen muß das von Unix bekannte Stream-Verhalten nachgebildet werden.
Die MPW Entwicklungsumgebung umfaßt die sogenannte SIOW-Bibliothek ("Simple Input/Output Window"), die zu einer beliebigen Applikation gebunden werden kann. Diese Bibliothek erzeugt, wie der Name vermuten läßt, ein einfaches Texteditorfenster und übernimmt die Behandlung der damit verbundenen Ereignisse. Sie bietet aber nicht die notwendige Funktionalität bezüglich der Textein-/-ausgabe-Streams. Darüber hinaus entspricht das Verhalten eines Texteditors nicht exakt dem einer Konsole, da Text an beliebigen Stellen eingefügt werden kann. Diese Punkte machen den Einsatz einer eigenständigen Lösung notwendig.
Für die Verwaltung der Benutzeroberfläche wurde auf der Grundlage einer Apple Beispielapplikation ein eingeschränkter Texteditor realisiert, dessen Fähigkeiten an die einer Unix-Konsole angepaßt wurden:

- Eingaben sind lediglich an der Einfügemarke in der untersten Zeile möglich.
- Texte an einer beliebigen Stelle des Textfensters können in die Zwischenablage kopiert werden.
- Texte aus der Zwischenablage können an der Stelle der Einfügemarke eingesetzt werden.
- Eingaben werden mit der Taste "Return" oder "Enter" abgeschlossen/bestätigt.

Dieser Texteditor wird im folgenden als "Tycoon-Konsole" bezeichnet.
Das Kernstück einer typische Apple Applikation bildet die sogenannte "Event-Loop". Dies ist eine Endlosschleife, in der alle erforderlichen Ereignisse wie Benutzereingaben usw. erkannt und entsprechende Funktionen für deren Bearbeitung aufgerufen werden. Dieser Ansatz läuft dem des Tycoon-Systems zuwider, bei dem der Byte-Code-Interpreter ebenfalls durch eine Endlosschleife realisiert ist. Es stoßen also zwei konkurrierende Sichten aufeinander, bei denen einmal der Benutzerinteraktion, im anderen Fall der Berechnung mehr Bedeutung eingeräumt wird.
Bei der Portierung wurde der Tycoon-Ansatz gewählt, um eine Kopplung zu ermöglichen: die Ereignisschleife ("Event-Loop") wurde zu einer Funktion aufgelöst, die bei jedem Aufruf lediglich einmal durchlaufen wird. Diese Funktion heißt "rtmac_sample" und wird aus der Tycoon-Interpreterschleife aufgerufen. Sie umfaßt eine Reihe von Vergleichsoperationen und gibt für kurze Zeit die Kontrolle des Programmes an das Betriebssystem ab. Dies ist notwendig, damit wichtige Systemfunktionen ausgeführt werden können oder die Kontrolle an eine andere Applikation übergeben werden kann. Aus diesem Umstand ergibt sich eine hoher Zeitbedarf der Funktion "rtmac_sample", der sich negativ auf die Abarbeitung von Tycoon-Programmen durch den Interpreter auswirkt. Als Lösung wurde ein Mechanismus eingeführt, der "rtmac_sample" lediglich nach einer bestimmten Anzahl von Byte-Codes

aufruft. Diese Zahl wurde heuristisch bestimmt und stellt einen Kompromiß zwischen Rechenleistung des Tycoon-Interpreters und Reaktionsfähigkeit der Tycoon-Applikation auf Benutzerereignisse dar.

Das Tycoon-System benötigt an mehreren Stellen die Information, ob für einen Stream neue, unbearbeitete Daten anliegen. Die Funktion "rtfile_ready" beschafft diese Information über verschiedene Stream- oder Dateioperationen: für Dateien wird die Anzahl unverarbeiteter Bytes als Differenz von Dateilänge und aktueller Dateiposition ermittelt; für STDIN wird die C-Funktion "select" ausgeführt, die den Bool'schen Wert "true" liefert, falls neue Daten anstehen. Die "select"-Funktion gehört nicht zum C-Sprachumfang der MPW Entwicklungsumgebung. Die Tycoon-Konsole muß also nicht nur das bekannte Verhalten von Streams nachbilden, sondern auch eine Funktion bereitstellen, die unbearbeitete Eingaben erkennt.

Die Tycoon-Konsole setzt auf mehreren Managern des Macintosh Betriebssystemes auf. Der Event-Manager stellt die Ereignisse bereit, die in der Funktion "rtmac_sample" verarbeitet werden. Ein Tastendruck z.B. stellt ein solches Ereignis dar. Der TextEdit-Manager (TE) bietet sämtliche Funktionen, die für einen Texteditor benötigt werden. Dazu gehört u.a. die Ausgabe von Zeichen in das Editorfenster.

Die Ein- und Ausgabe von Text in der Tycoon Language wird auf die C-Funktionen "read" und "write" abgebildet, die "Tycoon Virtual Machine" hingegen benutzt die Funktionen "fileRead" und "fileWrite" aus dem Modul "rtfile". Unter Unix operieren diese Aufrufe direkt auf den Streams STDIN und STDOUT. Die Anpassungen für den Macintosh müssen an dieser Stelle ansetzen: es wurden dazu die Funktionen "rtmac_read" und "rtmac_write" eingeführt. Diese Funktionen prüfen, ob der übergebene Stream-Parameter dem Konsole- oder Fehler-Stream entspricht. Ist dies der Fall, so werden für "read"/"fileRead" neue Tastaturereignisse verarbeitet, für "write"/"fileWrite" werden die Daten mittels TE-Manager-Funktionen auf dem Bildschirm angezeigt. Handelt es sich um einen Datei-Stream, so werden die normalen C-Funktionen "read" und "write" aufgerufen.

Die genannten Funktionen können aus zwei Richtungen aufgerufen werden: aus der Hochsprache Tycoon (TL) oder aus der virtuellen Maschine selbst. Für die erste Aufrufart wird wie in Kapitel drei beschrieben eine Hash-Tabelle genutzt, über die der benötigte Funktionszeiger ermittelt wird. In diese Tabelle werden für "read" und "write" jedoch die Zeiger auf die Funktionen "rtmac_read" und "rtmac_write" eingetragen, um so die ausschließliche Verwendung der neuen Funktionen sicherzustellen. Für die zweite Aufrufart wird in den Funktionen "fileRead" und "fileWrite" statt der jeweiligen C-Funktion die angepaßte Variante aufgerufen. Dies wird durch Präprozessoranweisungen realisiert (siehe Abbildung 4.2).

Für die Eingabe von Text wurde ein Ringpuffer eingeführt, um die noch nicht von Tycoon verarbeiteten Eingaben zwischenzuspeichern und um eine Funktion realisieren zu können, die anstehenden Text signalisiert. Dazu wurden globale Variablen für den Puffer selbst ("rtmac_sBuffer"), die aktuelle Lese-("rtmac_iRead") und Schreibposition ("rtmac_iWrite") in diesem Puffer und ein Zähler ("rtmac_iReady") eingeführt, der die Anzahl von Zeilenendezeichen speichert. Die Abbildung 4.3 zeigt die Funktionsweise des Ringpuffers. Der Puffer ist mit 100 KByte großzügig bemessen, um auch umfangreiche Quelltexte aufnehmen zu können.

```
static Word fileRead(rtstream_T stream, void *pBuffer, Word nBytes)
{
  rtdebug_ASSERT(stream);
  rtdebug_ASSERT(stream->hIn);
  rtdebug_ASSERT(pBuffer);
# ifdef rtstd\_LIB\_MPW
    return rtmac_read(stream, pBuffer, nBytes);
# endif
  return fread(pBuffer, /* nItemBytes */ 1, /* nItems */ nBytes,
               (FILE *) stream->hIn);
}
```

Abbildung 4.2: Die Funktion fileRead

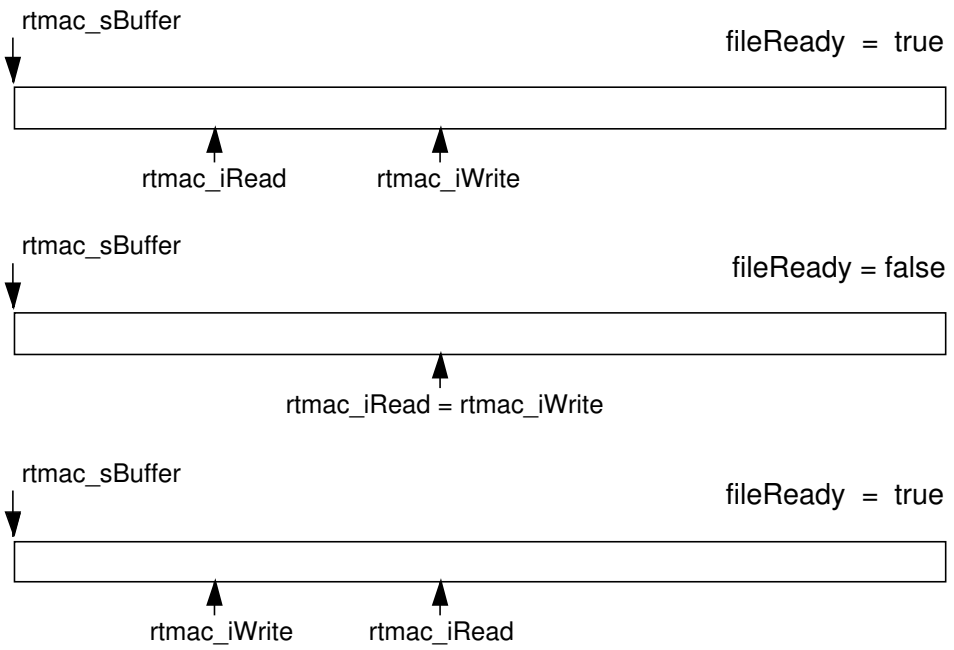

Abbildung 4.3: Ringpuffer für Textein-/Ausgabe

Der Puffer wird in zwei Fällen gefüllt. Ein Tastendruck wird durch das Ereignis "keyDown" signalisiert. Die entsprechende Behandlungsroutine nutzt die TextEdit-Funktion "TEKey", um das Zeichen im Fenster auszugeben. Wurde z.B. die Rückschrittaste (Backspace) gedrückt, so wird das letzte Zeichen korrekt gelöscht. Zusätzlich wird das Zeichen durch die Funktion "putChar" an der aktuellen Schreibposition des Puffers eingefügt und die Schreibmarke erhöht. Das Löschen von Zeichen erfolgt über das Rücksetzen der Schreibmarke. Wird ein Zeilenendezeichen gelesen, so muß es von "Carriage Return" in "Line Feed" gewandelt werden, da MacOS und Unix unterschiedliche ASCII-Codes verwenden und Tycoon Unix-Format erwartet. Bei Zeilenende wird der Zähler "rtmac_Ready" erhöht, um zu signalisieren, daß neue Daten anstehen.
Das Kopieren von Text aus der Zwischenablage in die Tycoon-Konsole ist die zweite Möglichkeit der Texteingabe. Der Text wird durch die TextEdit-Funktion "TEStylePaste" in das Fenster eingefügt und durch "bufferText" in den Tycoon-Ringbuffer übernommen. Innerhalb des Puffers muß wiederum das ASCII-Zeichen für Zeilenende umgewandelt und der Zähler "rtmac_Ready" entsprechend erhöht werden.
Die Funktion "fileReady" im Modul "rtfile" liefert unter Unix durch Aufruf von "select" die Information, ob auf einem Stream neue Daten für die Verarbeitung anstehen. Durch eine Präprozessoranweisung wird in der Macintosh-Umgebung statt der 'select'-Anweisung die Funktion "rtmac_fileReady" ausgeführt, um diese Aufgabe zu erfüllen. "rtmac_fileReady" wandelt die Zählervariable "rtmac_Ready" in einen Bool'schen Wert und liefert diesen als Ergebnis.

4.4 Speicherverwaltung

Die Speicherverwaltung unter dem Macintosh Betriebssystem weist einige Besonderheiten auf, die auf das Alter des Systems zurückzuführen sind und die den Programmierer in einigen Punkten einschränken. Für jede Macintosh-Applikation wird statisch die Größe des Speicherbereiches in Bytes angegeben, der ihr bei der Programmausführung zur Verfügung steht. Ist zur Laufzeit des Programms dieser Speicherbereich erschöpft, so bietet das Betriebssystem keine Möglichkeit, ihn dynamisch zu erweitern. Der Applikationsspeicher ist in mehrere Segmente unterteilt: bei der niedrigsten Speicheradresse beginnt der Heap (Halde), auf dem zur Laufzeit dynamisch Speicherblöcke durch die Applikation alloziert werden können. Darüber liegt der Stack (Keller/Stapel) der Applikation. Er wächst von seiner Obergrenze, der höherwertigen Adresse, dem Heap entgegen. Der Bereich zwischen dem Stack und der Obergrenze des Applikationsspeichers nimmt eine Reihe von globalen Variablen und Sprungtabellen auf. Diese Speicheraufteilung ist in Abbildung 4.4 dargestellt. Die Adresse der Grenze zwischen Heap und Stack ist in der globalen Betriebssystemvariablen "ApplLimit" gespeichert. Das Betriebssystem prüft in regelmäßigen Abständen, ob der Stack in den Heap hineingewachsen ist. Dieser Mechanismus wird als "Stack-Sniffer" bezeichnet. Im Falle eines Stack-Überlaufs würde die Applikation durch das Betriebssystem mit einem Fehler abgebrochen.
Das MacOS umfaßt einige Funktionen für das Allozieren von Speicherblöcken. Dabei wer-

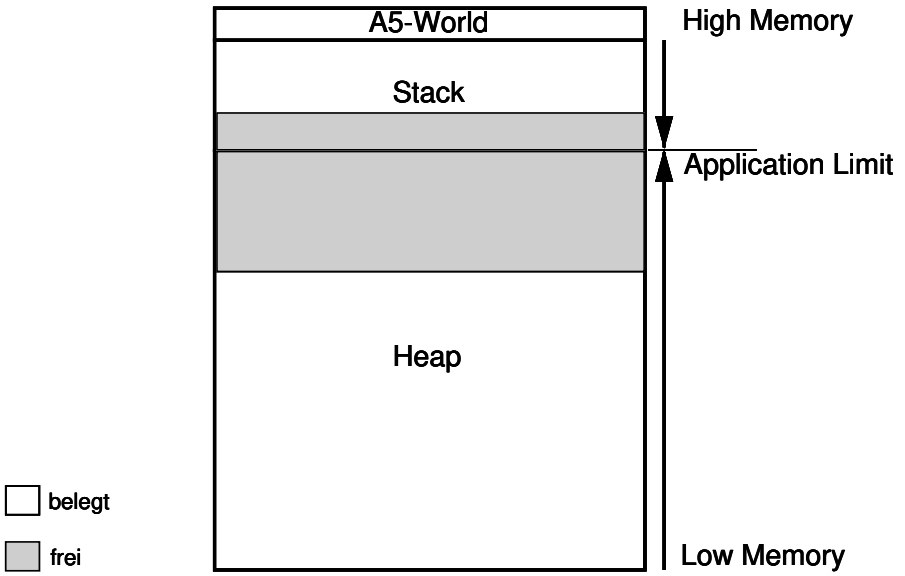

Abbildung 4.4: Das Speichermodell einer Macintosh-Appliaktion

den mehrere Arten von Blöcken unterschieden: "relocatable", "non relocatable", "purgeable" und "unpurgeable". Blöcke vom Typ "relocatable" werden mit der Funktion "NewHandle" angefordert. Sie können durch das Betriebssystem innerhalb des Heap verschoben werden, um zusammenhängende Speicherbereiche zu schaffen. Die Funktion "NewHandle" liefert keine direkte physikalische Adresse, sondern einen indirekten Zeiger (ein "Handle"). Für jedes "Handle" wird ein "Master Pointer" angelegt, ein Eintrag in einer Tabelle ("Master Pointer Block"), die ein Handle auf eine logische Adresse abbildet. Wird ein solcher relokatibler Block durch das Betriebssystem verschoben, so wird in die Tabelle lediglich die neue Startadresse eingetragen, die Adressierung innerhalb des Programms bleibt davon unberührt. In einigen Fällen ist es wünschenswert, nicht verschiebbare ("non relocatable") Speicherblöcke zu reservieren. Dies ist mit der Funktion "NewPtr" möglich, die einen Zeiger liefert, der direkt einen Speicherbereich addressiert. Die Betriebssystem verfolgt mehrere Strategien, um bei Speicheranforderungen zusätzliche Bereiche freizugeben. Verschiebbare Speicherblöcke können zusätzlich das Attribut "purgeable" besitzen. Derartige Speicherblöcke können durch das Betriebssystem deallociert werden. Der Zeiger ("Handle" und "Master Pointer") bleiben dabei erhalten. Das Freigeben von Speicher auf diesem Wege kann auch manuell durch Aufruf der Funktion "PurgeMem" ausgelöst werden.

In einigen Fällen kann der Speicherbedarf einer Applikation den vorhandenen physikalischen Speicher des Rechners überschreiten. Aus diesem Grund bietet das MacOS virtuellen Speicher, der durch den "Virtual Memory Manager" verwaltet wird. Dazu wird auf einem Speichermedium, wie z.B. einer Festplatte, eine Auslagerungsdatei angelegt, in die Speicherbereiche verschoben werden, die nicht unmittelbar in Benutzung sind. Werden Daten in einem ausge-

lagerten Bereich referenziert, wird dies durch den "Virtual Memory Manager" erkannt. Er lädt den entsprechenden Bereich in den Hauptspeicher des Rechners zurück und lagert stattdessen andere Daten aus. Damit diese Vorgänge transparent bleiben, müssen Datenblöcke mit den Funktionen "NewHandle" oder "NewPtr" alloziert werden.
Die Objektspeicher des Tycoon-Systems können beachtliche Größen erreichen, die insbesondere auf gering ausgestatteten Computermodellen wie Laptops den Einsatz von virtuellem Speicher nötig machen. In diesem Zusammenhang traten Programmabstürze auf, wenn der Speicherbereich für den Objekspeicher über die in Unix übliche Funktion "malloc" alloziert wurde. Es zeigte sich, das unter MPW die C-Funktionen "malloc" und "realloc" nur unvollständig auf den Betriebssystemfunktionen aufsetzen. Offensichtlich realisieren sie einen einfacheren Mechanismus zur Speicherverwaltung und greifen nur in einigen Fällen auf die Funktion "NewPtr" zurück. Dieses Verhalten ist wenig in den Handbüchern zum Apple Betriebssystem oden dem MPW C-Referenzhandbuch beschrieben. Die Vermutung liegt nahe, daß der "Virtual Memory Manager" keine Zugriffe auf Speicherbereiche überwacht, die mit "malloc" reserviert wurden. Durch Präprozessoranweisungen wurden diese Aufrufe daher durch sie Funktion "NewPtr" ersetzt.

4.5 Coroutinen-Implementation

Die Tycoon Coroutinen-Implementation stieß unter dem MacOS in erster Linie an Grenzen der Speicherverwaltung. Unter Unix kann eine beliebige Zahl von Coroutinen erzeugt werden, indem Speicher für die Datenstrukturen und insbesondere den Stack einer Coroutine alloziert wird. Auf dem Macintosh verhindert die Überwachung durch den "Stack-Sniffer", daß der Stack-Pointer in den Heap der Applikation zeigt. In einem solchen Fall würde die Applikation mit einer Fehlermeldung beendet. Der Stackbereich der Coroutinen muß also oberhalb der Heap-Grenze "ApplLimit" liegen (siehe Abbildung 4.4). Dazu wurden zur Portierung zwei verschiedene Ansätze erfolgreich erprobt. Die eine Lösung weitet den Stackbereich der Applikation beim Programmstart stark aus, um den Stack mehrerer Coroutinen aufnehmen zu können. Die Variable "ApplLimit" wird entsprechend umgesetzt. Im zweiten Ansatz wird für den Stack jeder Coroutine eine globale Variable ausreichender Größe definiert. Da globale Variablen im Bereich oberhalb des Stacks angelegt werden, entstehen keine Konflikte mit dem "Stack-Sniffer". In beiden Fällen muß gegenüber der Unix-Lösung der Nachteil in Kauf genommen werden, daß die Anzahl der Coroutinen begrenzt ist, da die Speicherblöcke für die Coroutinen-Stacks statisch festgelegt werden. Die Coroutinen existieren damit während der gesamten Laufzeit der Applikation. Neben dem Ring mit aktiven Coroutinen wurde ein zweiter Ring eingeführt, der inaktive Coroutinen aufnimmt (siehe dazu Abbildung 4.5).
Die Funktion "rtcorout_begin" wird für den ersten Lösungsansatz derart modifiziert, daß sie die Datenstrukturen ("rtcourout_T") der Coroutinen im Stackbereich anlegt. Dabei wird mit der niederwertigsten Adresse des Stack begonnen, also seinem oberen Ende. Zwischen den Datenstrukturen wird ein Bereich von der Größe des Coroutinen-Stacks belassen (siehe Abbildung 4.6). Der Aufruf von "rtcorout_new" nimmt unter MacOS keine Speicherallozierung

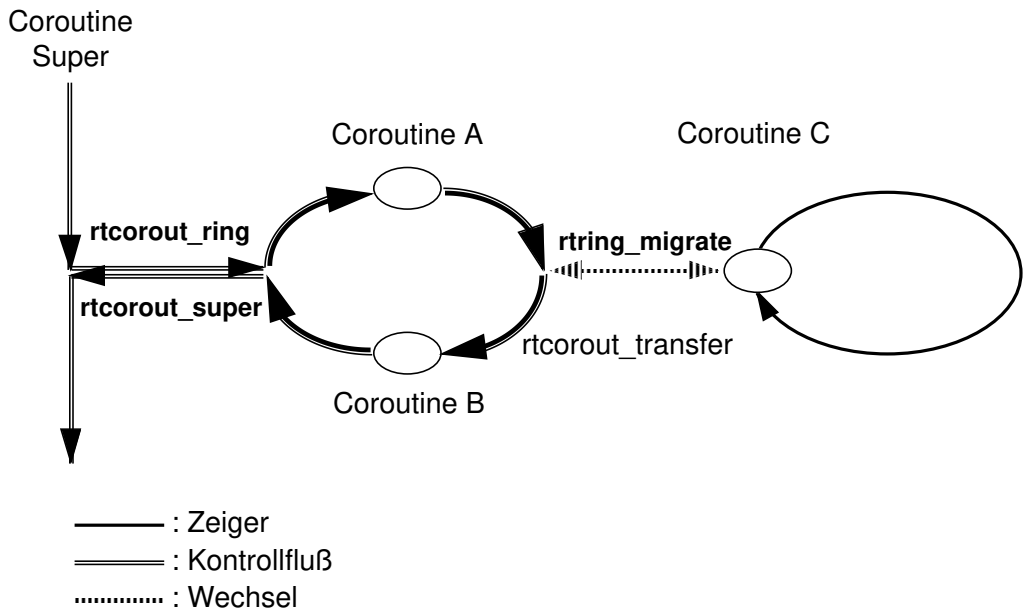

Abbildung 4.5: Ringe für aktive und inaktive Coroutinen

vor und fügt keine neue Coroutine in den aktiven Ring ein, sondern migriert eine existierende aus dem inaktiven in den aktiven Ring. Dabei wird der Stack-Zeiger ("stack pointer", SP) auf die höchste Adresse des Bereiches gesetzt, der durch "rtcorout_begin" über der jeweiligen Coroutine ausgespart wurde. Darüber hinaus wird der Programmzähler ("program counter", PC) mit einem Zeiger auf die Funktion "executeAndFree" initialisiert. Sie enthält den Aufruf des Byte-Code-Interpreters.

4.5.1 Coroutinen: 68k-Macintosh

Die Funktion "rtcorout_transfer" sichert die Prozessorregister, den sogenannten Kontext, der aktiven Coroutine in der Variable "context" der zugehörigen Datenstruktur. Danach übergibt sie die Kontrolle an die nächste Coroutine im Ring. Die Anzahl und die Reihenfolge der Prozessorregister hängt von der zugrundeliegenden Prozessorarchitektur ab. Damit die Funktion "rtcorout_new" die Initialisierung des Stack-Zeigers und des Programmzählers vornehmen kann, muß sie deren Position im Kontext kennen, der durch das "setjmp"-Kommando gesichert wird. Diese Information kann durch den Einsatz eines Debuggers gewonnen werden, indem die einzelnen Schritte der "setjmp"-Funktion nachvollzogen und zu einem geeigneten Zeitpunkt die Registerinhalte des Prozessors mit denen der Kontext-Datenstruktur verglichen werden. Für die Motorola 68k Prozessorarchitektur in Verbindung mit der MPW Entwicklungsumgebung wurde ein Versatz zum Kontextanfang von 6 Worten für den Programmzähler und 11 Worten für den Stack-Zeiger ermittelt. Dieser Versatz

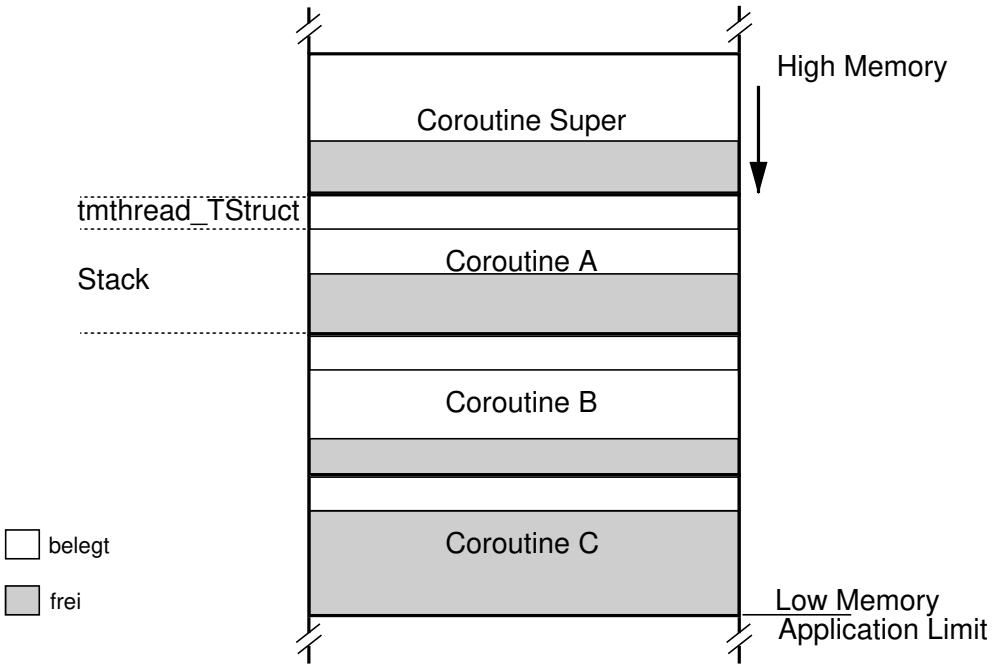

Abbildung 4.6: Stackaufteilung für eine feste Anzahl von Coroutinen

```
#ifdef rtstd_LIB_MPW
# ifdef powerc
#  define iSP               2
#  define iPC               0
#  define iTOC              3
# else
#  define iSP               11
#  define iPC               6
# endif
# define nALIGNMENT        4
# define nBYTES_STACK      (64 * 1024)
# define nBYTES_SPACE      4
# define nMAX_COROUTS      8
# endif
```

Abbildung 4.7: Versatz von Programmzähler und Kellerzeiger

wird in Form von Makros für jede Plattform angegeben, der entsprechende Abschnitt ist in Abbildung 4.7 dargestellt.

4.5.2 Coroutinen: PowerPC-Macintosh

Der PowerPC Prozessor verfügt über zahlreiche Register. Der durch "setjmp" gesicherte Kontext ist wesentlich umfangreicher als der einer 68k-Architektur. Unter der MPW Entwicklungsumgebung beträgt der Versatz von Programmzähler 0, der des Stack-Zeigers 2 Worte. Mit dem PowerPC wurde von Apple ein weiteres zentrales Prozessorregister eingeführt, das als "table of contents"-Register ("TOC") bezeichnet wird. Es ist auf eine neue Runtime-Architektur zurückzuführen, die mit den neuen Prozessoren eingeführt wurde. Es übernimmt die Aufgaben des 68k-Registers A5 und wird für den Zugriff auf globale Daten genutzt. Das Register wird vom Betriebssystem aktualisiert. Die Implementation der "setjmp" und "longjmp"-Kommandos unter MPW sichert den Inhalt dieses Registers zusätzlich 5 Worte oberhalb des Stack. Beide Informationen werden für einen Kontextwechsel über "longjmp" benötigt. Die Funktion "rtcorout_new" muß also das Register "TOC" sowohl im Kontext initialisieren (Versatz 3 Worte) als auch oberhalb des Stack eintragen, um eine funktionsfähige Coroutine zu instanziieren (siehe Abbildung 4.8). Es wurde dafür eine weitere Präprozessoranweisung speziell für PowerPC notwendig.

4.6 Umgebungsvariablen

Die graphische Benutzeroberfläche des Apple Macintosh bietet keine Möglichkeit, Umgebungsvariablen zu definieren, wie dies unter Unix und einigen anderen Betriebssystemen

```
# ifdef powerc
  if (!setjmp(rtcorout_VARIABLE(c, context)))
  {
  pw = (Word *) &rtcorout_VARIABLE(c, context);
  help2 = (Word *) pw[iSP] + 20;
  pw[iSP] = (Word) (c + 1) + nBYTES_STACK;
  pw[iSP] &= ~(nALIGNMENT - 1);
  help = (Word *) pw[iSP] + 20;
  *help = *help2;
  pw[iTOC] = *help2;
  }
  else
  executeAndFree();
  rtexcept_init(c);
  return c;
# else
```

Abbildung 4.8: Sichern des "Table-of-Contents"-Registers unter Power Macintosh

üblich ist[2]. Die entsprechende C-Funktion zur Bestimmung des Wertes einer Umgebungsvariablen ("getenv") ist nicht in den MPW-Bibliotheken enthalten. Tycoon macht zumindest von der Variablen "TYCOON_ROOT" Gebrauch, um den Verzeichnispfad zu bestimmen, unter dem das Tycoon-System installiert ist. Der Zugriff auf die meisten Dateien erfolgt über relative Pfadangaben, die Applikation muß unter Unix dazu im Unterverzeichnis "bootlib" abgelegt sein und dort gestartet werden.
Ein Teil des Macintosh Betriebssystems ist das sogenannte "Standard File Package", das u.a. Auswahlfenster für das Öffnen von Dateien bereitstellt. Das Betriebssystem unterscheidet ein "default directory", auf das zugegriffen wird, wenn bei Dateizugriffen nicht explizit ein Verzeichnis angegeben wurde, und ein "current directory", das durch das "Standard File Package" verwaltet wird. Das "current directory" entspricht dem letzten Verzeichnis, auf das über Dateiauswahlfenster zugegriffen wurde. Wird ein Macintosh-Programm direkt durch Doppelklick gestartet, so entspricht zu Programmbeginn das "current directory" dem Verzeichnis, in dem sich die Applikation befindet. Aus diesem Grund wurde bei der Portierung die Bedingung übernommen, daß sich die Tycoon-Applikation im "bootlib"-Verzeichnis des Tycoon-Dateibaumes befinden und dort gestartet werden muß. Über den Aufruf "LMGetCurDirStore" kann die Applikation in der Startphase das aktuelle Verzeichnis ermitteln, so die Variable "TYCOON_ROOT" herleiten und einer globalen Programmvariablen zuweisen. Die Funktion "rtmac_getenv" implementiert die fehlende C-Funktion "getenv": sie erhält

[2]Für MPW-Werkzeuge können innerhalb der MPW-Shell Umgebungsvariablen definiert werden. Tycoon als MPW-Werkzeug anzubieten würde jedoch eine starke Einschränkung bedeuten, da ein Anwender auch über die MPW-Shell verfügen müßte.

als Eingabeparameter den Namen der Umgebungsvariablen, deren Wert benötigt wird. Sie liefert für "TYCOON_ROOT" den Inhalt der globalen Variablen, die beim Programmstart gefüllt wurde. Andere Umgebungsvariablen werden derzeit nicht unterstützt, es wird in solchen Fällen der Wert "mac" zurückgegeben.

4.7 Socket-Implementation

Für die Kommunikation mit anderen, insbesondere entfernten Prozessen können auf TL-Ebene Sockets eingesetzt werden. Eine entsprechende TL-Bibliothek nutzt den C-Call-Mechanismus, um die unter C verfügbaren Funktionen auch in TL bereitszustellen.

Das Konzept von Sockets ist nicht Bestandteil des Apple Macintosh Betriebssystems. Die entsprechende Funktionalität wird aber durch die "Public-Domain"-Bibliothek "Grand Unified Socket Interface" (GUSI) implementiert. Sie muß zusätzlich zu einer Applikation gebunden werden und bietet mit wenigen Ausnahmen die unter Unix bekannten Funktionen für die Verwaltung von Sockets.

Die unter Unix eingesetzten Tycoon-Bibliotheken für Kommunikation über Sockets sind durch den Einsatz der GUSI-Lösung auch auf dem Macintosh Betriebssystem lauffähig. In einigen Fällen zeigten sich die GUSI-Bibliotheksfunktionen instabil. Insbesondere kann nach einem unerwarteten Verbindungsabbruch keine neue Verbindung über den zuvor genutzten Socket aufgebaut werden.

4.8 Zugriffe auf das Dateisystem

Das Macintosh-Betriebssystem verwendet ein streng hierarchisch aufgebautes Dateisystem ("Hierarchical File System, HFS"). Eine Datei wird dabei durch ihren Namen und eine numerische Verzeichnis-ID identifiziert. Ein Verzeichnis wird wiederum über seinen Namen und die Verzeichnis-ID des darüberliegenden Verzeichnisses angesprochen. Dieses Vorgehen deckt sich nicht mit dem unter Unix, bei dem eine Datei über eine explizite Pfadangabe aufgefunden wird. Das Unix-Verhalten wird durch die Standard-C-Funktionen auf Macintosh-Rechnern nachgebildet, so daß Funktionen wie "open" in gewohnter Weise verwendet werden können. Es ist weiterhin möglich, über die Verkettung von Betriebssystemaufrufen eine Pfadangabe zu einer Datei zu gewinnen.

Im Gegensatz zu Unix stehen für Verzeichnisoperationen keine Funktionen wie "chdir" zur Verfügung. Diese Funktionen werden in Tycoon durch das Modul "directory" in der Bibliothek "STDENV" genutzt. Die im Abschnitt 4.7 beschriebene Bibliothek "GUSI" bildet die Funktion "chdir" nach. Wird die Funktionalität des Moduls "directory" benötigt, so muß die ausführbare Tycoon-Applikation mit der GUSI-Bibliothek gebunden werden. Im Zuge der Portierung hatte diese Bibliothek jedoch negative Auswirkungungen auf die Performanz vor allem von Dateioperationen.

4.9 Datei-Namen

Der Aufbau von Dateinamen ist unter Unix und MacOS sehr ähnlich, weist jedoch folgende Unterschiede auf: als Trennzeichen zwischen Verzeichnis und Dateinamen wird auf dem Macintosh statt eines "/" ein ":" verwendet. Darüber hinaus erfolgt eine direkte Pfadangabe ohne vorangestellte Zeichen wie "/" oder ":", eine relative hingegen muß mit einem ":" eingeleitet werden. Die "Tycoon Language" übergibt Dateinamen im Unix-Format an die virtuelle Maschine. Bei der Portierung wurde die Funktion "rtmac_adaptFileName" eingeführt, um die Dateinamen an die Macintosh-Konventionen anzupassen. Die Umwandlungsroutine prüft, ob eine absolute Pfadangabe vorliegt. Dieser Fall sollte nicht auftreten, da trotz einer Übersetzung der Trennzeichen die Datei vermutlich nicht gefunden werden könnte. Für relative Pfadangaben wird zunächst überpfrüft, ob sie bereits den Verzeichnispfad enthält, der unter der globalen Variablen "TYCOON_ROOT" eingetragen ist. Diese Angabe wird sonst der eigentlichen Pfadangabe vorangestellt. Danach wird in Abhängigkeit weiterer führender Zeichen bestimmt, ob der Pfad zusätzlich um das Verzeichnis "bootlib" ergänzt werden muß. Dabei müssen drei unterschiedliche Kombinationen aus dem Verzeichnis "bootlib" und Trennzeichen berücksichtigt werden. Schließlich wird der so gebildete Pfadname mit dem übergebenen Dateinamen ergänzt, verbleibende Unix-Trennzeichen umgewandelt und die Zeichenkette als Ergebnis zurückgeliefert.

4.10 Dynamisches Binden auf Apple Macintosh

Dynamisches Binden von Funktionen war in frühen Versionen des MacOS nicht möglich. Für die 68k-Familie wurde daher der "Shared Library Manager" entwickelt, um die fehlende Funktionalität zur Verfügung zu stellen. Mit der PowerPC-Architektur wurde der sogenannte "Code Fragment Manager" (CFM) eingeführt. Er nimmt in der Laufzeitarchitektur der neuen Macintosh-Generation eine zentrale Stellung ein (siehe auch TOC-Register im Abschnitt 4.5.2) und ersetzt außerdem den "Shared Library Manager". Zum Zeitpunkt dieser Arbeit lag keine Version des CFM für 68k-Modelle vor.
Das Verhalten beider Lösungen weicht stark vom Ansatz der Shared Libraries unter Unix ab: für die Applikation bleibt der Aufruf von Funktionen stets transparent, gleichgültig ob sie statisch oder dynamisch gebunden sind. Zur Übersetzungszeit der Applikation werden dazu Stubs anstatt der eigentlichen Funktionen gebunden, die Informationen wie die Identifikation und die Version der Shared Library enthalten. Der Programmquelltext enthält gewöhnliche Funktionsaufrufe, der Code Fragment Manager löst diese Referenzen zur Laufzeit auf.
In Tycoon erscheinen Funktionsaufrufe auf dem Macintosh immer wie statische Aufrufe. Die gewünschte Flexibilität geht dadurch verloren und alle Funktionen müssen explizit registriert werden (siehe Abschnitt 3.3.2). Aus diesem Grund und aufgrund der Existens zweier unterschiedlicher Lösungen für 68k- und PowerPC-Modelle wurde bei der Portierung zunächst auf dynamisches Binden verzichtet.

5. Zusammenfassung und Ausblick

5.1 Zusammenfassung und Bewertung der Portabilität

Die Portierung des Tycoon-Systems auf Apple Macintosh wurde durch die eingesetzten Konzepte weitgehend unterstützt. Anpassungen blieben aufgrund der starken Modularisierung auf einige Teile beschränkt. Als überwiegendes Problemfeld stellte sich die Datenein- und -ausgabe im weiteren Sinne heraus.

Das Macintosh-Betriebssystem nimmt im Zusammenhang mit der Entwicklungsumgebung MPW eine Sonderstellung bezüglich der Textein- und -ausgabe ein. Das Konzept von Streams wird nicht unterstützt, die Kommunikation mit dem Benutzer wird über ein Textfenster realisiert. Ein solches Fenster wurde aus einer Apple-Beispielapplikation abgeleitet, in die Tycoon-Applikation integriert und seine Funktionalität an die von Streams unter Unix angepaßt. Die umfassende Unterstützung des Programmierers bei der Erstellung der Benutzerschnittstelle durch Betriebssystemfunktionen führt zu einer konsistenten und für den Benutzer gewohnten Oberfläche. Die Verwaltung des Ein-/Ausgabefensters durch die Tycoon-Applikation erlaubt den Zugriff auf Benutzer- oder Betriebssystemereignisse. Dieser Aspekt kann in zukünftigen Tycoon-Versionen größere Bedeutung erlangen und würde dann eine ähnliche Lösung für andere Betriebssysteme wie Windows95/Windows NT und OS/2 erfordern. Die dafür notwendigen Funktionen müssen in separaten Modulen gehalten werden, da hier keine Portabilität unter den Betriebssystemen zu erwarten ist.

Die Unterschiede zwischen den Dateisystemen unter Unix und MacOS erzwingen eine Anpassung von Dateinamen. Dies trifft auch für das Windows-Betriebssystem zu, so daß eine Lösung auf TL-Ebene sinnvoll erscheint. Die Einführung einer allgemeinen Namens- und Pfadkonvention mit automatischer Anpassung in Abhängigkeit vom zugrundeliegenden Betriebssystem kann in TL flexibel realisiert werden. Die besondere Verwaltung von Verzeichnisstrukturen führt auf dem Macintosh zu einer Einschränkung der Funktionalität des Moduls directory. Dies kann durch den Einsatz der GUSI-Bibliothek umgangen werden, eventuell beheben andere Entwicklungsumgebungen diesen Mangel.

Die Coroutinen-Implementation ist auf dem Macintosh-Betriebssystem sehr stabil. Die Abhängigkeit der Funktionen von der zugrundeliegenden Prozessorarchitektur führte zu Anpassungen sowohl für 68k- als auch für PowerPC-Modelle. Das Konzept könnte auf bestimmten Rechnerarchitekturen zu erhöhtem Anpassungsaufwand führen.

Die Kommunikation mit anderen Rechnern über Sockets ist in der Portierung umgesetzt

worden. Da auf eine Drittlösung zurückgegriffen werden mußte und keine entsprechenden Funktionen durch das Betriebssystem bereitgestellt werden, bleibt dieser Bereich vorerst instabil. Eine neue Version der eingesetzten Bibliotheken ist angekündigt und könnte Abhilfe schaffen.
Ein nicht unbedeutender Anpassungsaufwand ist auf Besonderheiten der MPW-Entwicklungsumgebung zurückzuführen. Es handelt sich meist um abweichende Funktionsnamen oder Umsetzungen des C-Sprachumfanges. Die Unterstützung für Unix-Funktionen fehlt gänzlich. Ein großer Nachteil sind lange Übersetzungszeiten. Die Erfahrungen in dieser Arbeit deuten darauf hin, daß andere Entwicklungsumgebungen besser für die Portierung von Unix-Applikationen geeignet wären.
Als Ergebnis der Portierung steht die Sprache Tycoon Language und die Funktionalität aus den zugehörigen TL-Standardbibliotheken zur Verfügung. Die Konzeption des Tycoon-Systems hat sich trotz zum Teil grundlegender Unterschiede in den Betriebssystemen bewährt. Änderungen wurden vielfach nicht durch Eigenheiten Tycoons, sondern durch die Kombination MacOS/MPW verursacht. Ein Bruch ist für den Objektspeicher TYMEM festzustellen, der nicht auf Funktionen des Runtime-Systems zurückgreift.

5.2 Ausblick

Für die Verbesserung der Portabilität steht eine einheitliche Lösung für die Behandlung von Datei- und Pfadnamen aus. Durch eine C-Funktion, die das zugrundeliegende Betriebssystem der TL-Ebene mitteilt, wäre eine flexible Lösung möglich. An die C-Funktionen für das Öffnen von Dateien würden dann bereits korrekte Zeichenketten übergeben.
Die Eignung von TL als Skriptsprache legt eine Koppelung von Tycoon mit den Skriptfunktionen des Macintosh-Betriebssystems nahe. Dadurch würde die Steuerung von Macintosh-Applikationen aus Tycoon möglich. Dies entspricht der Funktionalität, die für Geschäftsprozessmodellierung und -verwaltung notwendig ist. Bei einer solchen Koppelung würde zwangsläufig Tycoon als Systemprogrammiersprache eingesetzt werden müssen.
In Kürze wird für beide Macintosh Prozessorarchitekturen der Code-Fragment-Manager verfügbar sein. Dadurch wird der Einsatz von dynamischem Binden von Funktionen auch unter MacOS sinnvoll und kann realisiert werden.

Literaturverzeichnis

ApF92 Apple Computer Inc. *Inside Macintosh: Files.* Addison-Wesley Publishing Company, 1992.

ApM92 Apple Computer Inc. *Inside Macintosh: Memory.* Addison-Wesley Publishing Company, 1992.

ApO92 Apple Computer Inc. *Inside Macintosh: Overview.* Addison-Wesley Publishing Company, 1992.

ApP92 Apple Computer Inc. *Inside Macintosh: Processes.* Addison-Wesley Publishing Company, 1992.

ApT92 Apple Computer Inc. *Inside Macintosh: Text.* Addison-Wesley Publishing Company, 1992.

ATE92 Apple Computer Inc. *Inside Macintosh: Toolbox Essentials.* Addison-Wesley Publishing Company, 1992.

AMT92 Apple Computer Inc. *Inside Macintosh: More Macintosh Toolbox.* Addison-Wesley Publishing Company, 1992.

Bro74 P. J. Brown. *Macro Processors and Techniques for Portable Software.* John Wiley & Sons, London, 1974.

Bro77 P. J. Brown. *Software Portability.* Cambridge University Press, Cambridge, 1977.

CaWe85 L. Cardelli and P. Wegner *On Understanding Types, Data Abstraction, and Polymorphism.* ACM Computing Surveys, 1985.

CWI91 CWI, Amsterdam. *The Ergonomics of Software Porting*, 1991.

DBBB95 P. K. Dubey, K. OBrian, K. OBrian, C. Barton, *Single-Program Speculative Multithreading.* IBM Research Division, Yorktown Heights, 1995.

Hen88 J. Henderson. *Software Portability.* Gower Technical Press, Hants, England, 1988.

KR77 B.W. Kernighan and D.M. Ritchie. *The C Programming Language.* Prentice Hall, Englewood Cliffs, New Jersey, 1977.

Mag93 S. A. Maguire. *Writing Solid Code.* Microsoft Press, Redmond, Washington, 1993.

Mar80 C. D. Marlin. *Coroutines.* Springer-Verlag, 1980.

Math92 B. Mathiske. *Kodegenerierung für Programmiersprachen mit Persistenz, Polymorphie und Funktionen höherer Ordnung.* Diplomarbeit, Fachbereich Informatik, Universität Hamburg, Germany, December 1992.

Matt93 F. Matthes. *Persistente Objektsysteme: Integrierte Datenbankentwicklung und Programmerstellung.* Springer-Verlag, 1993. (In German.).

MMM93 B. Mathiske, F. Matthes, and S. Müßig. *The tycoon system and library manual.* DBIS Tycoon Report 212-93, Fachbereich Informatik, Universität Hamburg, Germany, December 1993.

MMueS94 B. Mathiske. *Persistent Polymorphic Programming in Tycoon: An Introduction.* FIDE Technical Report Series FIDE/94/106. FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ, August 1994.

MMS95 B. Mathiske. *On Migrating Threads.* In Proceedings of the Second International Workshop on Next Generation Information Technologies ans Systems Fachbereich Informatik, Universität Hamburg, Germany, June 1995.

MRS92 F. Matthes, A. Rudloff, J.W. Schmidt and K. Subieta. *A Gateway from a DBPL to Ingres: Modula-2, DBPL, SQL+C, Ingres.* FIDE Technical Report Series FIDE/92/54, FIDE Project Coordinator, Department of Computing Sciences, University of Glasgow, Glasgow G128QQ, August 1992.

MS92 F. Matthes and J.W. Schmidt. *Definition of the Tycoon language TL – a preliminary report.* Informatik Fachbericht FBI-HH-B-160/92, Fachbereich Informatik, Universität Hamburg, Germany, November 1992.

MS93 F. Matthes and J.W. Schmidt. *System construction in the Tycoon environment: Architectures, interfaces and gateways.* In P.P. Spies, editor, *Proceedings of EuroArch'93 Congress*, pages 301–317. Springer-Verlag, October 1993.

Sch80 H. J. Schneider, editor. *Portable Software.* German Chapter of the ACM, B. G. Teubner, Stuttgart, 1980.

TKB85 A. S. Tanenbaum, P. Klint, and W. Bohm. *Guidelines for program portability.* Technical Report IW 85/77, Amsterdam Mathematisch Centrum, Amsterdam, 1985.

Wal82 P. J. L. Wallis. *Portable Programmierung*. The Macmillan Press, London, 1982.