DIPLOMARBEIT

Entwicklung eines Playout Managers für ein Multimedia–Datenbankverwaltungssystem

ausgeführt am

Institut für Integrierte Publikations– und Informationssysteme
GMD — Forschungszentrum Informationstechnik GmbH
in Darmstadt

unter Betreuung von

Prof. Dr. Erich J. Neuhold
und
Dipl. Inform./New Jersey Inst. of Technology Heiko Thimm

durch

Cand. Inform. Steffen A. Jakob

Bahnhofstraße 7
64646 Heppenheim

Matrikel Nummer: 841201

Heppenheim, 17.5.1995

Eidesstattliche Versicherung

Hiermit versichere ich an Eides Statt, daß ich die vorliegende Arbeit selbständig und unter ausschließlicher Verwendung der angegebenen Hilfsmittel angefertigt habe.

Darmstadt, den 17.5.1995

Inhaltsverzeichnis

Danksagungen xii

I **1**

1 Einleitung **2**

1.1 Probleme bei multimedialen Präsentationen 3

1.2 Ziel dieser Arbeit 4

1.3 Gliederung dieser Arbeit 5

2 Grundlagen **7**

2.1 Multimedia 7

2.2 Medien 8

2.2.1 Zeitanforderungen 8

2.2.2 Art der Wahrnehmung 9

2.2.3 Kodierung 11

2.2.4 Datenvolumen 12

2.2.5 Klassifizierung gängiger Medien 13

3 Multimediale Datenbank–Verwaltungssysteme **14**

3.1 Vorteile von DBVS gegenüber Filesystemen 14

3.2 Übergang zum MMDBVS 16
3.3 Anforderungen an ein MMDBVS 16
3.3.1 Präsentation 16
3.3.2 Interaktionen 17
3.3.3 Interpretation multimedialer Daten 17
3.3.4 Integrierte multimediale Datentypen 17
3.3.5 Transparente Geräteansteuerung 18
3.3.6 Inhaltliche Abfragen 18
3.3.7 Behandlung hochvolumiger bzw. zeitkritischer Daten . 19

4 Die AMOS–System–Architektur 20
4.1 Anforderungen 20
4.2 Multimediale Erweiterungen von VODAK 21
4.2.1 Der V^3–Video–Server 21
4.2.2 Der Datentyp Audio 22
4.2.3 VML Klasse Video 22
4.3 AMOS–Architektur 23
4.4 Komponenten 24
4.4.1 Der Server 24
4.4.1.1 Das Datenbankmanagementsystem VODAK . 26
4.4.1.2 VML–Schema 26
4.4.1.3 Präsentations–Server 26
4.4.1.4 External Media Servers 26
4.4.2 Der Client 28
4.4.2.1 Applikation 28

4.4.2.2 VRAPI 29
4.4.2.3 Der Multimedia Playout Manager 29
4.4.2.4 Single Media Presenter 30
4.4.2.5 Interaction Object Presenter 30
4.4.2.6 Der STI–Protokoll–Interpreter 30
4.4.2.7 Der Continuous Object Manager 31

5 Der Multimedia Playout Manager 32

5.1 Playout Management 32
5.1.1 Begriffsdefinition 32
5.1.2 Aufgaben des Playout–Managements 32
5.1.2.1 Geräteansteuerung 33
5.1.2.2 Datenstrom–Verwaltung 34
5.1.2.3 Zustand der Präsentation 34
5.1.2.4 Synchronisationsbedingungen 35
5.1.2.5 Behandlung von Benutzer–Interaktionen . . . 35
5.1.3 Modellierung von interaktiven multimedialen Präsentationen 35
5.2 Interaktionen 36
5.2.1 Begriffsdefinition 36
5.2.2 Einfache und komplexe Interaktionen 37
5.2.3 Interaktionen auf Gruppen von Medien 37
5.2.4 Klassifizierung von Interaktionen 39
5.2.4.1 Interaktion bezüglich zeitabhängiger Attribute 39
5.2.4.2 Interaktion bezüglich auditiver Attribute . . . 39
5.2.4.3 Interaktion bezüglich räumlicher Attribute . . 40

5.2.4.4 Typische Interaktionen 40
5.3 AMOS–Konzept für einen MPM 41
5.3.1 Display Handler 43
5.3.2 Single Media Presenters 44
5.3.2.1 Monitor 45
5.3.2.2 Presenter 46
5.3.3 Interaction Object Presenter 46
5.3.4 Playout Dictionary 47
5.3.5 Interaktionshandler 47
5.3.6 Resource Manager 47
5.3.7 Presenter Manager 49
5.3.8 Synchronisationsmanager 50
5.3.9 Exception Handler 50
5.3.10 Extern Interface 51

II **52**

6 Implementierungskonzepte **53**
6.1 Die Bedienoberfläche Motif 53
6.1.1 Programmaufbau 56
6.1.2 Motif–Hauptschleife 57
6.1.3 Callback–Funktionen 58
6.2 Multithreading 60
6.2.1 Traditionelle UNIX–Prozesse 60
6.2.2 Multithreading durch Multitasking 60

6.2.3 Threads innerhalb eines Prozesses 62
6.2.3.1 Synchronisation 63
6.2.3.2 mt–safe Funktionen 65
6.2.3.3 Thread–spezifische Daten 65
6.2.3.4 Zustände von Threads 65
6.2.3.5 Lightweight–Prozesse 66
6.2.4 Vorteile durch Multithreading 68

7 Implementierung **70**

7.1 Entwicklungsumgebung 70
7.1.1 Solaris 70
7.1.2 C++ Compiler 71
7.1.2.1 Exception Handling 71
7.1.2.2 Multithreading 71
7.2 Benutzung von mt–unsafe Funktionen 72
7.3 Auf Motif basierende C++ Klassen 73
7.3.1 Klassenhierarchie 75
7.3.2 multithread–fähige Erweiterung 75
7.4 Kommunikation zwischen den MPM–Komponenten 75
7.5 Oberfläche 79
7.5.1 MPM–Präsentations–Fenster 79
7.5.2 MPM–Environment–Fenster 80
7.5.3 MPM–Debug–Fenster 80
7.5.4 “Send”–Fenster der Emulation des STI–Protokoll–Interpreter 83

7.5.5 "Receive"–Fenster der Emulation des STI–Protokoll–Interpreter 83

7.6 C++ Klassen 83

III **86**

8 Ausblick **87**

A Abkürzungen **91**

Literaturverzeichnis **92**

Abbildungsverzeichnis

4.1 AMOS–System–Architektur 24
4.2 Detaillierte AMOS–System–Architektur 25
4.3 Konverter für MM Standards 27

5.1 MPM–Architektur 41
5.2 Detaillierte MPM–Architektur 43
5.3 Single Media Presenter 44
5.4 Lebensdauer eines SMP 45
5.5 Interaktionshandler 48

6.1 Button Widget 54
6.2 Scrollbar Widget 54
6.3 Text Widget 55
6.4 Radio Button Widget 55
6.5 List Widget 55
6.6 Bibliothekshierarchie einer Motif–Applikation 56
6.7 Konzeptueller Aufbau einer Motif–Applikation 57
6.8 Die Motif–Hauptschleife 57
6.9 Motif: blockierende Funktion 59
6.10 Motif: nicht blockierende Funktion 59

6.11 Traditionelles UNIX–Prozeß–Modell 61
6.12 Aus mehreren Prozessen bestehende Applikation 62
6.13 Multithread–Modell 63
6.14 Synchronisations–Variablen 64
6.15 Zustände von Threads 66
6.16 Auf LWP basierende Threads 67
6.17 RPCs mit einem Thread 68
6.18 RPCs mit mehreren Threads 69

7.1 Komponenten von Solaris 2.x 71
7.2 Ausführung von mt–unsafe Funktionen 73
7.3 SJ–Motif Klassenhierarchie 76
7.4 Anwendung der Thread Extension Library auf Motif 77
7.5 Kommunikation zwischen Multimedia Playout Manager (MPM)–Komponenten 78
7.6 Oberfläche des MPM 80
7.7 MPM Fenster 81
7.8 Environment Fenster 82
7.9 Debug–Fenster 82
7.10 “Send” Fenster der STI–Protokoll–Interpreter 84
7.11 “Receive” Fenster der STI–Protokoll–Interpreter 84
7.12 C++ Klassen 85

Tabellenverzeichnis

2.1 Eigenschaften von Medien 13

5.1 Zulässige Interaktionsarten 40

5.2 Aktionen bzgl. SMP/IOP 50

Danksagungen

Das Gelingen dieser Arbeit wurde durch die Hilfe vieler Personen ermöglicht. Im Speziellen bedanke ich mich bei

- meiner Freundin Regina,
- meinem Betreuer Heiko Thimm,
- Linus Thorvalds, dafür, daß ich einen Teil meiner Diplomarbeit zu Hause durchführen konnte,
- Lars Bindzus für sein SALT–IPSI–HOWTO,
- allen Mitarbeitern und Studenten von DIMSYS für die angenehme Arbeitsatmosphäre, die ich 4 Jahre genießen durfte und
- bei meinen Freunden.

Teil I

Kapitel 1

Einleitung

Die Präsentation von multimedialen Informationen ist durch die fortgeschrittene Rechnertechnologie ein wichtiger Aspekt der Kommunikation zwischen dem Benutzer und dem Computer geworden. Komplexe Sachverhalte können durch die Kombination von Medien wie Video, Audio, Text, etc. effizient und anschaulich vermittelt werden.

Die in der Vergangenheit übliche pure Textdarstellung rührt noch aus Zeiten, in denen Computer lediglich als digitale *number crunsher* verwendet wurden und es einerseits nicht möglich, andererseits aber auch nicht notwendig war, Daten in einer anderen Form anzubieten.

Heutzutage wäre eine ausschließlich „primitive“ Mensch–Maschine–Kommunikation undenkbar. In fast allen Bereichen wird der Benutzer mit multimedialen Daten konfrontiert, angefangen von Informationssystemen über technische Anwendungen, Lehrprogramme, Simulationen und vielen anderen mehr bis hin zu Internetdiensten.

Waren vor ein paar Jahren solche Multimedia–Werkzeuge nur auf Workstations sinnvoll einzusetzen, gibt es mittlerweile sogar viele Programme für preiswerte Personal Computer. Die Entwicklung in diesem Bereich ist enorm und der Bedarf an leistungsfähigen schnellen Softwarekonzepten steigt stetig.

1.1 Probleme bei multimedialen Präsentationen anhand des Beispiels WWW

Komplex strukturierte multimediale Präsentationen, die zugleich Benutzerinteraktionen zulassen, sind vor allem für multimediale Anwendungsdomänen wie z.B. elektronisches Publizieren, computergestützte Aus– und Weiterbildung und Unterhaltung von großer Bedeutung. Der Computer wird immer häufiger als Instrument zum *Informieren* eingesetzt, was z.B. sehr deutlich an der explosionsartigen Verbreitung des World Wide Web (WWW) zu beobachten ist [BLCG+92]. Das WWW soll hier nun exemplarisch als Beispiel eines multimedialen Informationssystems dienen, anhand dessen verschiedene Nachteile und Probleme, die es zu lösen gilt, aufgezeigt werden.

Das WWW ist ein Internet–Dienst, der Hypermedia–Dokumente weltweit verfügbar macht. Diese Dokumente können aus Kombinationen von Text und Graphik, aber auch aus interaktiven Objekten wie z.B. Texteingabefeldern oder Buttons bestehen. Solche statischen Medien können innerhalb einer Applikation zu einem Dokument verknüpft werden. Zusätzlich besteht die Möglichkeit, weitere Medien wie z.B. die kontinuierlichen Medien Video und Audio oder weitere statische Medien (PostScript, ...) über sogenannte Hyperlinks getrennt zu präsentieren.

Ein Problem stellt hier allerdings die **Synchronisation** dar. Da es nicht möglich ist, statische und kontinuierliche Medien konsistent in einem WWW–Dokument zu vereinigen, muß stets auf externe Viewer zurückgegriffen werden. Externe Viewer sind von der verwendeten Plattform abhängig und kein integraler Bestandteil des WWW–Clients. In einer Konfigurationsdatei werden Medien, die nicht vom Client direkt präsentiert werden können, Präsentations–Applikationen zugeordnet (Z.B. unter UNIX: Bilder → xv, Audio → Active Media Object Stores (AMOS)–Audio–Tool). Diese Viewer sind eigenständige Prozesse, die nicht mit der restlichen Präsentation synchronisiert sind. Sie werden lediglich vom WWW–Client gestartet.

Die **Interaktionsmöglichkeiten**, die gängige WWW–Clients anbieten, beschränken sich meist nur auf das Drücken eines Buttons sowie das Selektieren eines Hyperlinks. Der Grund für diese bescheidene Auswahl liegt darin, daß bei der ausschließlichen Verwendung von statischen Medien das Anwendungsspektrum von Benutzerinteraktionen bei weitem nicht so groß ist wie

bei der Verwendung von kontinuierlichen Medien.

Ein weiterer Nachteil ist die **Datenverwaltung**. Gewöhnlich werden die Daten, auf die ein *WWW–Server* zugreift, in Dateien gespeichert, was bei wenigen Daten ein durchaus gangbarer Weg sein mag, mit großen Datenmengen aber die allseits bekannten Probleme bringt. Die Organisation einer großen Menge einzelner Dateien ist ohne weitere Hilfsmittel sehr mühsam.

Aus diesem Grunde empfiehlt es sich, ein Datenbank–Verwaltungssystem (DBVS) (siehe Abschnitt 3) zu verwenden, das systeminterne Dienste zur Verwaltung von multimedialen Daten anbietet. Dadurch ist eine optimale Unterstützung für multimediale Anwendungen gewährleistet. Desweiteren erlaubt ein DBVS mit integrierter Präsentationsmaschine *interaktive* Anfragen (Queries) gegenüber einer Multimedia–Datenbank.

1.2 Ziel dieser Arbeit

Die im letzten Abschnitt durchgeführten Betrachtungen und allgemeinen Überlegungen motivieren die Entwicklung eines DBVS, das

- eine Präsentationsmaschine für beliebige multimediale Dokumente, die sowohl aus kontinuierlichen als auch aus statischen Medien bestehen, als systeminterne integrierte Komponente enthält,
- dem Benutzer mittels einer Präsentationsmaschine die Möglichkeit bietet, die Präsentation auf vielfältige Art und Weise zu beeinflussen.

Der Vorteil eines solchen integrierten Ansatzes besteht darin, daß multimediale Präsentationen von einem einzigen System, dem DBVS, ausgeführt werden, wodurch optimales Handling von multimedialen Daten innerhalb eines DBVS möglich ist [TK95a]. Im Wesentlichen ist der Effizienzgewinn auf die Integration der Präsentationsmaschine mit anderen relevanten DBVS–Diensten zurückzuführen. Demgegenüber müssen bei der Kopplung zwischen einer Präsentationsmaschine und eines DBVS Effizienzeinbußen durch die hier auftretenden Reibungsverluste in Kauf genommen werden.

Vor diesem Hintergrund wurde am GMD[1]–IPSI[2] für das objektorientierte DBVS VODAK[3], das über eine **Client–Server–Architektur** (siehe Kapitel 4) verfügt, ein Konzept für einen Multimedia Playout Manager entworfen [TK95a]. Der MPM stellt als integrierte Client–Komponente dem Benutzer Laufzeitdienste für interaktive multimediale Präsentationen zur Verfügung.

Das Konzept des MPM ist neben der Entwicklung eines Continuous Object Manager, der Erweiterung um multimediale Sprachkonzepte, insbesondere für temporale Kompositionen von multimedialen Präsentationen, ein erstes Ergebnis der Forschungsaktivitäten zum Thema multimediale DBVS. Diese Forschungstätigkeiten sind in der Abteilung AMOS angesiedelt, in der das System Verteiltes Objektorientiertes Datenbanksystem (VODAK) um Sprachkonzepte und neue Komponenten zur besseren Unterstützung multimedialer Anwendungen erweitert wird. Im Nachfolgenden wird das so erweiterte System VODAK als AMOS–System bezeichnet.

Zielsetzung der vorliegenden Diplomarbeit war die Entwicklung eines generischen objektorientierten Implementierungskonzepts für den AMOS–MPM und dessen Realisierung. Insbesondere war dabei ein geeignetes *Multithreading–Lösungskonzept* zu finden, um die Parallelität multimedialer Präsentationen effizient zu unterstützen. Ein weiterer wichtiger Aspekt war, die möglichst konsistente Einbindung eines Window–Systems (z.B. Benutzerinteraktionen, Fensterumgebung) um eine weitgehend standardkonforme (X, Motif) Implementierung des MPM zu erreichen.

1.3 Gliederung dieser Arbeit

In dieser Ausarbeitung werden die erarbeiteten Konzepte beschrieben, die die Realisierung des MPMs ermöglichten. Dabei wurde folgende Gliederung vorgenommen.

- Der MPM ist ein zentraler Bestandteil einer komplexen Architektur. Er kann nicht losgelöst von seiner Umgebung diskutiert werden. In

[1]GMD – Forschungszentrum Informationstechnik GmbH
[2]Integrated Publication and Information Systems Institute
[3]Verteiltes Objektorientiertes Datenbanksystem

Abschnitt 2 werden daher zunächst die Grundlagen für das Folgende geschaffen.

- In Abschnitt 3 wird näher auf multimediale DBVS eingegangen und der Unterschied zu konventionellen DBVS beleuchtet.
- In Abschnitt 4 wird das AMOS–System, in das der MPM eingebettet ist, vorgestellt.
- Abschnitt 5 geht detailliert auf den MPM ein.
- In Abschnitt 6 werden spezielle Konzepte erklärt, die bei der Implementierung des MPM Eingang gefunden haben.
- Das Abschnitt 7 beleuchtet wie der erste Prototyp des MPM implementiert wurde.
- Abschnitt 8 gibt einen Ausblick auf die weitere mögliche Entwicklung des MPM.

Kapitel 2

Grundlagen

In diesem Kapitel sollen Begriffe und Definitionen, die für das weitere Verständnis dieser Arbeit notwendig sind, erläutert werden. Zuerst wird der Begriff **Multimedia** erläutert. Danach werden grundlegende Eigenschaften von Medien analysiert und Klasseneinteilungen vorgenommen.

2.1 Multimedia

Die in dieser Diplomarbeit entwickelte Komponente dient zur Präsentation **multimedialer** Dokumente. Leider gibt es keine allgemein akzepierte Definition des Begriffes **Multimedia**. In dieser Arbeit gilt folgende

> DEFINITION 1 *Ein* **multimediales Dokument** *ist ein Dokument, das mindestens zwei unabhängige Medien enthält.*

Nach dieser Definition ist auch ein Dokument, das aus zwei Video–Clips besteht ein multimediales Dokument. Die Anzahl der Medien ist entscheidend.

2.2 Medien

Ein multimediales Dokument setzt sich aus mehreren unter Umständen verschiedenen Einzelmedien zusammen. Als *Medium* wird allgemein ein Mittel zur Verbreitung und Darstellung von Informationen bezeichnet [Ste93]. Die jeweiligen Eigenschaften dieser Medien bestimmen die Anforderungen, die an ein Präsentationssystem gestellt werden.

Medien können nach verschiedensten Kriterien voneinander unterschieden werden, wobei im aktuellen Kontext für uns hauptsächlich die folgenden Aspekte relevant sind:

- Zeitanforderungen
- Art der Wahrnehmung
- Kodierung
- Datenvolumen

Nachfolgend wird nun jedes der hier aufgeführten Kriterien näher beleuchtet.

2.2.1 Zeitanforderungen

Für die Präsentation eines bestimmten Mediums ist eine der wichtigsten Eigenschaften, ob es statisch oder fortlaufend wahrgenommen wird.

Je nach Ausprägung werden Medien also eingeteilt in

- **statische** Medien
- **kontinuierliche** Medien.

Beispiele für **kontinuierliche** Medien:

- Video

- Audio

Beispiele für **statische** Medien:

- Text
- Rasterbild

Kontinuierliche Medien bestehen aus medien–spezifischen Einheiten, die jeweils zu *bestimmten* Zeitpunkten innerhalb gewisser Toleranzen präsentiert werden müssen. können diese Toleranzen nicht eingehalten werden, kommt es zu Verzögerungen, die nicht nur 'lästig' sind, sondern den Ablauf der Präsentation stören, vor allem wenn dieses kontinuierliche Medium noch mit anderen logisch zusammenhängt oder interagiert.

D.h., daß die Zeitinformation ein Bestandteil des Mediums ist und deshalb natürlich zeitliche Bedingungen an das Präsentationssystem gestellt werden, die zur korrekten Wiedergabe eingehalten werden müssen. Beispielsweise ist es erstrebenswert, bei einer Videosequenz 25 Frames pro Sekunde darzustellen, um Flimmern oder sprunghafte Bewegungsfolgen zu vermeiden.

Statische Medien enthalten keinerlei zeitliche Information und sind deshalb auch wesentlich leichter handzuhaben. Vom zeitlichen Aspekt gesehen ist lediglich sicherzustellen, daß das Medium innerhalb einer multimedialen Präsentation zum korrekten Zeitpunkt dargestellt wird.

In diesem Zusammenhang ist wichtig, daß man zwischen dem *Medium* und den *multimedialen Daten* unterscheidet. So kann z.B. eine Text–Datei sowohl als statischer Text als auch als kontinuierliche Laufschrift interpretiert werden.

2.2.2 Art der Wahrnehmung

Jedem Medium spricht mindestens einen menschlichen Wahrnehmungssinn an, wobei der Mensch die Computerausgabe hauptsächlich über die sogenannten Fernsinne, also das Sehen und Hören, 'verarbeitet' hat. Die zusätzliche Aufnahme von Informationen über den Tastsinn gewinnt immer mehr an Bedeutung und wurde in den letzten Jahren vor allem über den Bereich

der virtuellen Realitäten populär. Im Rahmen dieser Arbeit werden diese Neuentwicklungen nicht einbezogen, denn es sind hier lediglich

- *visuelle* Medien und
- *auditive* Medien.

von Bedeutung.

Beispiele für **visuelle** Medien:

- Bilder
- Text
- Video

Beispiele für **auditive** Medien:

- Musik
- Geräusche
- Sprache

Um **visuelle** Medien darstellen zu können, wird eine graphikunterstützende Oberfläche und entsprechende Software benötigt. Dazu wird in der Regel ein Window–System verwendet.

Die Ausgabe von visuellen Medien wird meistens durch auf die Hardware angepaßte Software bewerkstelligt. Nur in seltenen Fällen kann spezielle Graphik–Hardware eingesetzt werden, womit natürlich eine wesentlich bessere Performanz erreicht werden kann.

Bei **auditiven** Medien wird in jedem Fall spezielle Hardware zur Sound–Ausgabe benötigt. Im Zusammenhang mit multimedialen Dokumenten tritt das Problem auf, daß die gängigen Audio–Devices normalerweise nur einen einzigen Audio–Datenstrom ausgeben können. Dies kann zu Problemen führen, denn man kann leicht multimediale Präsentationen konstruieren, in

denen *mehrere* Audio–Kanäle *überlagert* werden (z.B. Dia–Show mit Hintergrundmusik und Sprache) müssen. Zu diesem Zweck muß unter Umständen ein logisches Gerät zwischengeschaltet werden, das die Datenströme mischt und an das physikalische Device weiterleitet.

2.2.3 Kodierung

Multimediale Daten können durch die rechnerinterne Darstellung klassifiziert werden. Die grobste Klassifizierung wird hierbei durch die Unterscheidung getroffen, ob die Daten des Mediums als

- kodierte Rohdaten,
- nicht kodierte Rohdaten oder als
- Beschreibungsdaten

vorliegen.

Bezieht man noch diverse Datenformate in die Überlegung mit ein, läßt sich die Darstellung zwar noch feiner gruppieren, es wäre jedoch keine konzeptionelle Unterteilung und deshalb an dieser Stelle ohne Bedeutung.

Beispiele für durch **Rohdaten** beschriebene Medien:

- X–Bitmaps
- Audio–Samples

Beispiele für durch **Beschreibungsdaten** beschriebene Medien:

- Animationen
- Computeruntergenerierte Musik

Das Datenvolumen von durch **Rohdaten** beschriebenen Medien ist bei graphischen Daten meistens sehr groß. Dadurch ist die Übertragung solcher

Daten über ein Kommunikationsnetzwerk oft kritisch, insbesondere, wenn die Daten nicht komprimiert sind. Die Präsentation von nicht kodierten Rohdaten ist dafür relativ zeitunkritisch, so können z.B. X–Bitmaps direkt ohne vorherige aufwendige Format–Konvertierung dargestellt werden.

Beschreibungsdaten sind hingegen geringvolumig, sodaß deren Übertragung eigentlich kein Problem darstellt. Um jedoch das Medium tatsächlich zu präsentieren, müssen die Beschreibungsdaten vorher interpretiert und das Ergebnis in ein darstellbares Format konvertiert werden. Dieser Vorgang kann sehr zeitintensiv sein und ist daher für den Ablauf der Präsentation kritisch.

2.2.4 Datenvolumen

Das Datenvolumen eines Mediums entscheidet darüber, wie groß der Durchsatz des verwendeten Kanals sein muß, damit das Medium ohne Verzögerung angezeigt werden kann.

Wir unterscheiden

- hochvolumige Medien
- Medien mit geringem Volumen.

Beispiele für **hochvolumige** Medien:

- Video
- Audio in DAT–Qualität

Beispiele für **geringvolumige** Medien:

- Text
- Audio in Telefon–Qualität

Hochvolumige Medien können Größen erreichen, die die interne Speicherkapazität eines Multimedia–Systems bei weitem überschreiten. Ein digitaler

Video kann beispielsweise mehrere GigaByte groß sein. Dadurch ist es nicht mehr möglich, die Daten als Einheit zu bearbeiten. Außerdem würde die Übertragungszeit den Start der Präsentation zu lange verzögern. Aus diesem Grund ist es notwendig, hochvolumige Medien in Teile zu zerlegen, deren Größe medienspezifisch ist, und diese kontinuierlich zum Ausgabegerät zu transportieren.

2.2.5 Klassifizierung gängiger Medien

In folgender Tabelle werden gängige Medien aufgelistet und den oben diskutierten Kriterien zugeordnet.

	Video	Bild	Audio	Text	Lauftext	MIDI	Animation	Simulation
Zeitanforderung								
statisch		•		•				
kontinuierlich	•		•		•	•	•	•
Wahrnehmung								
visuell	•	•		•	•		•	•
auditiv			•			•		
Kodierung								
kodierte Rohdaten	•	•	•					
unkodierte Rohdaten	•	•	•	•				
Beschreibungsdaten					•	•	•	•
Volumen								
hochvolumig	•*	•*	•*					
geringvolumig				•*	•*	•*	•*	•*

*tendenzielle Zuordnung

Tabelle 2.1: Eigenschaften von Medien

Kapitel 3

Multimediale Datenbank–Verwaltungssysteme

In diesem Kapitel werden die Eigenschaften eines MMDBVS, sowie die Anforderungen, die im Vergleich zu einem konventionellen DBVS gestellt werden, beschrieben.

3.1 Vorteile von DBVS gegenüber Filesystemen

Konventionelle DBVS haben sich in betriebswirtschaftlich–administrativen Anwendungen bewährt [MW91].

Die Vorteile, die bei der Verwendung eines DBVS im Vergleich zu herkömmlichen Dateisystemen erzielt werden, sind zahlreich [MW91, Kra94]. Im Folgenden werden die wichtigsten Eigenschaften aufgezählt.

Zentrale Kontrolle über die Daten. Durch die Verwendung einer Client–Server Architektur, wird die Kontrolle über die Daten einer zentralen Instanz, dem Datenbankserver, übertragen. Die Datenzugriffs– und Manipulationsfunktionalität braucht daher nicht von einer größeren Anzahl von Clients direkt durchgeführt werden.

Kontrollierte Redundanz. Ein DBVS vermeidet die Speicherung redundanter Informationen. Dadurch können keine inkonsistente Versionen redundanter Daten auftreten. Desweiteren wird der Speicherbedarf gering gehalten.

Mehrbenutzerbetrieb. Mehrere Benutzer können auf die selben Daten zugreifen. Dem einzelnen Benutzer steht dabei die volle funktionale Leistung des DBVS zur Verfügung. Dazu sind Mechanismen in das DBVS integriert, um Konflikte, die beim gleichzeitigen Zugriff auf Daten oder Systemresourcen auftreten können, zu behandeln.

Datensicherheit. Falls Manipulationen, die auf Dateien durchgeführt werden, fehlschlagen, befindet sich die Datei gewöhnlich in einem inkonsistenten Zustand. Manipulationen werden von einem DBVS innerhalb einer Transaktion vorgenommen. Eine Transaktion erscheint dabei als eine atomare Anweisung, die die Datenbasis von einem konsistenten Zustand in einen anderen konsistenten Zustand überführt. Bei Fehlersituationen wird die Transaktion nicht ausgeführt, und die Datenbasis bleibt weiterhin verwendbar.

Persistente Verwaltung von Daten. Datenbankverwaltungssysteme sichern, daß die Wirkung einer erfolgreich abgeschlossenen Transaktion nicht mehr verloren geht, es sei denn, daß sie durch eine weitere Transaktion ausdrücklich widerrufen wird [LS87]. Die betroffenen Daten überleben Programme bzw. Prozesse.

Datenschutz. Die Zugriffsrechte auf Dateien sind in der Regel sehr einschränkend. So bietet DOS z.B. überhaupt keine Zugriffsbeschränkungen. UNIX erlaubt, Zugriffsrechte drei verschiedenen Gruppen von Benutzern zuzuteilen (User, Group, Other). In Datenbankverwaltunssystemen hingegen sind beliebige Wege realisierbar, um Daten zu schützen.

Nebenläufigkeit. Die Konsitenzerhaltung der Datenbasis wird auch bei parallelen Transaktionen gewährleistet.

Abfragesprache. Durch generische Query–Mechanismen kann der Benutzer spezielle Dateninhalte selektieren. Diese Mechanismen sind von der Applikation unabhängig.

Effizienz. Durch die Integrierung von Abfrage– und Modifikationsfunktionalität in das DBVS werden diese Operationen effizienter als auf einem Dateisystem durchgeführt.

3.2 Übergang zum Multimedia–Datenbank–Verwaltungssystem

Wie bereits in der Einleitung motiviert wurde, ist es sinnvoll, auch multimediale Daten durch ein DBVS zu verwalten. Ein solches DBVS wird Multimedia–Datenbank–Verwaltungssystem (MMDBVS) genannt.

Es gibt drei Wege, um ein MMDBVS zu realisieren [MW91]:

- Erweiterung eines konventionellen DBVS
- Kombination eines konventionellen DBVS mit medienspezifischen DBVS
- Neuentwicklung eines MMDBVS

3.3 Anforderungen an ein MMDBVS

Im Abschnitt 2.2 wurden Unterschiede zwischen konventionellen Daten und multimedialen Daten aufgezeigt. Diese unterschiedlichen Eigenschaften haben neue Anforderungen zur Folge, die an das MMDBVS gestellt werden.

3.3.1 Präsentation

Die Benutzerschnittstelle muß Geräte bedienen können, die zur Ausgabe der veschiedenen Medien benötigt werden. Es ist erforderlich, eine auf einem Window–System basierende Präsentationsmaschine in das MMDBVS zu integrieren.

3.3.2 Interaktionen

Das MMDBVS muß Interaktionen unterstützen. D.h. es muß auf Echtzeitreaktionen wie z.B. das Anhalten einer Präsentation oder die Veränderung der Lautstärke eines Audios reagieren können. Neben diesen elementaren Interaktionen müssen außerdem komplexe benutzerdefinierte Interaktionen durchführbar sein.

3.3.3 Interpretation multimedialer Daten

Multimediale Daten können abhängig von der Applikation und der Systemresourcen unterschiedlich interpretiert werden. So kann es z.B. durch die Hardware bedingt erforderlich sein, Bilder nur schwarz–weiß darzustellen. Neben diesen system–immanenten Auswirkungen muß es dem Benutzer ermöglicht werden, explizit die Dateninterpretation zu beeinflussen. Beispielsweise kann er spezifizieren, ob in einer bestimmten Präsentation ein Audio mono oder stereo abgespielt werden soll, oder ob ein Text in Normalschrift oder Fettschrift angezeigt wird.

3.3.4 Integrierte multimediale Datentypen

Um multimediale Daten effektiv verwalten zu können, ist es notwendig, entsprechende Datentypen in das MMDBVS zu integrieren. Diese Datentypen stellen medienspezifische Operationen zur Verfügung (z.B. `play`, `stop`, `cut`, `backward`, ... bei einem Video–Datentyp). Die interne Repräsentierung bzw. das Speicherformat der Daten (z.B. GIF[1], JPEG[2], TIFF[3], PBM[4], ... bei Bildern) soll gekapselt werden und damit dem Benutzer unsichtbar bleiben. Dabei sind generische Konzepte zur Einbindung neuer Formate nützlich. Auf diese Weise können bereits heute Anwendungen für zukünftige Datenformate entwickelt werden [MW91].

[1] Graphics Interchange Format
[2] Joint Photographic Experts Group
[3] Tagged Image File Format
[4] Portable Bitmap File Format

3.3.5 Transparente Geräteansteuerung

Das MMDBVS muß verschiedenste Typen von Sekundärspeichern (z.B.: optische Platten, WORM[5]–Speichermedien, analoge Medien, ...) verwalten können, wobei deren technische Ansteuerung dem Benutzer verborgen bleiben soll. Dazu muß die Schnittstelle zum Anwendungsprogramm von gerätespezifischen Eigenschaften abstrahieren. Der Vorteil ist, daß Datenbank–Applikationen geräteunabhängig implementiert werden können. Falls neue Speichermedien in das System integriert werden, ist keine Änderung der Anwendungssoftware notwendig.

3.3.6 Inhaltliche Abfragen

Neben der Suche nach beschreibenden Daten ist eine inhaltliche Abfrage der multimedialen Daten erstrebenswert. Dies kann auf zwei Arten durchgeführt werden:

- automatische Inhaltserschließung
- manuelle Zuordnung semantischer Informationen

Die automatische Ermittlung von Eigenschaften und Strukturen multimedialer Daten erfordert Mechanismen zur Ähnlichkeitsbestimmung von Mustern. Eine automatische Inhaltsadressierung ist derzeit nicht realistisch.

Manuell können multimedialen Daten beliebige semantische Informationen zugeordnet werden, über die gesucht werden kann.

> Beispiel 1 *Einem Video, welcher einen Spielfilm enthält, können bestimmte Zeitintervalle als Vorspann, Hauptteil und Nachspannn interpretiert werden. Der Hauptteil kann wiederum in eine Menge von Innen– und Außenaufnahmen unterteilt werden, usw.*

[5]Write Once Read Multiple

3.3.7 Behandlung hochvolumiger bzw. zeitkritischer Daten

Hochvolumige Daten erfordern eine zu konventionellen Daten unterschiedliche interne Verwaltung. Es muß vermieden werden, redundante Versionen im Speicher zu halten, da sonst die Systemkapazität schnell überlastet wird. Anstelle von Kopien sind Referenzen notwendig.

Multimediale Daten können oftmals nicht als eine Einheit in den Speicher geladen werden. Deshalb muß das MMDBVS in der Lage sein, auf Teile der Gesamtdaten zuzugreifen, was jedoch für den Benutzer transparent bleibt [Kra94].

Das Lesen und Schreiben von zeitkritischen Daten muß innerhalb vorgegebener zeitlicher Schranken durchgeführt werden. Damit diese Schranken eingehalten werden, sind Scheduling–Mechanismen erforderlich.

Kapitel 4

Die AMOS–System–Architektur

Der MPM ist ein Bestandteil der AMOS–System–Architektur. In diesem Kapitel wird nun näher auf das AMOS–System eingegangen.

Das AMOS–System stellt eine Erweiterung des objektorientierten DBVS VODAK dar. Seine Aufgabe ist es, Mechanismen zur Verwaltung von multimedialen Informationen zur Verfügung zu stellen, und mehreren Benutzern gleichzeitigen Zugriff darauf zu gestatten.

Bisherige Multimedia–Systeme besitzen meistens keine Datenbank–Funktionalität und sind nur für Einzelanwender konzipiert. Diese Systeme stoßen schnell an Grenzen, wenn größere Mengen von multimedialen Daten verwaltet werden sollen [RL94].

4.1 Anforderungen

Durch die Verwendung von VODAK können konventionelle Daten wie Strings und Integer–Werte bereits ausreichend gehandhabt werden. Zusätzliche Konzepte sind jedoch im Zusammenhang mit multimedialen Daten notwendig. Wichtige Aspekte sind hierbei

- Medienspeicherung,
- Medientransport,

- Medienbearbeitung und
- Medienpräsentation.

Vor allem kontinuierliche Medien (siehe Abschnitt 2.2.1), die im Gegensatz zu konventionellen Daten zeitabhängig sind, bedürfen einer besonderen Behandlung.

Das AMOS–System soll folgende Anforderungen erfüllen:

- Verwalten großer Mengen multimedialer Daten
- mächtige Modellierungsmöglichkeiten für multimediale Dokumente
- Integrierte Mechanismen zum Präsentieren multimedialer Daten
- Mehrbenutzerzugriff auf die Daten
- Datenbank–Funktionalität

4.2 Multimediale Erweiterungen von VODAK

Bisher wurden für das DBVS VODAK bereits einige Speziallösungen entwickelt. Anhand der dabei gesammelten Erfahrungen wurde sichtbar, welche Mängel VODAK beim Bearbeiten multimedialer Daten aufweist und welche Erweiterungen deshalb noch notwendig sind.

4.2.1 Der V^3–Video–Server

Die erste Multimedia–Anwendung für VODAK war der V^3–Video–Server [Jak93, RM93]. Dort wurden auf Schema–Ebene Objekttypen für die Medien Audio und Video sowie für die Ausgabegeräte Laser Video Rekorder, MPEG[1]–Player und Imestre–Player modelliert. Der Benutzer kann in einer Motif–Oberfläche nach bestimmten Video–Clips suchen; als Suchkriterien dienen diverse Beschreibungsdaten der Videos wie z.B. Titel, Autor oder

[1]Moving Pictures Experts Group

Länge des Videos. Die ausgewählten Videos können über ein Control–Panel, wie es von herkömmlichen Video–Rekordern bekannt ist, präsentiert werden.

Bei dieser Realisierung mußte die medienspezifische Präsentations–Funktionalität in externen Modulen implementiert werden. Dies bedeutet, daß die Anwendung die Daten selbst interpretieren muß. Diese Aufgabe kann bei weitem nicht so effizient gelöst werden, als wenn in das System integrierte Module diese Aufgabe übernehmen würden.

Desweiteren bestand keine Möglichkeit, die Video–Daten kontinuierlich von der Datenbank zum Präsentations–Modul zu transportieren. Durch das gewaltige Volumen der digitalen Video–Daten hatte dies auch schon bei kurzen Video–Clips große Verzögerungen zur Folge.

Als vorteilhaft erwies sich bei der Benutzung eines objektorientierten DBVS dessen Mächtigkeit bei der Modellierung der Multimedia–Objekte. So konnte z.B. das Format der Videos (MPEG, Imestre, analog) durch Kapselung transparent gehalten werden.

4.2.2 Der Datentyp Audio

In [Har94] wurde der Datentyp Audio in das Datenbanksystem integriert; d.h. die Audiofunktionalität wurde nicht auf der Anwenderebene implementiert, sondern ist nun ein integrierter Bestandteil von VODAK. Eine auf diesem Datentyp basierende Anwendung ist das AMOS–Audio–Tool [RL94].

Bei dieser Applikation zeigte sich, daß sich die Integration von nicht konventionellen Medien in ein konventionelles DBVS schwierig gestaltet. Die daraus resultierende Schlußfolgerung für die Entwicklung des MPM war die Wahl eines **generischen** Konzeptes für das Einbinden verschiedener Medien.

4.2.3 VML Klasse Video

Eine Weiterentwicklung von 4.2.1 stellt die in einem an der TH–Darmstadt durchgeführten Praktikum entwickelte VML[2] Klasse Video dar [GSS+94]. Bei dem dort gewählten Ansatz konnten bereits die Medien Video und Audio

[2] VODAK Manipulation Language

kombiniert und synchronisiert präsentiert werden. Desweiteren wurden die Videodaten kontinuierlich mittels der VODAK–Komponente Continuous Object Manager (COM) zum Präsentations-Modul transportiert.

Die hierbei gesammelte Erfahrungen, beispielsweise über den kontinuierlichen Datenfluß oder die Synchronisation, sind sehr wichtig für die Entwicklung des AMOS–Clients. Die Klasse Video stellt jedoch eine Speziallösung mit einer vorgegebenen Benutzeroberfläche dar. Innerhalb des AMOS–Systems sollen Medien hingegen frei kombinierbar sein.

4.3 AMOS–Architektur

Das AMOS–System besitzt eine Client–Server–basierte Architektur. Es existiert ein zentraler DBVS–Server (VS). Die Clients besitzen im Gegensatz zur früheren VODAK–Implementierung keine Datenbank–Funktionalität. Datenbankanfragen müssen daher von den Clients an den Datenbank–Server geschickt werden. Abbildung 4.1 zeigt ein grobes Modell der AMOS–Architektur.

Gegenüber konventionellen Daten stellen multimediale Daten zusätzliche Anforderungen an das Speichermedium. Daher wird hier zwischen der Behandlung von konventionellen und multimedialen Daten unterschieden. Multimediale Daten können optional auf externen Speichermedien abgelegt sein und von External Media Server (XMS) verwaltet werden oder auch direkt auf dem Server gespeichert sein. Aus Gründen der Performanz kann eine direkte Verbindung von den Clients zu den XMS hergestellt werden, damit der zeitkritische Transport der hochvolumigen Informationen nicht den Umweg über den Server nehmen muß. Diese Verbindung gilt nur für den Transport der multimedialen Daten. Kontrolldaten und konventionelle Daten werden weiterhin über den Server geleitet. XMS können medienspezifisch sein, um den individuellen Anforderungen der verschiedenen Medien zu genügen.

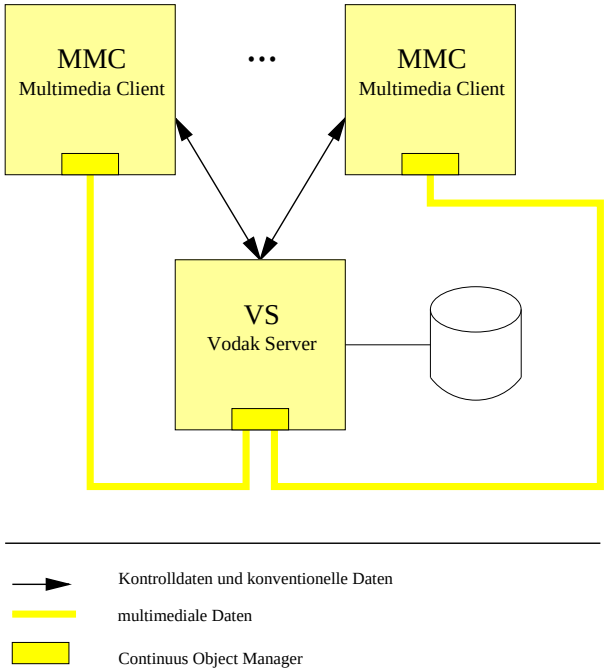

Abbildung 4.1: AMOS–System–Architektur

4.4 Komponenten

Im Folgenden werden die in Abbildung 4.2 abgebildeten Komponenten des AMOS–Systems näher beleuchtet.

4.4.1 Der Server

Der Datenbank–Server dient zur Speicherung der Beschreibungsdaten der multimedialen Dokumente. Das Format der Dokumente kann prinzipiell ein beliebiger Multimedia–Modellierungsstandard sein (z.B. HyTime[3] oder MHEG[4]). Die Spezifikation wird von dem Präsentationsdesigner in VML vorgenommen. Aus dieser Beschreibung wird zur Laufzeit von einem Konvertierungsmodul ein Präsentationsplan erzeugt. Dieser Vorgang läuft dynamisch ab; d.h. der so generierte Plan kann unter Umständen durch Benutzerinter-

[3]Hypermedia/Time–Based Structuring Language
[4]Multimedia and Hypermedia Information Coding Experts Group

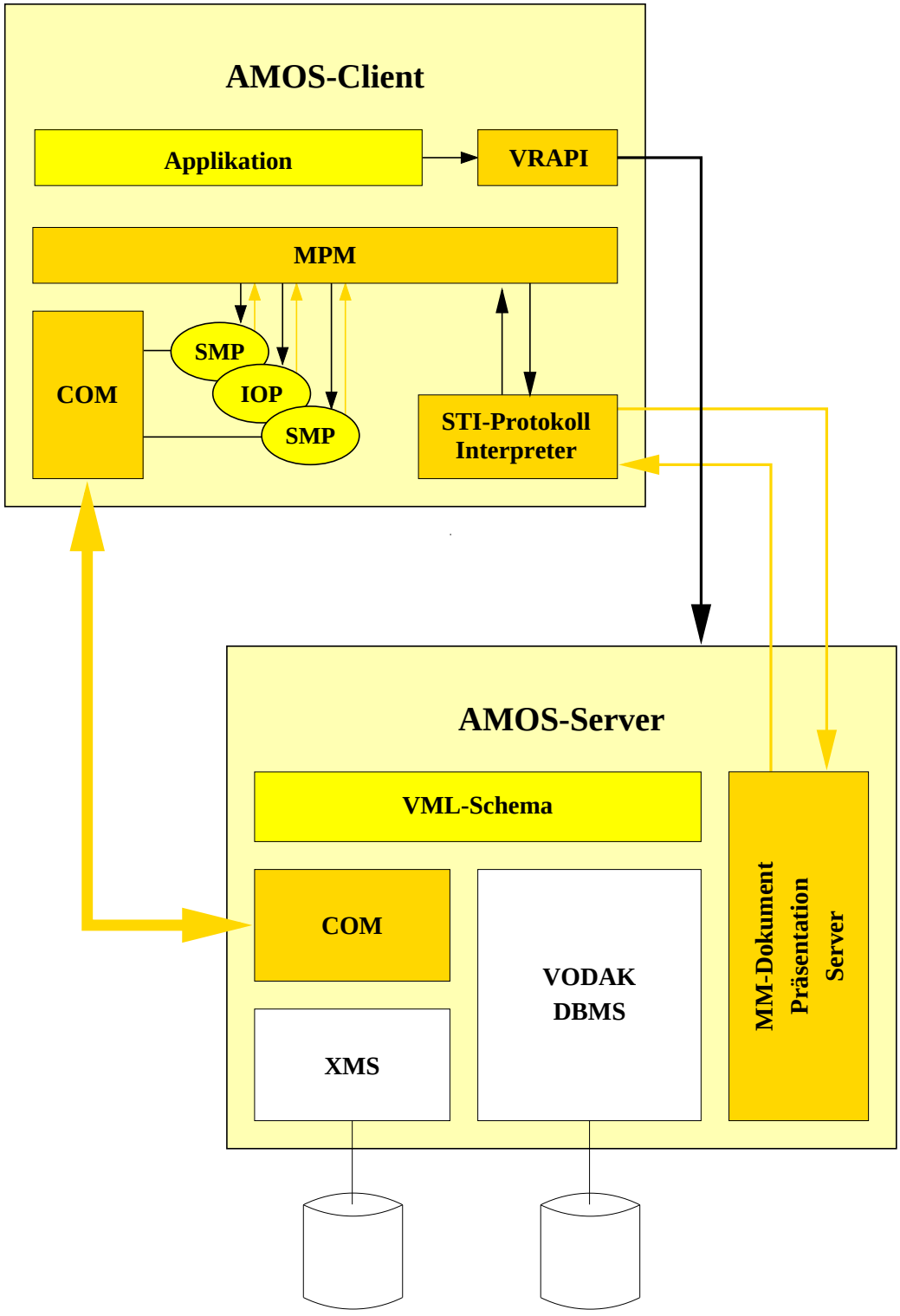

Abbildung 4.2: Detaillierte AMOS–System–Architektur

aktionen, die auf dem Client stattfinden, modifiziert werden. Der generierte Plan wird also nicht als Einheit zum AMOS–Client transportiert, sondern kontinuierlich.

4.4.1.1 Das Datenbankmanagementsystem VODAK

Als Server wird das am IPSI entwickelte objektorientierte DBVS VODAK verwendet. Dadurch können Daten derart modelliert werden, daß durch Kapselung interne Datenstrukturen und darauf anwendbare Operationen dem Benutzer und der Anwendung verborgen bleiben [GMD94]. Die Benutzerschnittstelle zu VODAK stellen die Definitions– und Programmiersprache VML sowie die Query Language VODAK Query Language (VQL) dar.

4.4.1.2 VML–Schema

Die multimedialen Dokumente werden in VML modelliert. Ein VML–Datenbankschema beschreibt die logische Struktur der Dokumente. Der Designer legt hier nicht nur fest, aus welchen Medien das Dokument besteht, sondern er spezifiziert außerdem, welche Interaktionsmöglichkeiten dem Anwender geboten werden sollen.

4.4.1.3 Präsentations–Server

Momentan werden die Präsentationspläne direkt in VML modelliert. In Zukunft sollen sie zur Laufzeit dynamisch generiert werden. Als Basis werden abstrakte Beschreibungsformen dienen. Für jedes Datenmodell wird dann ein Konverter benötigt, der aus dem VML–Modell ein internes Format generiert (siehe Abb. 4.3). Dadurch können viele verschiedene Arten von Multimedia–Dokumenten in **einem** DBVS integriert werden.

4.4.1.4 External Media Servers

Multimediale Medien stellen hohe Anforderungen an

- Speicherbedarf

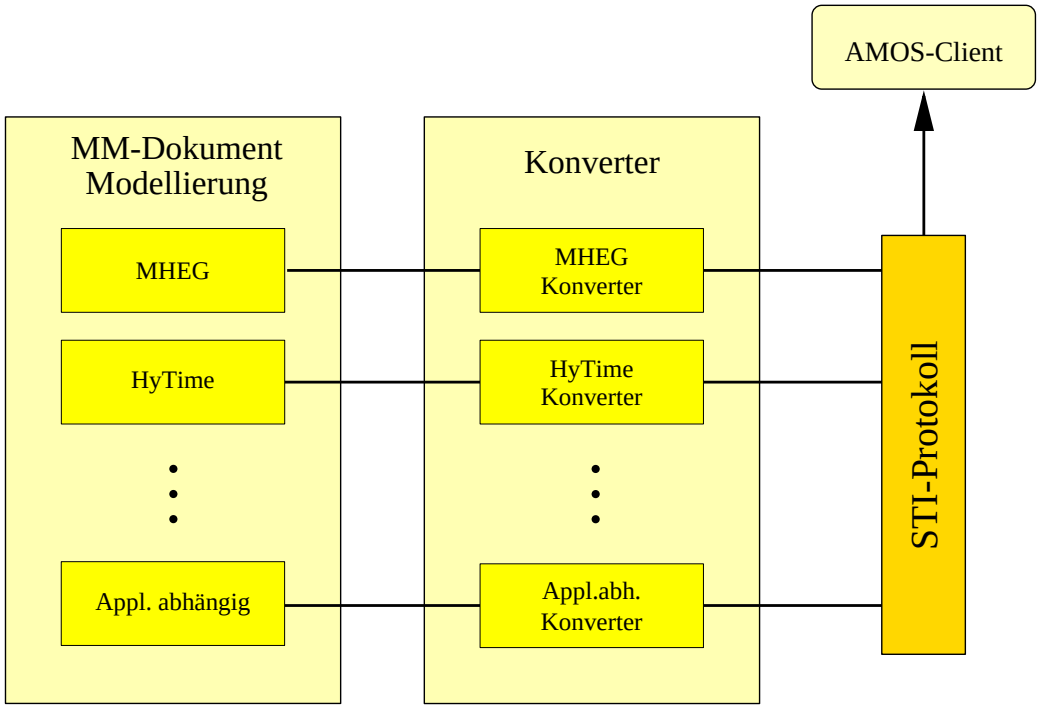

Abbildung 4.3: Konverter für MM Standards

- Zugriffszeit
- Übertragungsgeschwindigkeit

Konventionelle Massenspeicher sind in der Regel für eine Präsentation in Echtzeit nicht ausreichend. Deshalb können XMS in das AMOS–System integriert werden, die externe Massenspeicher verwalten, die für bestimmte Medientypen optimal sind.

Beispiele für solche Massenspeicher sind

- optical disks
- Laser Video Rekorder
- CD–ROM

Für den Anwender bleibt die Sonderstellung der XMS transparent, da die Schnittstelle zu den Daten der COM ist.

4.4.2 Der Client

Der AMOS–Client stellt die Schnittstelle zwischen dem Benutzer und dem AMOS–System dar.

Eine wichtige Aufgabe des Clients ist die Bereitstellung von Präsentationsfunktionalität für multimediale Dokumente. In Abbildung 4.2 sind die wichtigsten Komponenten des Clients erkennbar, die im Folgenden erläutert werden:

- Applikation (Abschnitt 4.4.2.1)
- VRAPI (Abschnitt 4.4.2.2)
- MPM (Abschnitt 4.4.2.3)
- SMP (Abschnitt 4.4.2.4)
- *STI–Protokoll–Interpreter* (Abschnitt 4.4.2.6)
- COM (Abschnitt 4.4.2.7)

4.4.2.1 Applikation

Eine auf dem Client ablaufende Applikation kann Datentypen verwenden, die dem Server bekannt sind. Die in den Applikationen verwendeten Daten sind jedoch nicht persistent. Auf dieser Ebene ist keine komplexe VML – Modellierung möglich, sondern nur eine Untermenge von VML verwendbar. Dem Benutzer wird eine Schnittstelle zur Verfügung gestellt, über die er die im Client integrierte Präsentations–Funktionalität verwenden kann.

> BEISPIEL 2 *Durch eine VQL–Query werden alle multimedialen Dokumente selektiert, die mindestens Video und Audio enthalten und außerdem von einem bestimmten Autor sind. Diese VQL–Query wird am Datenbank–Server abgesetzt und die selektierten Dokumente werden präsentiert.*

4.4.2.2 VRAPI

Das VODAK Remote Application Programming Interface (VRAPI) bietet dem Anwender Zugriffsmöglichkeiten auf den Datenbankserver. Dies ist notwendig, da der Client selbst **keine** Datenbankfunktionalität besitzt.

Das VRAPI stellt folgende Methoden zur Verfügung:

1. `sendMethod(...)`: Sendet einen Methodenaufruf an den Datenbank–Server und wartet auf das Ergebnis
2. `createDb(...)`: Erzeugt eine neue Datenbank auf dem Server
3. `openDB(...)`: Öffnet eine existierende Datenbank auf dem Server
4. `closeDB(...)`: Schließt eine offene Datenbank auf dem Server

Das VRAPI besteht aus dem Client–Messagehandler, der für (1) verantwortlich ist und aus dem Client–DBI[5]–Handler, der (2), (3) und (4) bereitstellt.

Durch die Versendung spezieller Methoden (`TransactionBegin` und `TransactionEnd`) besitzen Clients außerdem die Fähigkeit, Transaktionen zu starten und zu beenden.

4.4.2.3 Der Multimedia Playout Manager

Der Multimedia Playout Manager (MPM) ist eine Präsentationsmaschine für multimediale, interaktive Dokumente. Er ist für den Ablauf der Präsentation verantwortlich und stellt dazu elementare Präsentations–Funktionen zur Verfügung, die vom *STI–Protokoll–Interpreter* (siehe Abschnitt 4.4.2.6) aufgerufen werden.

Der MPM hat zwei Hauptaufgaben:

- Realisierung der Präsentation unter möglichst konfliktfreier Einhaltung

[5]Database Interface

der in einem STI–Protokoll definierten Restriktionen (z.B. Datenqualität, Synchronisationsrestriktionen) und Behandlung von Benutzerinteraktionen durch

- Instantiierung, Koordinierung und Steuerung von Single Media Presenter (SMP)–Threads (siehe Abschnitt 4.4.2.4) und Interaction Object Presenter (IOP) (siehe Abschnitt 4.4.2.5)

In Abschnitt 5 auf Seite 32 wird dieses Modul im Detail beschrieben.

4.4.2.4 Single Media Presenter

Jedes Medium der Präsentation wird durch einen Single Media Presenter (SMP) kontrolliert und dargestellt. SMP werden technisch durch *Threads* realisiert. Die Kontrollinstanz für die SMP ist der MPM, der die SMP instanziert und überwacht.

4.4.2.5 Interaction Object Presenter

Ein Interaction Object Presenter (IOP) ist für die Visualisierung und Verwaltung von graphischen Objekten verantwortlich, mit deren Hilfe der Anwender Interaktionen auslösen kann (siehe Abschnitt 5.3.3).

4.4.2.6 Der STI–Protokoll–Interpreter

Der *STI–Protokoll–Interpreter* verarbeitet jene Teile des Präsentationsplans, welche kontinuierlich von dem Datenbank–Server geliefert werden.

Damit Verzögerungen beim Erzeugen der SMP vermieden werden, wird zwischen Methoden zum

- Vorbereiten und
- Starten

der Präsentation eines Einzelmediums unterschieden. Dazu muß der *STI–Protokoll–Interpreter* die Informationen vom Server mit einer entsprechenden Vorlaufzeit erhalten.

Diese Vorlaufzeiten hängen ab von

- dem zu präsentierenden Medium,
- der Rechnerauslastung und
- der Netzbelastung.

Der *STI–Protokoll–Interpreter* dient als **"driving force"** und verwaltet die Zeitachse, auf die sich die Präsentation bezieht.

4.4.2.7 Der Continuous Object Manager

Der Continuous Object Manager (COM) ist die AMOS–Komponente, die für den kontinuierlichen Datentransport verantwortlich ist. Er wurde speziell für die Behandlung multimedialer Daten konzipiert [Kra94].

Der COM ist in der Lage, mehrere Datenströme parallel zur Verfügung zu stellen. Die elementare Einheit eines derartigen Datenstroms wird Continuous Object Data Unit (CODU) genannt. Im Falle eines Audio–Objektes entspricht ein CODU einem Sample. Da ein Audio–Sample in der Regel sehr klein ist, würde ein großer Overhead entstehen, wenn jedes Sample getrennt übertragen werden würde. Aus diesem Grund wird eine bestimmte Anzahl von CODUs zu COPU[6]s zusammengefaßt. Im Falle des Mediums Audio entspricht ein CODU einem Block bestehend aus einer Sequenz von Samples.

[6] Continuous Object Presentation Unit

Kapitel 5

Der Multimedia Playout Manager

In diesem Kapitel wird das Lösungskonzept für den MPM dargstellt. Zunächst wird der Begriff des *Playout Managements* in Abschnitt 5.1 erklärt. Abschnitt 5.2 geht dann näher auf Interaktionen ein. Danach wird in Abschnitt 5.2 das Lösungskonzept des MPM im Detail erläutert.

5.1 Playout Management

5.1.1 Begriffsdefinition

> Definition 2 *Das Playout Management umfaßt alle Systemfunktionen, die notwendig sind, um gespeicherte Präsentationen darzustellen [TK95a].*

Die Präsentation wird vom Zeitpunkt der Initiierung bis zum Ende der Darstellung vom *Playout Management* kontrolliert.

5.1.2 Aufgaben des Playout–Managements

In [TK95a] werden folgende vom *Playout Management* zu erfüllende Aufgaben definiert:

- Geräteansteuerung,
- Datenstrom–Verwaltung,
- Überwachung des aktuellen Zustandes der Präsentation,
- Einhaltung vordefinierter Synchronisationsbedingungen und
- Behandlung von Benutzer–Interaktionen.

Im Folgenden werden diese Aufgaben des *Playout Managements* diskutiert

5.1.2.1 Geräteansteuerung

Multimediale Präsentationen benötigen im allgemeinen verschiedene Ausgabegeräte zur Darstellung (siehe Abschnitt 2.2.2 auf Seite 9). Dabei unterscheidet man zwischen physikalischen und logischen Geräten:

- physikalisch
  - Bildschirm
  - Lautsprecher
  - LED
  - Keyboard mit einer MIDI–Schnittstelle
- logisch
  - Window–System
  - Audio–Device

Jedem Einzelmedium $m \in \mathbb{M}$ ist durch eine Abbildung α ein Ausgabegerät $d \in \mathbb{D}$ zugeordnet, wobei $\mathbb{M}$ die Menge aller Medien und $\mathbb{D} = \mathbb{D}_{PH} \cup \mathbb{D}_L$ die Menge aller physikalischen und logischen Ausgabegeräte ist.

$$\alpha : \mathbb{M} \rightarrow \mathbb{D} \tag{5.1}$$

Die Menge aller in einer Präsentation p vorkommenden Medien sei $\mathbb{M}_p$. Um p darstellen zu können, muß das Playout–Management–System folgende Menge $\mathbb{D}_p$ von Geräten unterstützen:

$$\mathbb{D}_p := \bigcup_{m \in \mathbb{M}_p} \alpha(m) \tag{5.2}$$

5.1.2.2 Datenstrom–Verwaltung

Das *Playout Management* ist für die Instanziierung und die Überwachung von Datenströmen zwischen Speichermedien und Ausgabegeräten verantwortlich. Dabei müssen benutzerdefinierte *Restriktionen* eingehalten werden. Falls die Resourcen, die für die Präsentation notwendig sind, im Laufe der Präsentation nicht mehr ausreichen, muß eine Adaption stattfinden, damit der Durchsatz erhöht werden kann.

Folgende Adaptionsstrategien sind z.B. denkbar:

- Auslassen von Video–Frames,
- Umschalten von Farb– auf Schwarz–Weiß–Qualität,
- Umschalten auf niedrigere Bildqualität,
- Verwendung eines alternativen Mediums (z.B. Text statt Audio)

5.1.2.3 Überwachung des aktuellen Zustandes der Präsentation

Eine ständige Überwachung des Präsentationszustandes ist notwendig, um auf eventuell auftretende kritische Situationen reagieren zu können. *Playout Management* schließt Synchronisationsmanagement ein. Um die Präsentation synchronisieren zu können, muß das System stets über die aktuell aktiven Medien, deren Abhängigkeit voneinander und deren *Restriktionen* informiert sein.

5.1.2.4 Synchronisationsbedingungen

Das *Playout Management* muß die Einhaltung der Synchronisations–Restriktionen, die bei der Modellierung einer multimedialen Präsentation definiert wurden, sicherstellen. Es existieren *Restriktionen* bzgl. der

- **Intermedia–Synchronisation** (Synchronisation zwischen mehreren Medien)
- **Intramedia–Synchronisation** (medieninterne Synchronisation)

Bei der Synchronisation einer multimedialen Präsentation ist das gesamte Geflecht der Abhängigkeiten der Einzelmedien voneinander zu berücksichtigen. Ein allgemeiner Ansatz für das AMOS–System wird zur Zeit im Rahmen einer Diplomarbeit entwickelt.

5.1.2.5 Behandlung von Benutzer–Interaktionen

An das *Playout Management* wird die Forderung gestellt, daß eine laufende Präsentation zu jedem Zeitpunkt vom Benutzer beeinflußt werden kann. Dazu muß bereits bei der Modellierung der Präsentation definiert werden, welche potentiellen Interaktionen in bestimmten Zeiträumen dem Benutzer zur Verfügung gestellt werden. Es können immer nur jene Interaktionen ausgelöst werden, die für den aktuellen Zeitpunkt definiert sind.

> Beispiel 3 *Dem Benutzer wird (gemäß der Modellierung der Präsentation) erlaubt, ein Video während der ersten fünf Minuten anzuhalten und die Lautstärke während der Gesamtdauer der Präsentation zu ändern.*

Potentielle Interaktionen sind also ein Bestandteil der Präsentation.

5.1.3 Modellierung von interaktiven multimedialen Präsentationen

Im AMOS–System wird die Speicherung der multimedialen Daten durch ein DBVS übernommen. Um diese Daten zu einer multimedialen interaktiven

Präsentation verknüpfen zu können, sind spezielle Modellierungsmechanismen erforderlich, mit deren Hilfe ein Präsentationsplan erstellt wird.

Ein Präsentationsplan enthält neben der Beschreibung der Einzelmedien und deren gegenseitigen Abhängigkeiten außerdem noch Bedingungen (*Restriktionen*), die von der Präsentation erfüllt werden müssen. In bestimmten Fällen kann es jedoch notwendig sein, die benutzerdefinierten *Restriktionen* temporär zu verletzen, um den Fluß der Präsentation zu gewährleisten.

> BEISPIEL 4 *Ein Benutzer fordert, daß ein bestimmter Video–Clip mit* $25\frac{frames}{sec}$ *dargestellt werden soll. Falls zur Laufzeit nicht genügend Resourcen frei sind, ist es für den Anwender kaum erkennbar, wenn für wenige Sekunden die Frame–Rate auf* $22\frac{frames}{sec}$ *herunter gesetzt wird.*

5.2 Interaktionen

Bei der Präsentation der vom AMOS–System verwalteten Multimedia–Dokumente soll der Benutzer die Möglichkeit zur Einflußnahme auf den Präsentationsablauf haben. Er soll auf vielfältige Art und Weise mit dem System interagieren können und so die Präsentation (in gewissem Maße) steuern.

5.2.1 Begriffsdefinition

> DEFINITION 3 *Unter* **Interaktion** *wird hier die Modifizierung von Attributen eines oder mehrerer Medien aufgrund einer Aktion des Benutzers verstanden.*

Man muß hierbei zwischen einer *Aktion* und einer *Modifikation* unterscheiden. Eine *Aktion* könnte z.B. das Drücken eines Buttons sein, was als *Modifikation* zur Folge haben könnte, daß ein Medium angehalten wird.

> ANMERKUNG 1 *Wenn im Folgenden bei der Verwendung des Begriffes* **Interaktion** *nur die Art der Modifikation spezifiziert ist, so ist die Aktion des Benutzers nicht relevant.*

5.2.2 Einfache und komplexe Interaktionen

Im Hinblick auf Client–Server–Verteilung wird zwischen

- einfachen Interaktionen und
- komplexen Interaktionen

unterschieden.

Einfache Interaktionen werden lokal auf dem Client bearbeitet. **Komplexe Interaktionen** erfordern eine Modifikation des Präsentationsplans zur Laufzeit und müssen daher zum Server gesendet werden.

Beispiele für **einfache Interaktionen**:

- Lautstärkeregelung
- Pause

Beispiele für **komplexe Interaktionen**:

- benutzerdefinierte Interaktionen
- Auswahl einer Alternative aus einer Menge von möglichen Fortsetzungen der Präsentation

5.2.3 Interaktionen auf Gruppen von Medien

Jedem Einzelmedium $m \in \mathbb{M}$ ist über eine Abbildung β eine Menge $\mathbb{I}_m \subseteq \mathbb{I}$ aller auf m anwendbaren Interaktionen zugeordnet. $\mathbb{I}$ sei die Menge aller Interaktionen.

$$\beta : \mathbb{M} \rightarrow \mathbb{I} \tag{5.3}$$

ANMERKUNG 2 *Zulässig ist auch $\beta(m) = \varnothing$. Für solche Medien sind keine Interaktionen definiert.*

Wenn nun $\widetilde{\mathbb{M}} \subseteq \mathbb{M}$ die Menge aller vom MPM darstellbaren Medien ist, dann muß das System eine Teilmenge $\widetilde{\mathbb{I}}$ der Menge $\mathbb{I}$ von Interaktionen unterstützen:

$$\widetilde{\mathbb{I}} = \bigcup_{m \in \widetilde{\mathbb{M}}} \beta(m) \tag{5.4}$$

Die Menge aller Medien einer Präsentation p, die zum Zeitpunkt t präsentiert werden, sei $\mathbb{M}_{p_t} \subseteq \mathbb{M}_p$.

Dem Benutzer sollen Interaktionen, die sich nicht nur auf ein Einzelmedium sondern auf eine Gruppe von mehreren Medien gleichzeitig auswirken, ermöglicht werden. Dazu werden Modellierungsmechanismen bereitgestellt, um Medien $\widetilde{\mathbb{M}}_i \subseteq \mathbb{M}_{p_t}$ zu gruppieren. Dabei sei $i \in \{1, ..., n_t\}$, wobei n_t die Anzahl der definierten Gruppierungen zum Zeitpunkt t ist.

Es existieren nun zwei naheliegende Arten, Interaktionen auf $\widetilde{\mathbb{M}}_i$ zu definieren:

$$\mathbb{I}_{\min,i} := \bigcap_{m \in \widetilde{\mathbb{M}}_i} \beta(m) \tag{5.5}$$

und

$$\mathbb{I}_{\max,i} := \bigcup_{m \in \widetilde{\mathbb{M}}_i} \beta(m) \tag{5.6}$$

$\mathbb{I}_{\min,i}$ enthält alle Interaktionsmöglichkeiten, die für alle Medien der Gruppe i *zulässig* sind. $\mathbb{I}_{\max,i}$ hingegen beinhaltet die Vereinigungsmenge aller *möglichen* Interaktionsarten.

Die Verwendung von $\mathbb{I}_{\min,i}$ wäre zu restriktiv. Es ist durchaus sinnvoll, Interaktionen auf eine logisch zusammengehörige Gruppe von Medien auszuführen, wobei unter Umständen die Interaktion für manche Einzelmedien der Gruppe nicht definiert ist. Solche Interaktionen werden von den entsprechenden Medien ggfs. ignoriert.

Für jede Präsentation p werden **globale Interaktionsobjekte** bereitgestellt, deren zugeordnete Interaktionen auf die gesamte aktuelle Präsentation wirken.

5.2.4 Klassifizierung von Interaktionen

Eine genauere Analyse der Funktion (5.3) ermöglicht eine Klassifizierung der Interaktionen ähnlich wie sie in Abschnitt 2.2 vorgenommen wurde. Die hier dargelegte Gruppierung richtet sich nach der vom Anwender gewünschten Modifizierung. Interaktionstypen können mehreren Klassen angehören.

5.2.4.1 Interaktion bezüglich zeitabhängiger Attribute

Der größte Teil der Interaktionen gehört dieser Klasse an. Der Grund dafür liegt darin, daß der Benutzer in der Regel auf Veränderungen während der Präsentation reagiert und darauf Einfluß nimmt.

Interaktionen dieser Klasse sind nur auf *kontinuierliche* Medien anwendbar.

Beispiele:

- Geschwindigkeitsänderung,
- Richtungsänderung,
- Pause,
- Catchup[1] und
- Stop.

5.2.4.2 Interaktion bezüglich auditiver Attribute

Hier ist hauptsächlich die Änderung der Lautstärke relevant. Es sind aber auch Interaktionen wie das Ein/Ausschalten eines Kanals bei einem Audio in Stereo–Qualität denkbar.

[1] Wiedereingliederung eines sich im Pause–Zustand befindlichen Mediums in den aktuellen Ablauf der Präsentation

5.2.4.3 Interaktion bezüglich räumlicher Attribute

Diese Klasse der Interaktionen bezieht sich auf *visuelle* Medien. Typischerweise (jedoch nicht notwendigerweise) werden die Aktionen des Benutzers direkt auf dem zu modifizierenden Medium ausgeführt:

- Verschieben,
- Überlappen und
- Skalieren.

5.2.4.4 Typische Interaktionen

Tabelle 5.1 zeigt exemplarisch die für multimediale Präsentationen relevanten Interaktionen.

	Video	Bild	Audio	Text	Lauftext	MIDI	Animation	Simulation
Geschwindigkeit	•		•		•	•	•	•
Richtung	•		•		•	•	•	•
Lautstärke			•			•		
Positionierung*	•		•	•	•	•	•	•
Pause	•		•		•	•	•	•
Catchup	•		•		•	•	•	•
Stop	•		•		•	•	•	•
Verschiebung	•	•		•	•		•	•

*innerhalb eines Mediums

Tabelle 5.1: Zulässige Interaktionsarten

5.3 Das AMOS–Konzept für einen Multimedia Playout Manager

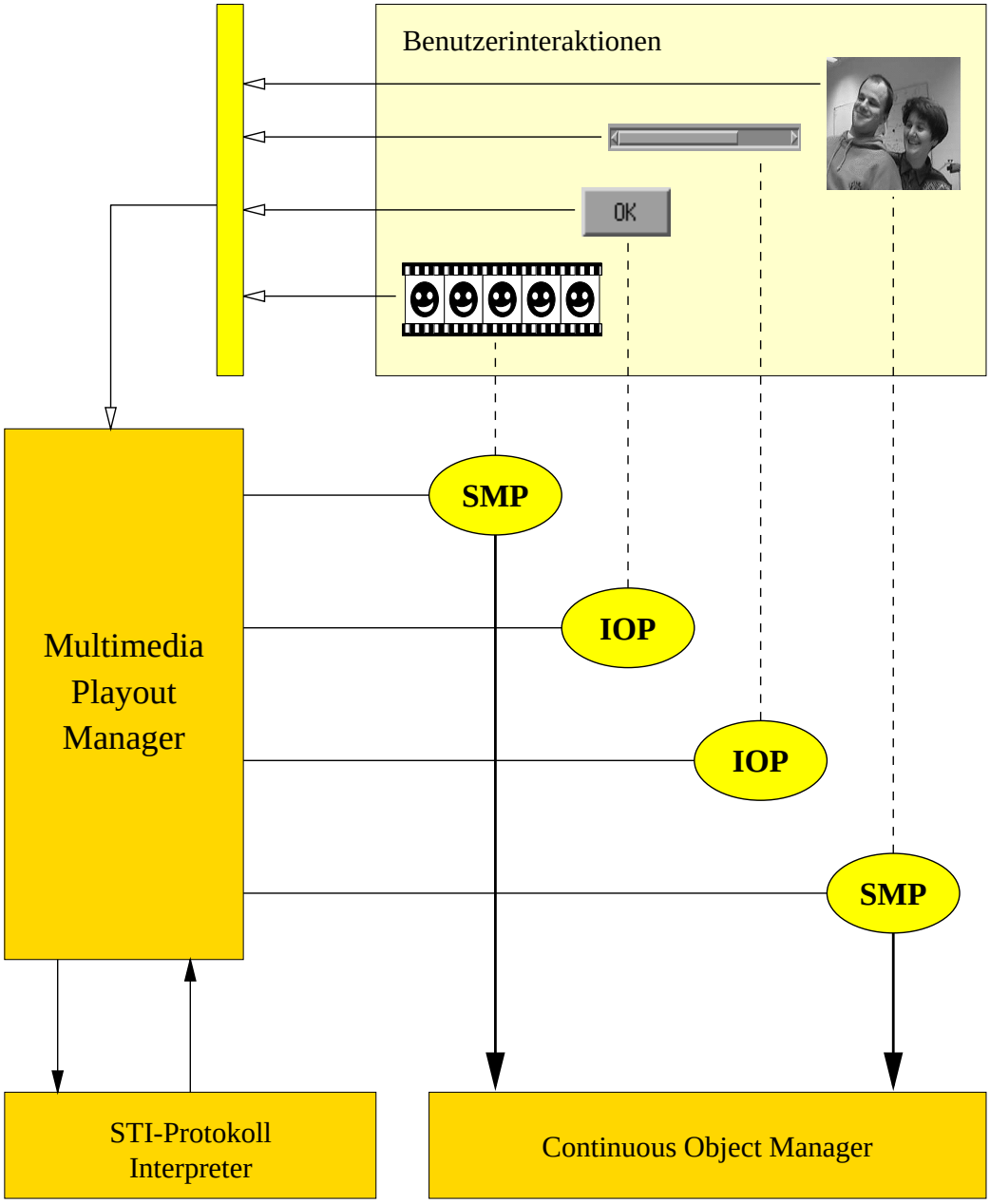

Abbildung 5.1: MPM–Architektur

Abbildung 5.1 zeigt den konzeptionellen Aufbau des MPM für das AMOS–System. Die interaktive multimediale Präsentation setzt sich aus Medien und Interaktionsobjekten zusammen. Jedes Medium wird von einem **Single Media Presenter (SMP)** und jedes Interaktionsobjekt von einem **Interaction Object Presenter (IOP)** verwaltet. Diese Module werden zur Laufzeit in beliebiger Anzahl dynamisch erzeugt.

Benutzerinteraktionen werden über die Schnittstelle IOP ausgelöst und zurück zum MPM geleitet, wo sie verarbeitet werden.

Zwischen dem Continuous Object Manager und einem SMP existiert in der Regel ein Datenkanal, über den multimediale Daten transportiert werden.

Die Komponente, von der der MPM seine Präsentationsbefehle erhält, ist der *STI–Protokoll–Interpreter*. Der MPM kommuniziert ausschließlich mit dem *STI–Protokoll–Interpreter*. Die Verbindung zum *STI–Protokoll–Interpreter* ist bidirektional. Status–Informationen und komplexe Interaktionen werden zum *STI–Protokoll–Interpreter* gesendet.

Im Folgenden werden die Komponenten des MPM, die in Abbildung 5.2 zu sehen sind, beschrieben.

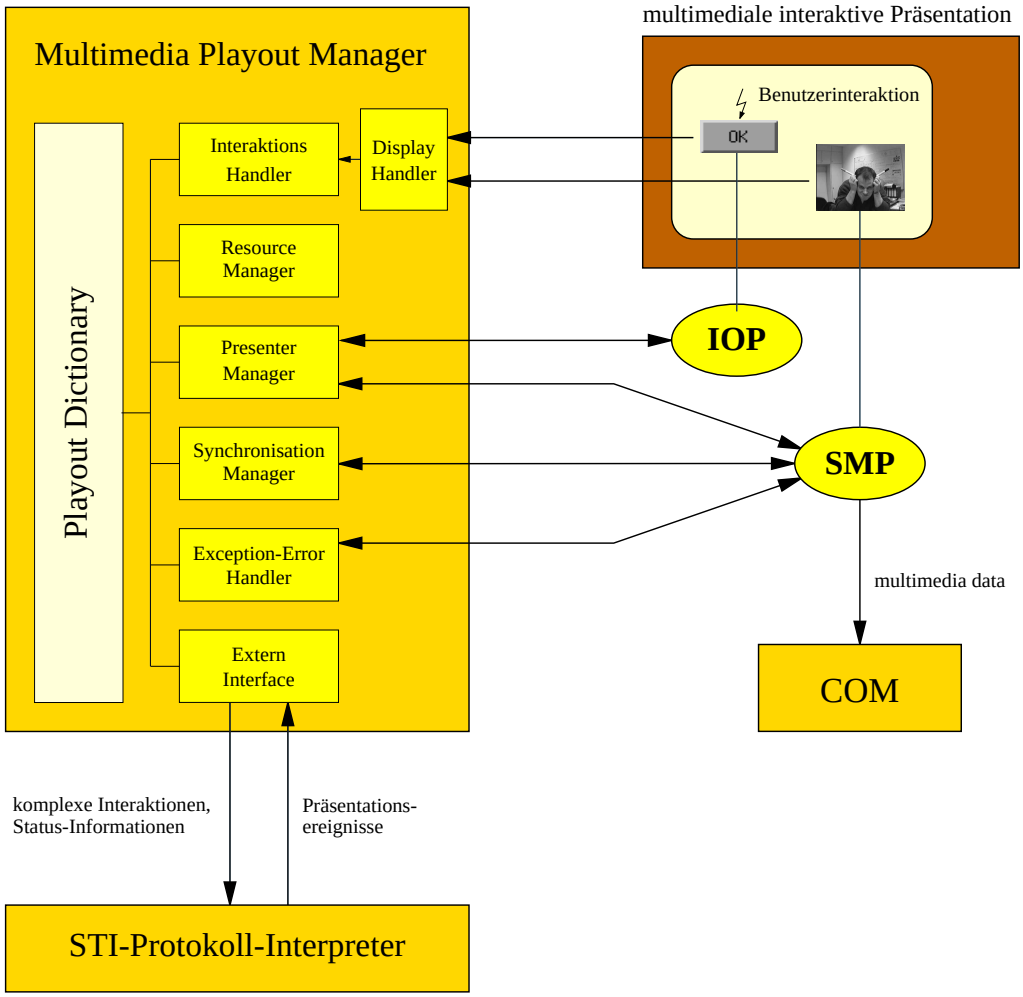

Abbildung 5.2: Detaillierte MPM–Architektur

5.3.1 Display Handler

Die Aufgabe des *Display Handlers* besteht in der

- Darstellung des gesamten visuellen Teils der Präsentation
- Weiterleitung der Benutzerinteraktionen an den *Interaktionshandler*

Der *Display Handler* kann demnach als logisches Ausgabegerät betrachtet werden, welches in der Lage ist, eine Vielzahl von visuellen Medien parallel zu präsentieren.

5.3.2 Single Media Presenters

Ein Single Media Presenter (SMP) s ist für die Präsentation genau eines Einzelmediums $m \in \mathbb{M}_p$ der multimedialen Präsentation p auf dem Ausgabegerät $\alpha(m)$ verantwortlich. Die Menge aller SMP sei $\mathbb{S}$.

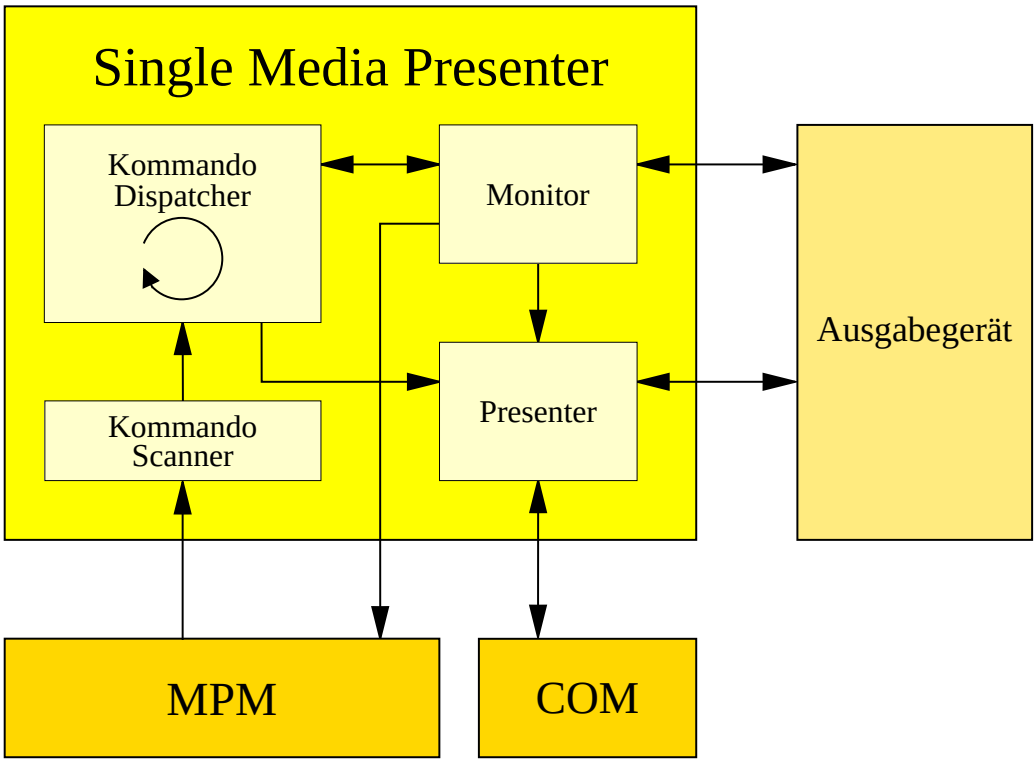

Abbildung 5.3: Single Media Presenter

Die Lebensdauer eines SMP liegt in einem bei der Modellierung des multimedialen Dokumentes definierten Intervall $t_1, ..., t_2$. Für die dynamische Erzeugung eines SMP wird eine bestimmte Zeitdauer benötigt, die abhängig vom Medium und den Systemresourcen ist. Für jedes Medium m wird daher die Vorlaufzeit $\delta(m)$ festgelegt.

Zum Zeitpunkt $t_1 - \delta(m)$ erhält der MPM vom *STI–Protokoll–Interpreter* eine Nachricht, daß ein SMP für ein Medium m vorbereitet werden soll. Der MPM verwendet diese Zeit, um ein SMP–Objekt zu erzeugen und einen Datenkanal zum COM zu initialisieren. Nach der Vorlaufzeit wird die Präsentation des Mediums m gestartet (siehe Abb. 5.4). Dies geschieht wiederum durch eine Nachricht des *STI–Protokoll–Interpreter*, der die Kontrolle über den zeitlichen Ablauf der Präsentation besitzt.

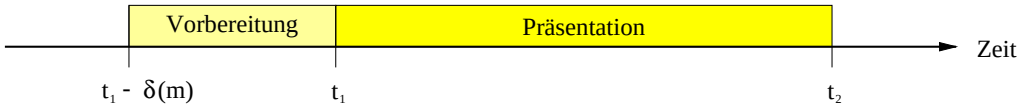

Abbildung 5.4: Lebensdauer eines SMP

Im Laufe der Präsentation nimmt der SMP Befehle vom MPM über den **Kommando–Scanner** an. Die Kommandos werden vom **Kommando–Dispatcher** interpretiert und entweder an den

- Monitor oder
- Presenter

weitergeleitet.

5.3.2.1 Monitor

Der **Monitor** überwacht den Zustand des Ausgabegerätes und reicht auf Anfrage Statusmeldungen an den MPM weiter. Falls Ausnahmebedingungen auf Seite des Ausgabegerätes auftreten, werden diese ebenfalls vom Monitor an den MPM gemeldet.

Beispiele für solche Ausnahmebedingungen:

- Gerät ist für den Benutzer gesperrt
- Gerät kann die Daten nicht schnell genug präsentieren
- Verbindung ist unterbrochen
- Präsentation korrupter Daten

Desweiteren überwacht der Monitor den Presenter und stellt den Datenfluß zum Ausgabegerät sicher.

5.3.2.2 Presenter

Der **Presenter** steuert das Ausgabegerät.

Für ein Ausgabegerät d kann es mehrere verschiedene SMP s_i geben. D.h., daß die Abbildung

$$\gamma : \mathbb{S} \rightarrow \mathbb{D} \tag{5.7}$$

surjektiv ist ($d \in \mathbb{D} \Rightarrow \exists s \in \mathbb{S} \mid \gamma(s) = d$).

5.3.3 Interaction Object Presenter

Ein IOP visualisiert und verwaltet die Komponenten der Benutzerschnittstelle, die dem Anwender Interaktionen ermöglichen. Auftretende Interaktionen werden nicht direkt vom IOP verarbeitet, da dieser kein Wissen über die Präsentation hat; statt dessen werden Interaktionen vom *Display Handler* erkannt und an den *Interaktionshandler* weitergeleitet.

Interaktionsobjekte werden bereits bei der Modellierung multimedialer Dokumente definiert und können selbst als Medien angesehen werden.

Sei $\mathbb{O}$ die Menge aller IOP, die der MPM unterstützt, und ζ eine Funktion, die jedem IOP i genau eine Interaktion $\tilde{i} \in \widetilde{\mathbb{I}}$ zuordnet.

$$\zeta : \mathbb{O} \rightarrow \widetilde{\mathbb{I}} \tag{5.8}$$

5.3.4 Playout Dictionary

Das *Playout Dictionary* ist eine globale Komponente, über die alle MPM–Komponenten Informationen über den aktuellen Zustand der Präsentation abfragen können. Dementsprechend ist der Inhalt des *Playout Dictionary* temporär und ändert sich während der Laufzeit.

Das *Playout Dictionary* verwaltet die aktuell existierenden SMP und IOP, die vom *Presenter Manager* erzeugt und auch eingetragen werden. Es enthält Informationen über deren Eigenschaften (z.B. Quality of Service (QoS) – Parameter) und Abhängigkeiten.

5.3.5 Interaktionshandler

Der *Interaktionshandler* wird vom *Display Handler* benachrichtigt, wenn eine Benutzerinteraktion durchgeführt wird. Er ist für die Durchführung der gewünschten Modifikation des Präsentationszustandes verantwortlich.

Die Nachricht vom *Display Handler* enthält unter anderem die Information, welcher IOP die Interaktion auslöste. Der *Interaktionshandler* liest nun aus dem *Playout Dictionary*, welche Interaktion dem IOP zugeordnet ist und auf welche SMP/IOP sie angewendet werden soll.

Einfache Interaktionen können lokal vom Client bearbeitet werden. Im Gegensatz dazu müssen komplexe Interaktionen vom Server behandelt werden, da sie eine Änderung des Präsentationsplans erzwingen (siehe Abschnitt 5.2.2). Abbildung 5.5 verdeutlicht die hier dargestellten Zusammenhänge und den Kontrollfluß.

5.3.6 Resource Manager

Der *Resource Manager* überwacht die verfügbaren Resourcen. Dabei wird unterschieden zwischen

- statischen
- dynamischen

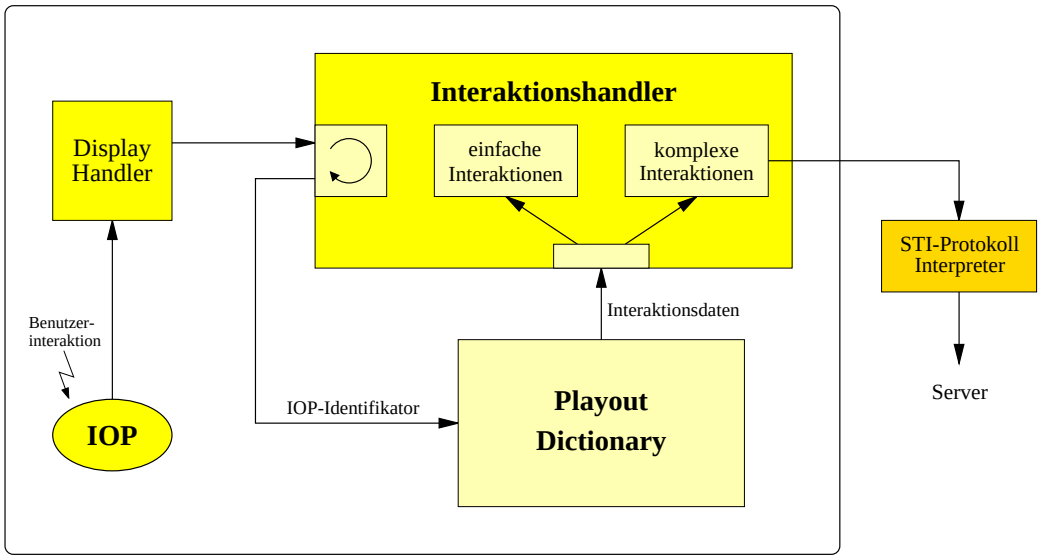

Abbildung 5.5: Interaktionshandler

Resourcen.

Beispiele für statische Resourcen:

- Farbtiefe
- Bildschirmauflösung
- Taktfrequenz der CPU

Beispiele für dynamische Resourcen:

- freier Speicher
- Auslastung der CPU
- Netzbelastung

Bevor ein SMP erzeugt wird, muß überprüft werden, ob die über QoS[2]–Parameter geforderten Resourcen verfügbar sind. Z.B. muß eine spezielle

[2] Quality of Service

Graphikkarte vorhanden sein, damit ein True–Color–Video angezeigt werden kann.

Desweiteren kann der *Synchronisationsmanager* während der Präsentation dynamische Resourcen abfragen, mit deren Wissen er die Präsentation sychronisiert.

5.3.7 Presenter Manager

Die Aufgabe des *Presenter Manager* besteht darin, alle SMP und IOP einer Präsentation zu instanziieren und kontrollieren. Der *Presenter Manager* ist die zentrale Komponente, die von weiteren Modulen verwendet wird, um das Verhalten der SMP bzw. IOP zu beeinflussen.

Folgende Module benutzen die Funktionalität des *Presenter Manager*:

- *STI–Protokoll–Interpreter*
- *Interaktionshandler*
- *Synchronisationsmanager*
- *Error Handler*

Der *STI–Protokoll–Interpreter* liefert Kommandos, die entweder bereits im Plan definiert sind, oder die aus komplexen Interaktionen resultieren. Der *Presenter Manager* erhält Befehle vom *Interaktionshandler*, falls einfache Interaktionen auftreten (siehe Abschnitt 5.3.5). Der *Synchronisationsmanager* verwendet den *Presenter Manager*, um Intramedia–Synchronisation durchzuführen. In fatalen Situationen kann der *Error Handler* bestimmte SMP oder IOP zerstören.

Die hier beschriebenen vom *Presenter Manager* ausgeführten Aktionen und deren Auslöser sind in Tabelle 5.2 beschrieben.

Auslöser	**Aktion**	**SMP/IOP**	
		obligat.	optional
Interaktionshandler	einfache Interaktionen		•
STI–Protokoll–Interpreter	Vorbereitung	•	
	Start der Präsentation	•	
	Zerstören		•
	Attributänderungen[a]		•
	Zustandsänderungen[b]		•
Synchronisationsmanager	Änderung von QoS – Parametern		•
	Zustandsänderungen		•
Error Handler	Zerstören		•

[a] z.B.: Verschieben eines Bildes, Änderung einer Button–Beschriftung, ...
[b] z.B.: Anhalten eines Videos, ...

Tabelle 5.2: Aktionen bzgl. SMP/IOP

5.3.8 Synchronisationsmanager

Ohne weitere Vorkehrungen kann der MPM lediglich Grobsynchronisation gewährleisten. Das bedeutet, daß mehrere Medien zu diskreten Zeitpunkten gleichzeitig gestartet bzw. beendet werden. Um jedoch auch Intra– und Intermedia–Synchronisation für eine multimediale Präsentation (siehe Abschnitt 5.1.2.4) gewährleisten zu können, ist eine kontinuierliche Überwachung der SMP und ggfs. deren Adaption notwendig.

5.3.9 Exception Handler

Der *Error Handler* behandelt fatale Situationen des Gesamtsystems. Beispiele für solche Zustände wurden in Abschnitt 5.3.2.1 aufgelistet. Das Auftreten eines Fehlers wird vom

- *Resource Manager* oder
- einem SMP

bemerkt, worauf der *Error Handler* vom entsprechenden Modul benachrichtigt wird.

5.3.10 Extern Interface

Das *Extern Interface* dient als Schnittstelle zwischen dem MPM und den restlichen Komponenten des AMOS–Systems. Es ist sinnvoll, solch ein zentrales Interface zur Verfügung zu stellen, da ein zu großer Overhead entstünde, wenn eine Kommunikation von externen Modulen zu MPM–internen Komponenten erlaubt wäre.

Das *Extern Interface* scannt die vom *STI–Protokoll–Interpreter* geschickten Nachrichten und leitet diese an die entsprechenden internen Komponenten weiter.

Teil II

Kapitel 6

Implementierungskonzepte

Als Implementierungskonzepte für die Realisierung des MPM wurden

- die Bedienoberfläche Motif und
- das Multithreadingkonzept

verwendet. Im Folgenden sollen beide Konzepte näher beschrieben werden.

6.1 Die Bedienoberfläche Motif

Die Visualisierung der multimedialen Dokumente wird mit der Bedienoberfläche Motif durchgeführt.

Da die Benutzeroberfläche und die allgemeine Darstellung ein wichtiger Teil der Implementierung war, werden in diesem Kapitel die Konzepte des auf das X–Window–Systems [OQL88] aufbauenden Motif–Widget–Sets vorgestellt.

Motif stellt elementare graphische Objekte wie

- Buttons (siehe Abb. 6.1)
- Schieberegler (siehe Abb. 6.2)
- Fenster

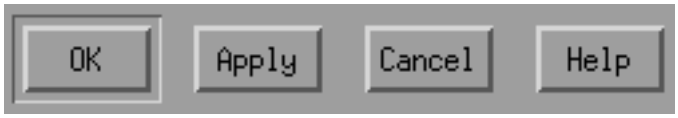

Abbildung 6.1: Button Widget

Abbildung 6.2: Scrollbar Widget

- Dialog–Anfragen
- Radio Buttons (siehe Abb. 6.4)
- Menüs
- Listen (siehe Abb. 6.5)
- Text–Objekte (siehe Abb. 6.3)

zur Verfügung. Diese Objekte werden "Widgets" genannt. Widgets werden sowohl von der Motif– als auch von der Intrinsics–Library zur Verfügung gestellt. Das Motif–Widget–Set ist seit Mitte 1989 verfügbar und wurde von der Open Software Foundation (OSF) entwickelt. Dieses Widget–Set ist dynamisch um benutzerdefinierte Widgets erweiterbar.

Ein paar der wichtigsten Widgets sind in den nachfolgenden Snapshots zu sehen, wobei diese Beispiele rein exemplarische Ausprägungen sind. Dabei gilt zu beachten, daß jedes Widget eine Menge von Attributen (Resourcen) besitzt (z.B. Zeichensatz, Breite, Höhe, Farben, ...), die u.a. das Erscheinungsbild beeinflussen.

Das mehrzeilige Text Widget (siehe Abb. 6.3) ist ein Motif–Widget, das intern verschiedene Widgets (Schieberegler, Eingabefeld) verwendet und diese zu einem komplexen Objekt verknüpft.

Bei den Radio Buttons in Abbildung 6.4 ist zu beachten, daß die unteren drei Knöpfe nicht selektierbar (sensitiv) sind, da das entsprechende Attribut auf `FALSE` gesetzt wurde.

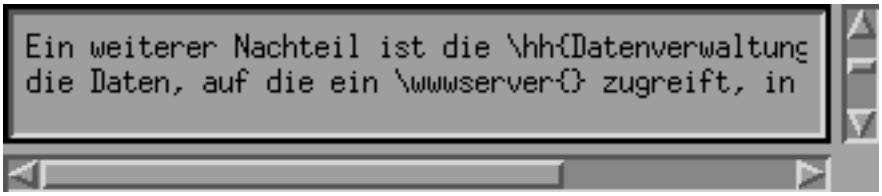

Abbildung 6.3: Text Widget

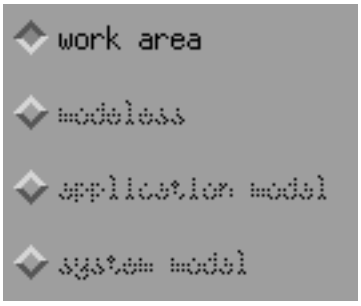

Abbildung 6.4: Radio Button Widget

Abbildung 6.5: List Widget

Von der Motif–Library sowie von der Xt–Intrinsics–Library werden geeignete Funktionen zur Verfügung gestellt, um mit den oben aufgelisteten graphischen Objekten ein Benutzer–Interface zu modellieren. Beide Bibliotheken basieren auf der X–Library, wobei die Motif–Library wesentlich mächtiger als die Xt Intrinsics–Library ist.

> ANMERKUNG 3 *In der Regel ist man von der Benutzung der technisch schwer zu handhabenden XLib bei ausschließlicher Verwendung der Xt Intrinsics nicht entbunden, was jedoch bei der Kombination Motif /Intrinsics in fast allen Anwendungen der Fall ist.*

Abbildung 6.6 zeigt die Bibliothekshierarchie, die bei einer Motif–Applikation verwendet wird (aus [Hel91]) .

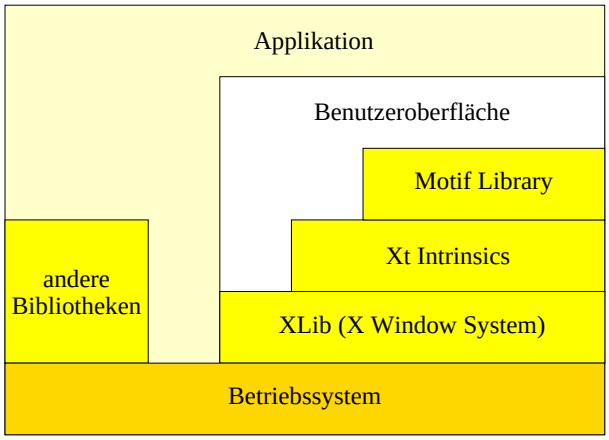

Abbildung 6.6: Bibliothekshierarchie einer Motif–Applikation

Wie man in dieser Abbildung erkennt, kann die Anwendung mit allen Ebenen des Window–Systems interagieren.

6.1.1 Programmaufbau

Konzeptionell ist eine Motif–Applikation wie in Abbildung 6.7 aufgebaut. Da bei der Realisierung des MPM nicht direkt Motif, sondern eine darauf basierende C++ Klassen–Bibliothek verwendet wurde, wird nur der Pseudo–Code angegeben.

```
1    hauptfunktion()
2    {
3      [...]
4
5      init motif;
6      create motif-objects;
7
8      start hauptschleife;
9
10     [...]
11   }
```

Abbildung 6.7: Konzeptueller Aufbau einer Motif–Applikation

Charakteristisch für Motif ist das **ereignisorientierte Arbeiten**. Bei der zunächst auszuführenden **Initialisierung** wird einerseits die Verbindung zum X–Server aufgebaut und andererseits die von Motif benötigte Strukturen initialisiert. Danach können vom Benutzer die gewünschten Widgets erzeugt werden, die jedoch erst graphisch dargestellt werden, wenn die Hauptschleife aufgerufen wird.

6.1.2 Motif–Hauptschleife

```
1    hauptschleife() {
2       WHILE TRUE {
3          wait for x-event;
4          consume x-event;
5       }
6    }
```

Abbildung 6.8: Die Motif–Hauptschleife

Die in Abbildung 6.8 gezeigte Hauptschleife ist eine Endlosschleife, in der auf X–Events reagiert wird. X–Events können z.B. durch folgende Ereignisse ausgelöst werden:

- Benutzerinteraktionen
- interne Funktionen
- "Expose–Events" (Freigabe eines bisher verdeckten Fensterbereiches)

6.1.3 Callback–Funktionen

Um den graphischen Elementen benutzerdefinierte Funktionalität verleihen zu können, existiert ein Mechanismus, mit dem den Widgets bestimmte Funktionen zugeordnet werden können. Unter bestimmten Bedingungen werden diese sog. **Callback–Funktionen** aufgerufen, unter anderem bei einem Ereignis wie das

- Drücken eines Mausbuttons,
- Loslassen eines Mausbuttons,
- Zerstören eines Widgets,
- Bewegen der Maus über ein Widget,
- Selektieren eines Listen–Elementes

> BEISPIEL 5 *Während der Präsentation eines multimedialen Dokumentes soll die Lautstärke eines auditives Mediums regelbar sein. Dazu wird ein Widget des Typs* scrollbar *(Schieberegler) erzeugt, das als Callback eine Funktion erhält, die den Wert des Schieberegler liest und entsprechend das Audio–Device ansteuert, um die Lautstärke wie gewünscht zu modifizieren.*

Innerhalb der Callbacks können weitere Widgets erzeugt werden. So kann z.B. durch den Callback, der durch das Drücken eines Buttons aktiviert wird, ein Fenster–Widget erzeugt werden, welches wieder weitere Objekte enthält.

Der Programmablauf bei Motif–Anwendungen wird nicht durchgehend durch einen sequentiellen Programmcode repräsentiert, da Funktionen blokkieren können. Dadurch kann der Programmfluß die Hauptschleife nicht mehr erreichen, und ankommende X–Events können demnach nicht verarbeitet werden. Eine solche Blockade äußert sich z.B. dadurch, daß der Fensterinhalt nicht mehr aktualisiert wird.

Im Beispiel aus Abbildung 6.9 würde das aktive Fenster während der Präsentation der Slideshow einfrieren, da das Statement `cin` blockierend ist. Auch wenn die Funktion `show-next-picture` die korrekten Modifikationen

```
1    slideshow() {
2       cin >> eingabe;
3       WHILE ( cin >> eingabe <> ende ) {
4         show-next-picture();
5       }
6    }
```

Abbildung 6.9: Beispiel einer blockierenden Eingabefunktion unter Motif

der Motif–Widgets durchführt, werden diese nicht sichtbar, da die daraus resultierenden X–Events nicht verarbeitet werden.

Bei einer korrekten Version (siehe Abb. 6.10) dieses Programmstückes wird die Kontrolle über die Slideshow über zwei Motif–Buttons gesteuert.

```
1    slideshow() {
2      picture = CREATE picture( first );
3      exit-button = CREATE button( slide-exit );
4      next-button = CREATE button( show-next-picture);
5    }
6
7    slide-exit() {
8      DELETE exit-button;
9      DELETE next-button;
10     DELETE picture;
11   }
12
13   show-next-picture() {
14     picture = slideshow( next );
15   }
```

Abbildung 6.10: Beispiel einer nicht blockierenden, korrekten Eingabefunktion

6.2 Multithreading

Unter einem Thread versteht man eine Sequenz von Ausführungsschritten, die von einem Programm abgearbeitet werden[1]. Bei herkömmlichen Programmen existiert normalerweise ein einziger Programmfluß.

In [Fan93] wird folgende Definition angegeben:

> DEFINITION 4 *Ein* **Thread** *ist ein Paar* $\langle T, T_e \rangle$*, wobei* T *ein sequentieller Instruktionsstrom und* T_e *dessen Kontext ist.*

6.2.1 Traditionelle UNIX–Prozesse

Der Ablauf eines traditionellen UNIX–Prozesses (siehe Abb. 6.11), welcher aus nur einem Thread [2] besteht (*single threaded*), bzw. der selbst als abstrakter Thread angesehen werden kann, ist stets synchron. Der Thread bewegt sich dabei durch das Programm, durch System–Funktionen, durch Kernel–Funktionen und wieder zurück zum Programm.

Der traditionelle UNIX–Prozeß besitzt einen eigenen Adreßraum, der vor anderen Prozessen geschützt ist.

6.2.2 Multithreading durch Multitasking

Applikationen, die mehrere Kontrollflüsse parallel benötigten, mußten bisher durch mehrere konkurrierende quasiparallele UNIX–Prozesse realisiert werden (siehe Abb. 6.12).

Zur Synchronisation bzw. Kommunikation solcher Prozesse sind diverse Interprozeß–Kommunikationsmechanismen notwendig. UNIX bietet dabei folgende Möglichkeiten an:

[1] Im Gegensatz zu dieser abstrakten Begriffsdefinition wird der Begriff Thread später als Teil des Betriebssystems Solaris verwendet werden.

[2] dargestellt durch eine Wellenlinie; der Pfeil markiert die Stelle innerhalb des Threads, die gerade ausgeführt wird

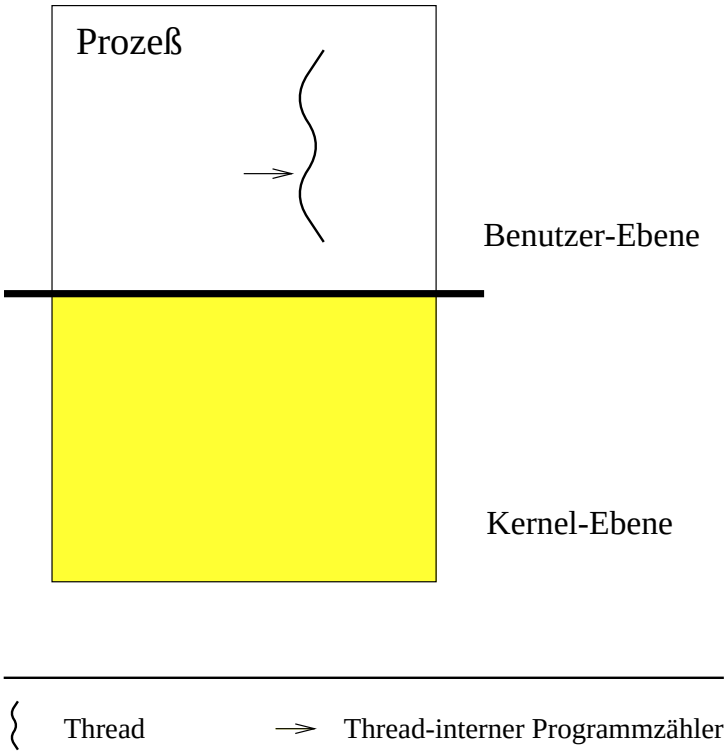

Abbildung 6.11: Traditionelles UNIX–Prozeß–Modell

- Message Queues
- Sockets
- Signals

Damit mehrere Prozesse auf denselben Speicher zugreifen können, gibt es unter UNIX Mechanismen, mit denen bestimmte Speicherbereiche (*shared memory*) für diesen Zweck reserviert werden können. Ein Beispiel einer Applikation für das in Abbildung 6.12 dargestellte Modell ist der VODAK–Server (Version 3.0), der bei jeder neuen Verbindung zu einem Client einen neuen Prozeß erzeugt, der diesem Client zugeordnet wird.

In diesem Ansatz ist der Aufwand der Kommunikation und der Synchronisation zwischen den Threads (hier: Prozesse) relativ hoch.

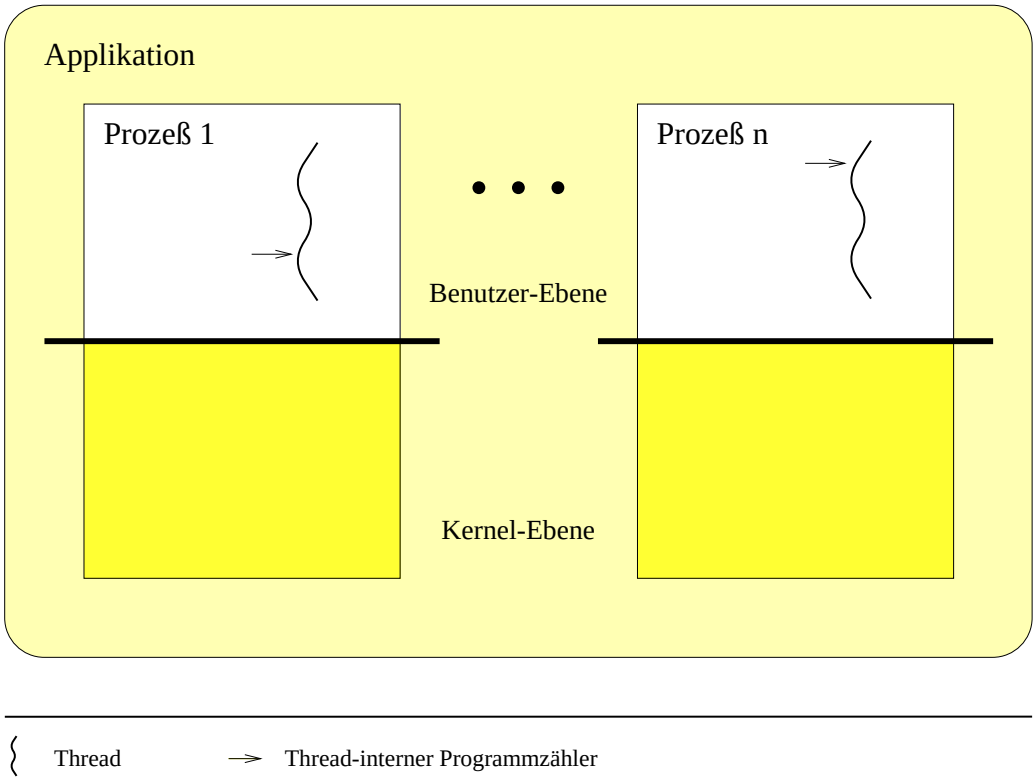

Abbildung 6.12: Aus mehreren Prozessen bestehende Applikation

6.2.3 Threads innerhalb eines Prozesses

Die auch auf diesem Gebiet fortschreitende Entwicklung brachte eine besondere Neuerung auf dem Gebiet der Threads. Dabei handelt es sich um **mehrere** Threads **innerhalb eines** Prozesses (siehe Abb. 6.13).

Hierbei wird ein Prozeß in **mehrere** Threads aufgespaltet. Die Pfeile markieren jeweils die Stelle innerhalb eines Threads, die gerade ausgeführt wird. Im allgemeinen sind die Threads außerhalb des Prozesses nicht sichtbar.

Dieser Ansatz bringt gegenüber herkömmlichen *single threaded* Imple-

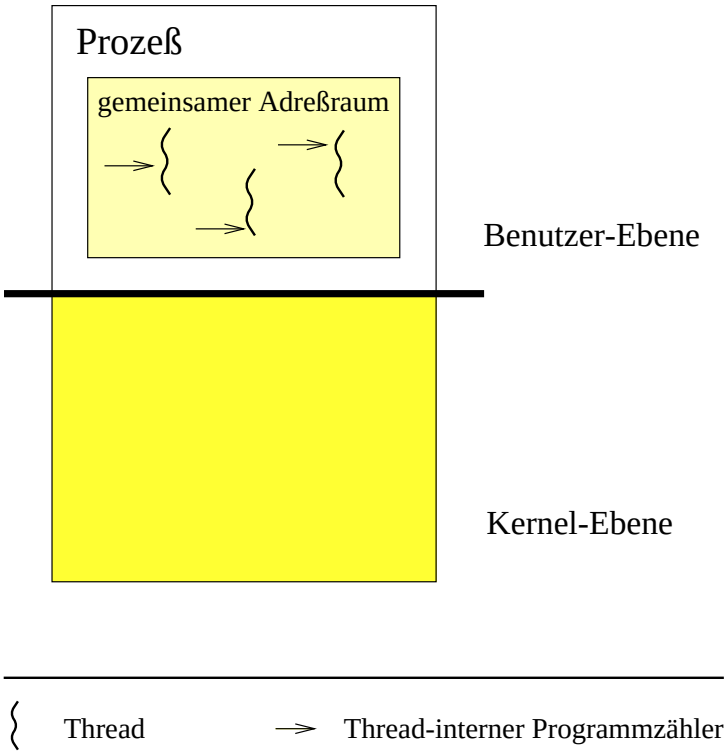

Abbildung 6.13: Multithread–Modell

mentierungen viele Vorteile:

- einfachere Synchronisation
- Kapselung der Threads
- Quasiparallelität bei nur einem Prozessor (*virtual concurrency*)
- ...

6.2.3.1 Synchronisation

Die Kommunikation zwischen den Threads gestaltet sich relativ einfach. Die Standard UNIX IPC[3]–Mechanismen (siehe auch 6.2.2) lassen sich neben weiteren thread–spezifischen Techniken auch auf Threads anwenden. Threads,

[3]Interprozeß–Kommunikation

die zu dem selben Prozeß gehören, teilen sich den selben Adreßraum. Damit keine Konflikte entstehen, wenn Threads auf das shared memory zugreifen, werden vom Betriebssystem entsprechende Mittel zur Synchronisation bereitgestellt. Es existieren folgende Arten von Synchronisationsvariablen [Sun93b]:

- mutual exclusion (mutex) locks
- condition variables
- semaphoren
- read/write locks

Selbst Threads, die sich in unterschiedlichen Prozessen befinden, sind synchronisierbar, obwohl sie gegenseitig nicht sichtbar sind. Dazu müssen die Synchronisationsvariablen im, den beiden Prozessen bekannten, shared memory oder in Files abgelegt werden (siehe Abb. 6.14).

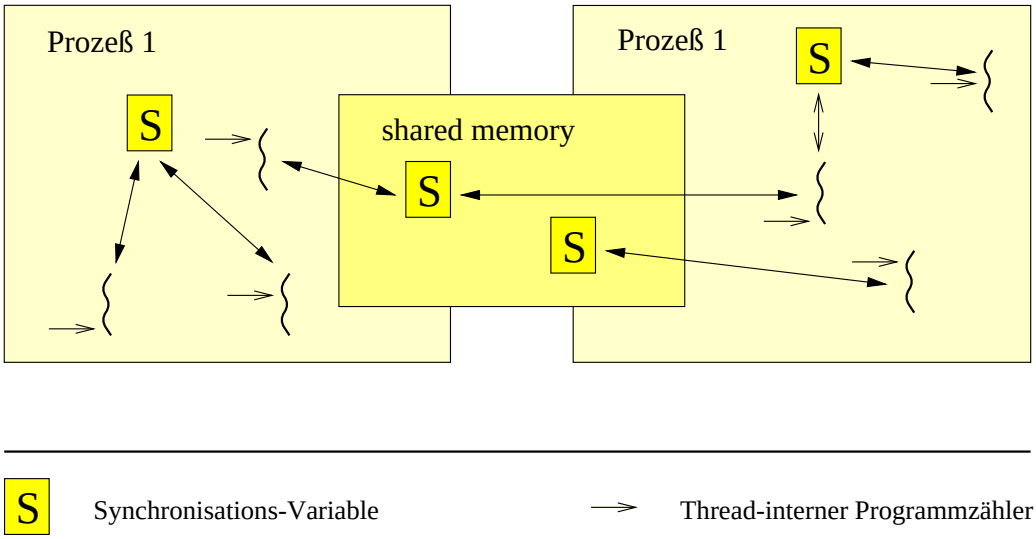

Abbildung 6.14: Synchronisations–Variablen

Threads können selbst weitere Threads dynamisch erzeugen. Das Scheduling wird entweder innerhalb des Prozesses oder vom Kernel durchgeführt.

6.2.3.2 mt–safe Funktionen

> DEFINITION 5 *Eine Funktion, die* **simultan** *von mehreren Threads ausgeführt werden kann, ohne daß Konflikte auftreten, wird* **mt–safe** *(oder auch reentrant) genannt.*

Es gilt zu beachten, daß nicht alle Betriebssystem–Funktionen *mt–safe* sind. Darüberhinaus sind auch die meisten frei verfügbaren Libraries nicht innerhalb von Threads verwendbar.

6.2.3.3 Thread–spezifische Daten

Obwohl Threads den Datenbereich und den Instruktionsbereich mit anderen Threads desselben Prozesses teilen, sind jedem Thread folgende private Bereiche zugeordnet:

- Programmzähler
- lokale Variablen
- Stack
- Signal–Maske
- Rücksprungadressen

6.2.3.4 Zustände von Threads

Zur Laufzeit befindet sich ein Thread in einem der Zustände aktiv, bereit, blockiert oder zombie. Die Übergänge zwischen diesen Zuständen und die Ereignisse, die diese auslösen, werden in Abb.6.15 gezeigt.

Threads, die sich im Zustand aktiv befinden, werden gerade bearbeitet. Bei Vorhandensein von einer CPU kann immer nur ein Thread zur selben Zeit aktiv sein. Echte Parallelität wird erreicht, indem dem Prozeß mehrere CPUs zur Verfügung gestellt werden.

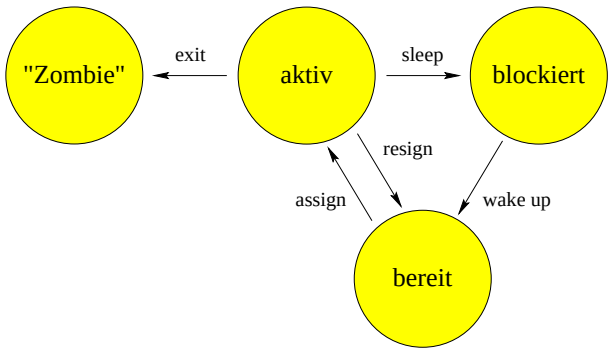

Abbildung 6.15: Zustände von Threads

Threads, die zwar ablaufbereit sind, für die jedoch noch keine freie CPU zur Verfügung steht, befinden sich im Zustand bereit. Solche Threads befinden sich in einer Warteschlange.

Im Zustand blockiert warten Threads auf bestimmte Ereignisse bzw. Bedingungen wie z.B. auf das Freiwerden einer Resource.

Zombie–Threads sind Threads, die beendet wurden, deren Resourcen aber noch nicht freigegeben wurden.

6.2.3.5 Lightweight–Prozesse

Threads stellen die Schnittstelle zwischen Programmierer und *Multithreading* dar. Technisch ist ein Thread so implementiert, daß er auf einer darunterliegenden virtuellen CPU basiert, die Lightweight–Prozeß (LWP) genannt wird. Eine detailliertere Darstellung ist in Abbildung 6.16 [Sun93a] zu sehen. LWPs stellen die Verbindung zwischen User–Level–Threads und dem Kernel dar.

Ein LWP ist eine systemweit gültige Resource, die für den Kernel und außerhalb der Prozesse sichtbar ist. Wie man sieht, werden Threads auf der Benutzer–Ebene an LWPs gebunden. Dies kann entweder direkt geschehen, indem einem Thread genau ein LWP zugeordnet wird, oder durch vorheriges Scheduling, falls mehrere Threads an eine oder mehrere LWPs gebunden werden.

Jeder LWP besitzt einen Kernel–Level–Thread, über den die Verbindung zur CPU hergestellt wird. Kernel–Level–Threads können fest an CPUs ge-

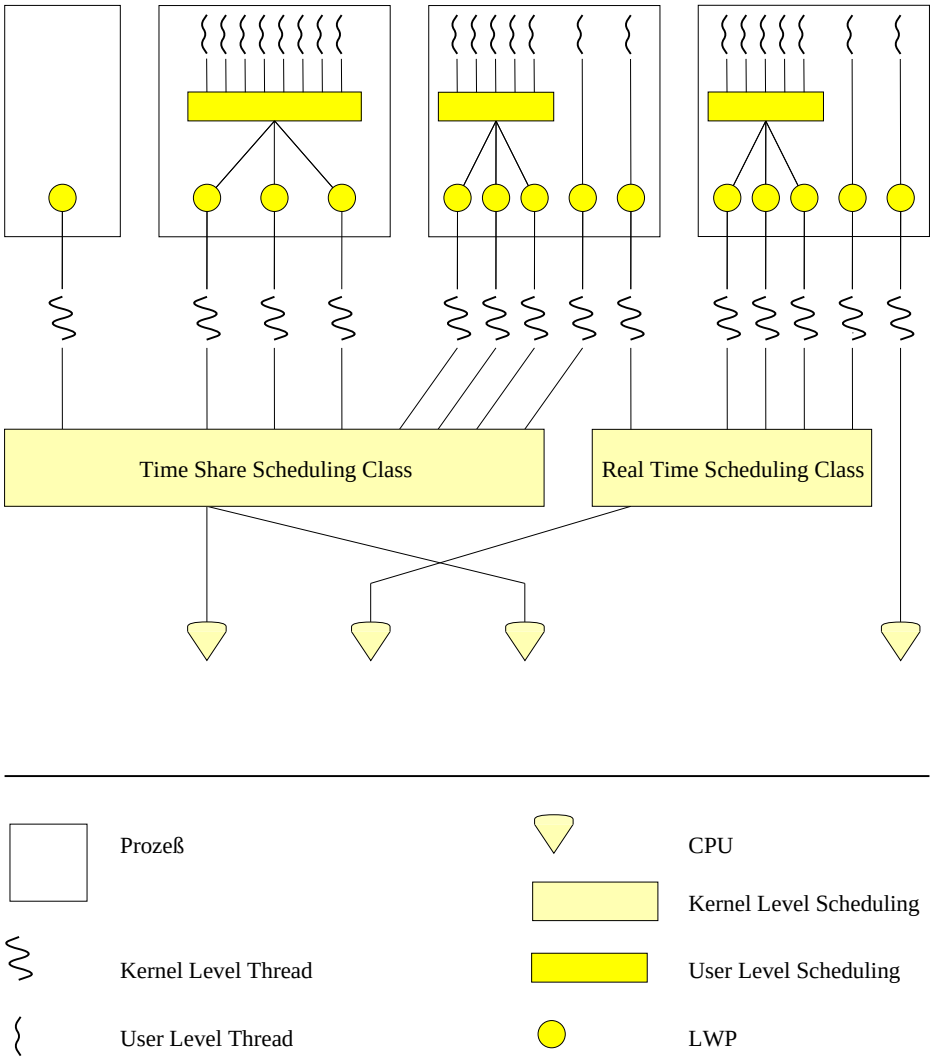

Abbildung 6.16: Auf LWP basierende Threads

bunden werden oder über das vom Kernel durchgeführte Scheduling an die verfügbaren CPUs verteilt werden. Sie sind unabhängig von den User–Level–Threads.

6.2.4 Vorteile durch Multithreading

Die Verwendung des *Multithreading*–Konzeps bringt wie schon in Abschnitt 6.2.3 schon angesprochen, verschiedene Vorteile. So kann die im Programm inhärente Parallelität herausfaktorisiert und in einzelne Threads gekapselt werden.

Die vorhandene Hardware wird effektiver ausgenutzt als bei allen bisherigen Methoden. Traditionelle UNIX–Prozesse können dahingegen auch bei einer Mehr–Prozessor–Maschine tatsächlich nur eine CPU ausnutzen.

Applikationen, die *Multithreading* benutzen, sind natürlich auch auf Ein–Prozessor–Maschinen lauffähig. Selbst auf solcher Plattformen ist es empfehlenswert, *Multithreading* zu verwenden, denn Befehle, die vorher sequentiell abgearbeitet werden mußten, können nun durch das Kernel– bzw. User–Scheduling effektiver von der CPU bearbeitet werden (*virtual concurrency*).

Desweiteren sind Threads bei I/O–intensiven Prozessen sehr vorteilhaft. Eine Applikation kann z.B. mehrere **remote procedure calls** (RPCs) an verschiedene unabhängige Hosts senden, um mehrere Ergebnisse zu erhalten. In einem Programm mit nur einem einzigen Thread mußte dies sequentiell geschehen. Nach jedem RPC müßte auf das Ergebnis gewartet (siehe Abb. 6.17) und so wertvolle CPU–Zeit vergeudet werden.

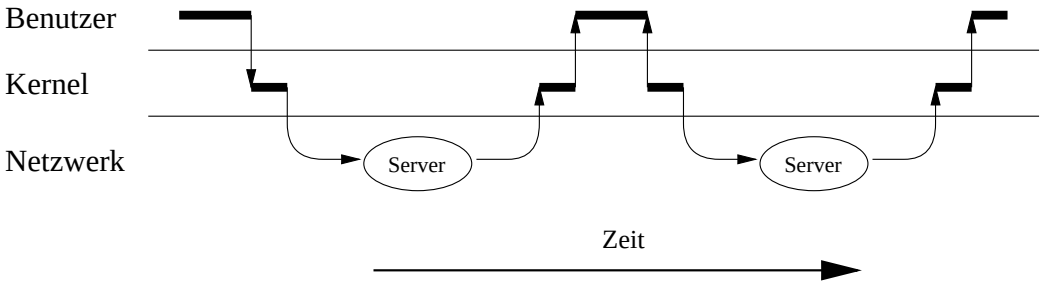

Abbildung 6.17: RPCs mit einem Thread

Eine Version dieses Programms mit mehreren Threads erlaubt, daß gleich-

zeitig auf mehrere Resultate gewartet werden kann (siehe Abb. 6.18) und in dieser Zeit die Anfragen parallel bearbeitet werden können.

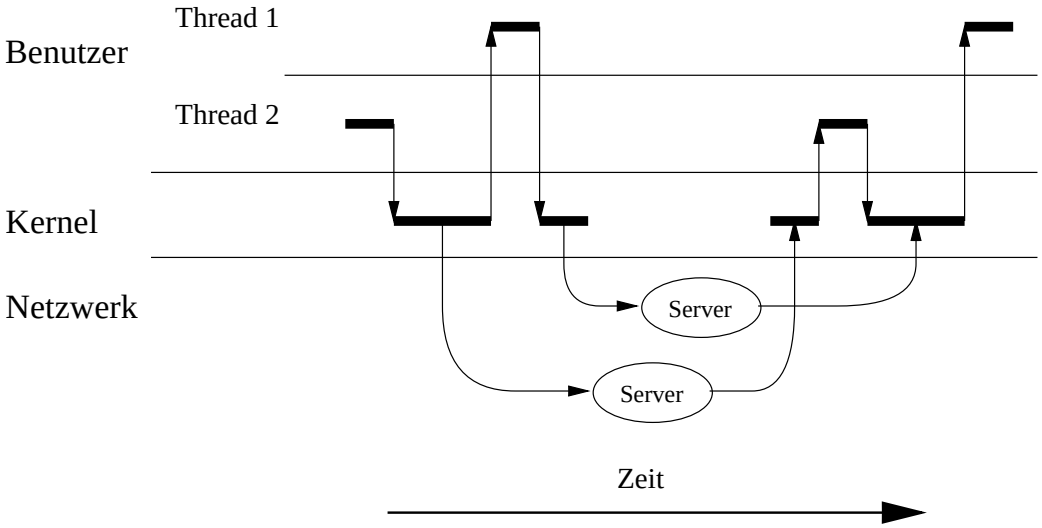

Abbildung 6.18: RPCs mit mehreren Threads

Die Kapselung voneinander unabhängiger Teile einer Applikation erlaubt es außerdem, den Quellcode gut zu strukturieren, und damit die allseits bekannten Probleme mit Wartung, Lesbarkeit und Updating leichter zu bewältigen.

Ein weiterer Vorteil besteht darin, daß Applikationen seltener zum Blokkieren gezwungen werden. Selbst während ein Thread blockiert ist, kann der Rest der Anwendung weiterhin aktiv sein.

Kapitel 7

Implementierung

In diesem Kapitel werden zunächst einige allgemeine Aspekte der Implementierung beleuchtet. Danach werden die einzelnen Module (C++ Klassen), die der Implementierung zugrunde liegen, beschrieben.

7.1 Entwicklungsumgebung

Der MPM sowie der Emulator des *STI–Protokoll–Interpreter* wurde in der Programmiersprache C++ entwickelt [Lip89]. Als Entwicklungsplattform diente eine Sparc Station 20 mit 4 Prozessoren der Firma SUN mit dem Betriebssystem SunOS 5.3 bzw. SunOS 5.4. Desweiteren fand das CASE–Tool **sniff** Einsatz.

7.1.1 Solaris

Der Begriff "Solaris" wird fälschlicherweise häufig als Synonym für das zugehörige Betriebssystem SunOS 5.x verstanden. Solaris 2.x setzt sich hingegen aus dem Window–System **OpenWindows 3.x** und dem Betriebssystem **SunOS 5.x** zusammen (siehe Abb. 7.1).

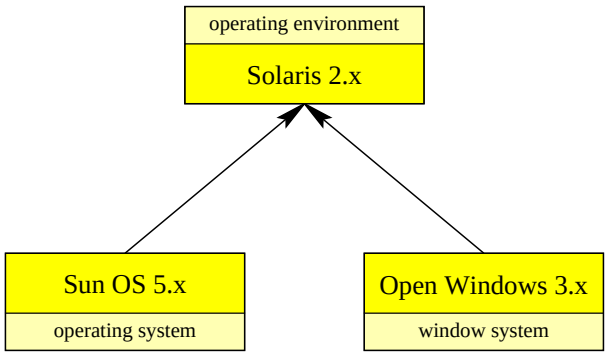

Abbildung 7.1: Komponenten von Solaris 2.x

7.1.2 C++ Compiler

Der unter Solaris verwendete C++ Compiler hat die Versionsnummer 4.0.1. Dieser Compiler erzeugt im Gegensatz zu seinen Vorgängern direkt ausführbaren Code. Frühere Versionen generierten zunächst C–Code, der dann von einem C–Compiler übersetzt werden mußte. C++ 4.0.1 kann daher effizientere ausführbare Programme erzeugen.

7.1.2.1 Exception Handling

Der C++ 4.0.1 Compiler unterstützt das kürzlich in den C++ Standard integrierte *Exception Handling*. Die Fehlerbehandlung des MPM macht sich dieses Konzept zunutze.

7.1.2.2 Multithreading

Unter Solaris 2.x stehen eine Vielzahl von Standard–Bibliotheken zur Verfügung, wobei die meisten Funktionen *mt–safe* (siehe Abschnitt 6.2.3.2) sind. In entsprechenden Dokumentationen (Answerbook, man–pages, ...) wird darüber Auskunft gegeben, ob eine Funktion *mt–safe* ist oder nicht.

Ein Modul einer Applikation, die *Multithreading* verwendet, muß mit der Option `-mt` übersetzt werden. Diese Option übergibt `-D_REENTRANT` an den Präprozessor und `-lthread` an den Linker. Im folgenden Beispiel werden zwei

Compileraufrufe gezeigt, die dasselbe bewirken.

```
fimbabahn:> CC -mt -o dings dings.C -lc
fimbabahn:> CC -D_REENTRANT -o dings dings.C -lthread -lc
```

7.2 Benutzung von mt–unsafe Funktionen

Es existieren zwei Arten von Threads unter Solaris:

- Root–Threads und
- externe Threads.

Der **Root–Thread** nimmt insofern eine Sonderstellung ein, als er sowohl *mt–safe* als auch *mt–unsafe* Funktionen ausführen kann. Die **Thread Extension Library** [WM94] macht sich diesen Umstand zunutze und stellt folgende Funktionen zur Verfügung, durch die Funktionsaufrufe (mittels einer Pipe) an den Root–Thread transportiert und dort ausgeführt werden:

- `thre_proxy_init()` (wird einmal aufgerufen, um die **proxy pipe** zu initialisieren)
- `thre_proxy()` (wird von externen Threads aufgerufen, um eine Anfrage an den Root–Thread zu senden)
- `_thre_proxy_readQ()` (wird vom Root–Thread benutzt, um die Anfragen von externen Threads zu erhalten und um sie auszuführen)

Dadurch geht der Vorteil der echten parallelen Ausführung von verschiedenen Threads verloren. Das Problem ist jedoch gekapselt, und wenn eine *mt–safe* Version der Funktion vorhanden ist, die an den Root–Thread geschickt wurde, ist die Änderung im Programmcode minimal.

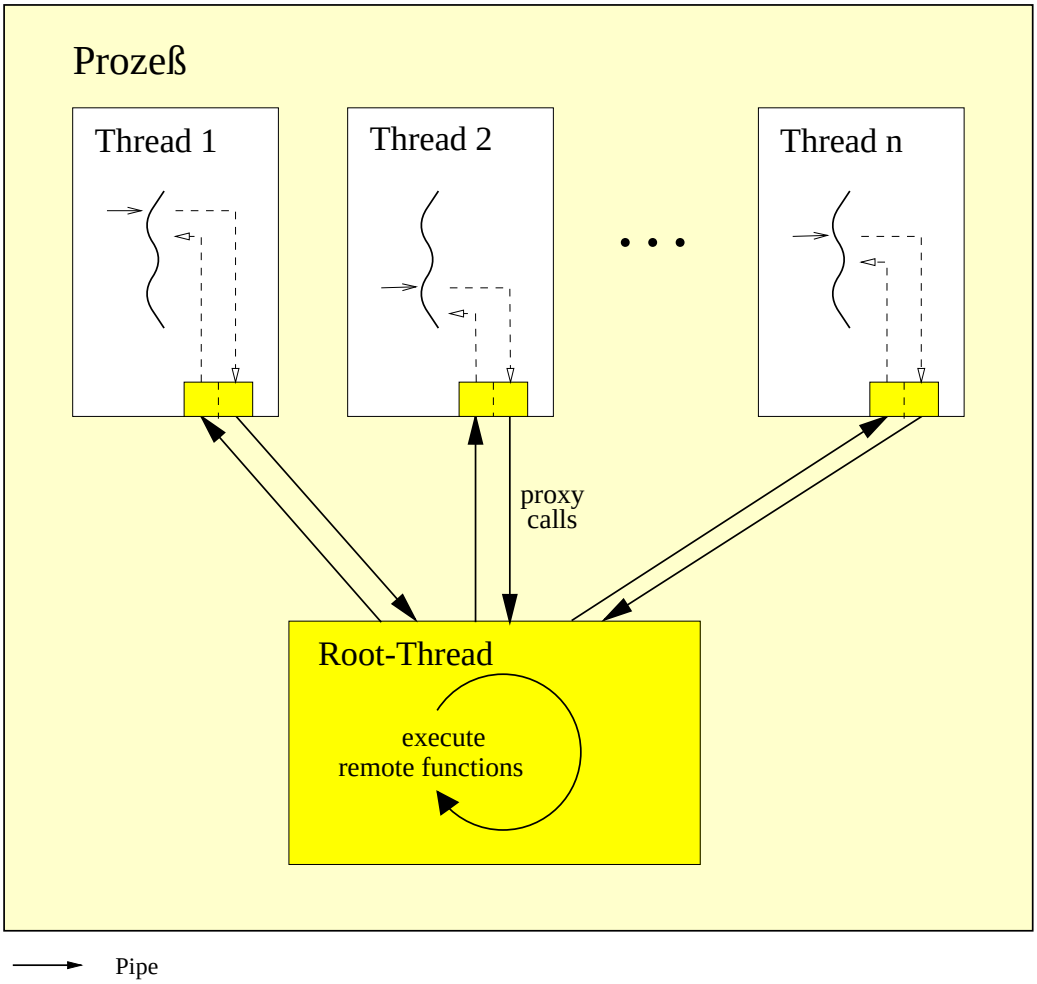

Abbildung 7.2: Ausführung von mt–unsafe Funktionen

7.3 Auf Motif basierende C++ Klassen

Bei der Implementierung des MPM fiel die Wahl der Oberfläche auf die vom Autor dieser Diplomarbeit im Rahmen einer früheren Studienarbeit entwickelten objektorientierten Klassenbibliothek. Diese Klassenbibliothek sei im Folgenden *SJ–Motif* genannt. *SJ–Motif* kapselt Motif–Widgets in C++ Klassen und stellt dem Programmierer eine bequemes Werkzeug zur Oberflächenprogrammierung zur Verfügung.

SJ–Motif ist nicht getrennt von Motif zu sehen, sondern eher als eine Art Makrosprache. Daher bezieht sich die folgende Auflistung der Vor– und Nachteile der Verwendung von *SJ–Motif* hauptsächlich auf Eigenschaften von Motif, die auch für *SJ–Motif* gelten.

Vorteile

Standard. Motif hat sich als Standard–Bibliothek für X–Windows herauskristallisiert. Dadurch ist es auf vielen Plattformen verfügbar.

Verwendung von Motif und *SJ–Motif*. Motif–Pakete können leicht in *SJ–Motif*–Klassen gekapselt werden. Es ist außerdem möglich, *SJ–Motif* und Motif zusammen in einer Applikation zu verwenden. Sowohl *SJ–Motif*–Klassen können auf Motif–Resourcen zugreifen wie auch umgekehrt.

Literatur. Es existiert eine Vielzahl an Literatur über Motif.

freie Software. Viele Bibliotheken, die auf Motif basieren, sind frei verfuegbar. So wurde z.B. der MPEG[1]–SMP mit Hilfe einer PD[2]–Bibliothek realisiert. Desweiteren sind alle auf die X–Lib und X–Intrinsics–Lib basierenden Pakete leicht in *SJ–Motif* integrierbar.

X11R6 ist *mt–safe*. Die X–Lib Release 6 und die entsprechende X–Intrinsics Version ist *mt–safe*. Dadurch können Teile des MPM *mt–safe* realisiert werden, die **nur** diese beiden Bibliotheken verwenden dürfen.

Nachteile

***mt–unsafe*.** Motif ist nicht *mt–safe*. Daher mußte zunächst der in Abschnitt 7.2 beschrieben Mechanismus in die *SJ–Motif*–Bibliothek integriert werden.

freie Software nicht *mt–safe*. Obwohl die Releasenummer 6 von X–Lib und die entsprechende X–Intrinsics Version *mt–safe* ist, trifft dies nicht für die meisten darauf basierenden freie Widgets bzw. Bibliotheken zu,

[1] Moving Pictures Experts Group
[2] Public Domain

da diese meistens zu Zeiten entwickelt wurden, in denen das *Multithreading*–Konzept noch nicht aktuell war. Eine Konvertierung solcher Pakete zu *mt–safe* Paketen kommt einer Re–Implementierung gleich. Es besteht jedoch immer die Möglichkeit, den in Abschnitt 7.2 beschriebenen Mechanismus zu wählen.

Motif ist nicht frei. Obwohl es viele freie Applikationen und Bibliotheken gibt, die auf Motif basieren, ist Motif selbst nicht PD.

SJ–Motif fand bereits Anwendung in dem V^3–Video–Server (siehe Abschnitt 4.2.1). Im Rahmen dieses Projektes wurde *SJ–Motif* um eine Vielzahl von Klassen erweitert.

7.3.1 Klassenhierarchie

Die C++ Klassenhierarchie ist in Abbildung 7.3 dargestellt. Die Klassennamen entsprechen in der Regel den Namen von Motif–Widgets.

7.3.2 multithread–fähige Erweiterung

Damit die *SJ–Motif*–Library *mt–safe* wird, mußten die gekapselten Motif–Aufrufe über **proxy calls** an den Root–Thread weitergeleitet werden. Bisher existierte lediglich eine Lösung für XView.

Im Rahmen dieser Diplomarbeit wurde eine Lösung für Motif entwickelt, die nun auch Bestandteil der **Thread Extension Library** ist. Abbildung 7.4 zeigt die Verwendung der **proxy calls** in Verbindung mit der Motif–Funktion `XmCreateLabel`.

7.4 Kommunikation zwischen den MPM–Komponenten

Alle Komponenten des MPM wurden durch Threads realisiert. Obwohl die einzelnen MPM–Komponente zum Teil sehr unterschiedliche Aufgaben zu verrichten haben, ist es sinnvoll, einen gemeinsamen Kommunikationsmechanismus zu verwenden.

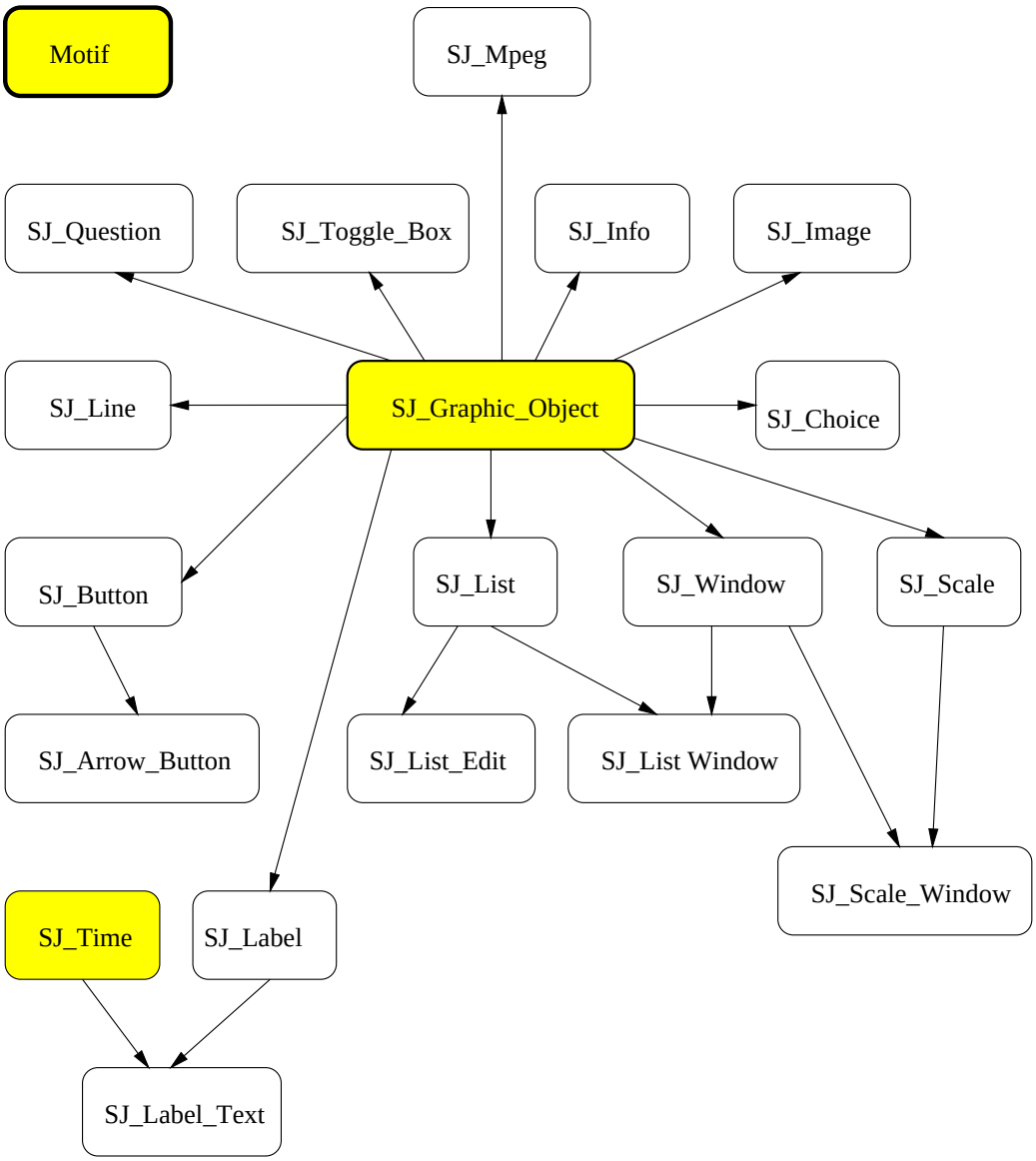

Abbildung 7.3: SJ–Motif Klassenhierarchie

```
1   #include <thrext.h>     // Deklarationen der Thread Extension Library
2
3   root_thread()
4   {
5
6     [...]
7
8     int mt_pipe[2];       // Input/Output-Pipe
9
10    pipe(mt_pipe);        // Erzeuge Pipes
11
12    //========================================================================
13    // Pipes werden dem proxy-system bekanntgegeben
14    //========================================================================
15    thre_proxy_init(mt_pipe[1], mt_pipe[0]);
16
17    //========================================================================
18    // Initialisierung von Xt/Motif
19    //========================================================================
20    toplevel = XtVaAppInitialize(&context,
21                            class_name, NULL, 0, &argc,
22                            argv, NULL, NULL);
23
24    //========================================================================
25    // XtAppAddInput fuegt dem X-Prozess eine neue Event Quelle zu.
26    // In diesem Falle ist es die mt_pipe. Falls Daten an dieser Quelle
27    // anliegen wird _thre_proxy_readQ aufgerufen (siehe man XtAppAddInput).
28    //========================================================================
29    XtAppAddInput(context, mt_pipe[0],
30                  (XtPointer)XtInputReadMask,
31                  (XtInputCallbackProc)_thre_proxy_readQ,
32                  NULL);
33
34    [...]
35
36    XtAppMainLoop(context);  // starten der Hauptschleife
37
38    [...]
39  }
40
41
42  extern_thread()
43  {
44    [...]
45
46    //========================================================================
47    // Ein Label-Widget wird erzeugt. Dafuer wird die Funktion
48    // thre_proxy mit XmCreateLabel und entsprechenden Attributen
49    // als Parameter aufgerufen.
50    //========================================================================
51    Widget label = (Widget)thre_proxy(MTWAIT, 5,
52                                (Proxy_Proc)XmCreateLabel,
53                                (char *)parent, name, al,
54                                ac, NULL);
55
56    [...]
57  }
```

Abbildung 7.4: Anwendung der Thread Extension Library auf Motif

Dadurch werden folgende Vorteile erzielt:

- konsistentes Kommunikationssystem
- einfache Integration neuer MPM–Komponenten, da ein Thread auf ein und die selbe Weise mit verschiedenen weiteren Threads kommuniziert.

In der Implementierung wurden hierfür *Messagequeues* verwendet. Damit wird eine ereignisgesteuerte Kommunikation erreicht. Jeder MPM–Komponente wird eine *Messagequeue* zugeordnet.

Eine MPM–Komponente befindet sich normalerweise im Wartezustand. In diesen Zustand gelangt sie durch einen blockierenden `receive`–Aufruf der ihr zugehörigen *Messagequeue*. Wenn die MPM–Komponente Daten über die *Messagequeue* erhält, werden diese als Befehl interpretiert und ausgeführt. Danach geht die MPM–Komponente wieder in den Wartezustand über (siehe Abb. 7.5).

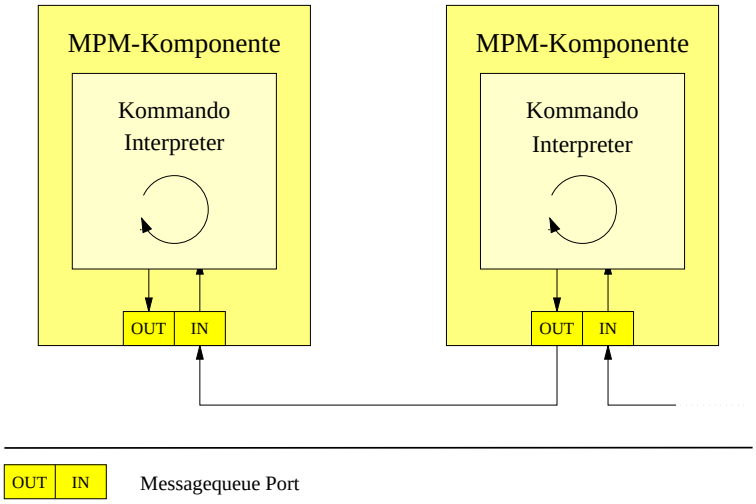

Abbildung 7.5: Kommunikation zwischen MPM–Komponenten

Dieser Mechanismus wurde generisch realisiert und kann von neuen MPM–Komponenten geerbt werden. Lediglich ein neuer Befehlssatz sowie dessen Ausführung muß implementiert werden.

7.5 Oberfläche

Im Folgenden wird anhand von exemplarischen Snapshots die Oberfläche des MPM gezeigt.

Nach dem Starten des Systems werden folgende Fenster geöffnet (siehe Abb. 7.6):

- MPM–Präsentations–Fenster
- MPM–Environment–Fenster
- MPM–Debug–Fenster
- "Send"–Fenster der Emulation des *STI–Protokoll–Interpreter*
- "Receive"–Fenster der Emulation des *STI–Protokoll–Interpreter*

Alle Fenster außer dem Präsentations–Fenster dienen zum Testen des MPM. Sobald der MPM in das AMOS–System integriert wird, sind diese Fentster überflüssig.

7.5.1 MPM–Präsentations–Fenster

Im Präsentations–Fenster werden die visuellen Teile der multimedialen Präsentationen dargestellt. Aus Gründen der Übersichtlichkeit werden alle visuellen Medien in einem einzigen Fenster angezeigt. Dennoch besteht die Möglichkeit, durch einen speziellen SMP externe Viewer zu verwalten, auf die jedoch, wie auch beim WWW, vom AMOS–System kein Einfluß genommen werden kann.

Abbildung 7.7 zeigt einen Ausschnitt aus einer Schach–Präsentation. Zu diesem Zeitpunkt sind sechs Text–SMP, zwei Image–SMP, ein Chess–SMP und ein Audio–SMP aktiv. Der Chess–SMP [Koh95] ist ein applikationsspezifischer Presenter, der Brettkoordinaten als Eingabedaten erhält und selbständig Schachzüge visualisiert.

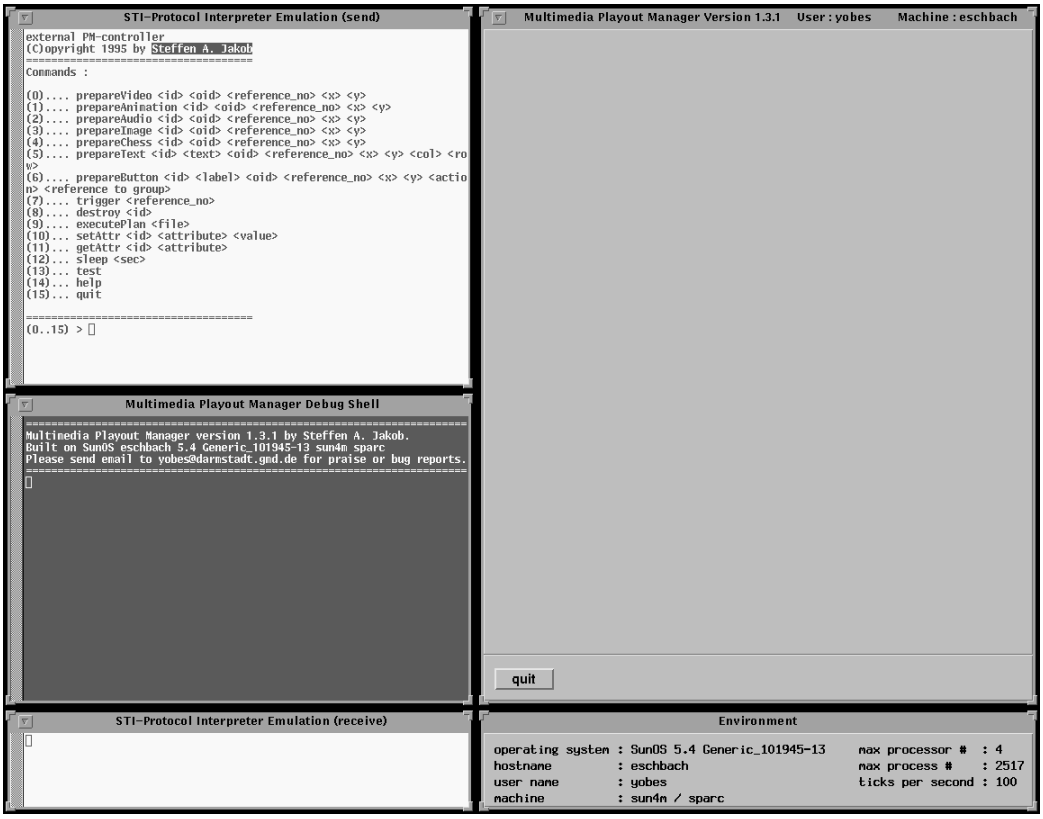

Abbildung 7.6: Oberfläche des MPM

7.5.2 MPM–Environment–Fenster

Das Environment–Fenster (siehe Abb. 7.8) visualisiert statische Resourcen, die von dem *Resource Manager* geliefert werden.

7.5.3 MPM–Debug–Fenster

Im Debug–Fenster (siehe Abb. 7.9) werden vom MPM empfangene Kommandos sowie Informationen über interne Vorgänge angezeigt.

Abbildung 7.7: MPM Fenster

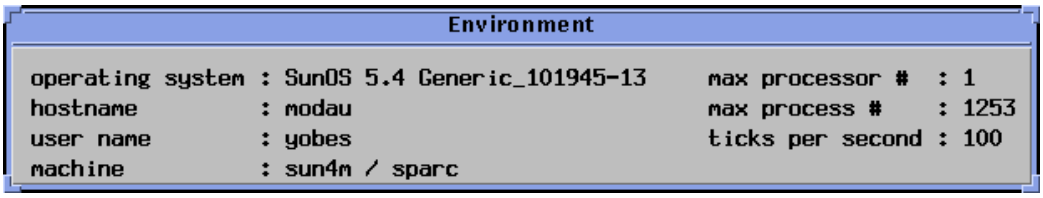

Abbildung 7.8: Environment Fenster

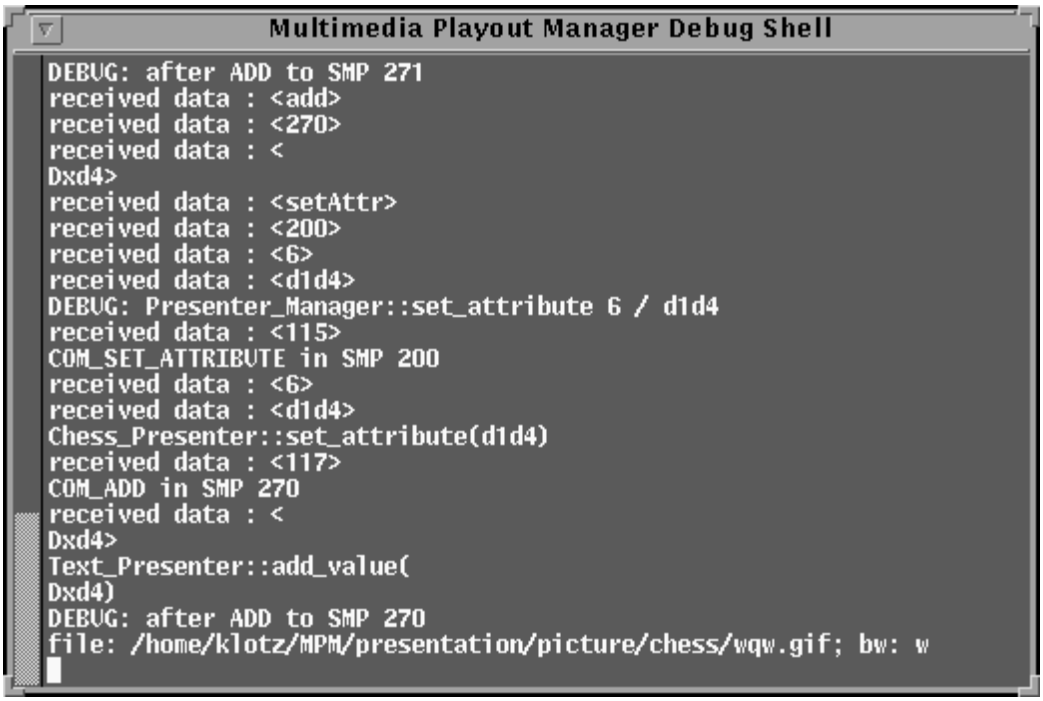

Abbildung 7.9: Debug–Fenster

7.5.4 "Send"–Fenster der Emulation des STI–Protokoll–Interpreter

Im "Send"–Fenster (siehe Abb. 7.10) wird dem Entwickler ein textuelles Menü angeboten, mit dem er die Befehle des *STI–Protokoll–Interpreter* emulieren kann. Hierbei wird unterschieden zwischen

- Kommandos an den MPM
- interne Kommandos

Kommandos an den MPM werden von der Emulation nicht interpretiert, sondern über eine Messagequeue zum MPM geschickt. Interne Kommandos hingegen werden lokal von der Emulation behandelt. Solche Befehle sind zum Beispiel:

- sleep $\langle sec \rangle \Longrightarrow$ Der Emulator pausiert $\langle sec \rangle$ Sekunden bevor der nächste Befehl abgearbeitet wird
- execute_plan $\langle file \rangle \Longrightarrow$ Sequentielles Ausführen von Emulationsbefehlen, die in der Datei $\langle file \rangle$ gespeichert sind.

7.5.5 "Receive"–Fenster der Emulation des STI–Protokoll–Interpreter

Das Receive–Fenster zeigt empfangene Daten an, ohne sie zu interpretieren. Im Moment sind dies Informationen über komplexe Interaktionen, die vom *Interaktionshandler* an den *STI–Protokoll–Interpreter* weitergegeben werden.

7.6 C++ Klassen

Abbildung 7.12 zeigt die Vererbungshierarchie der wichtigsten C++ Klassen, die im Rahmen der Implementierung des MPM entwickelt wurden.

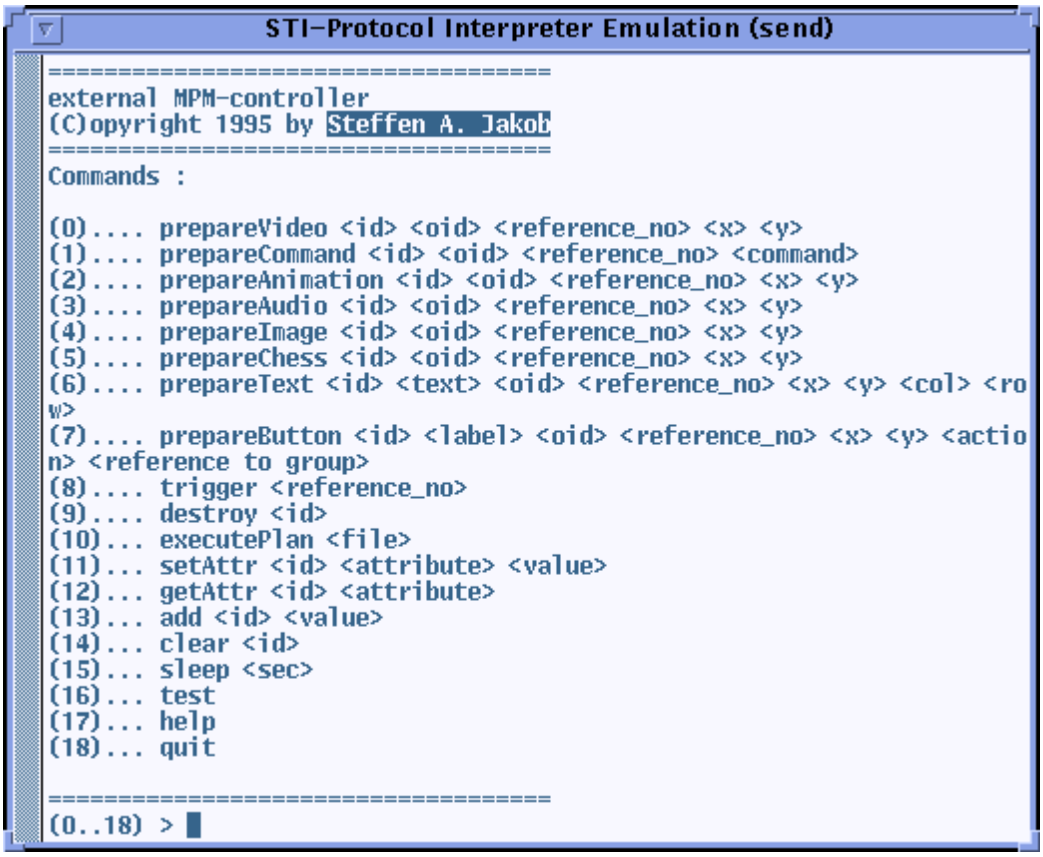

Abbildung 7.10: “Send” Fenster der STI–Protokoll–Interpreter

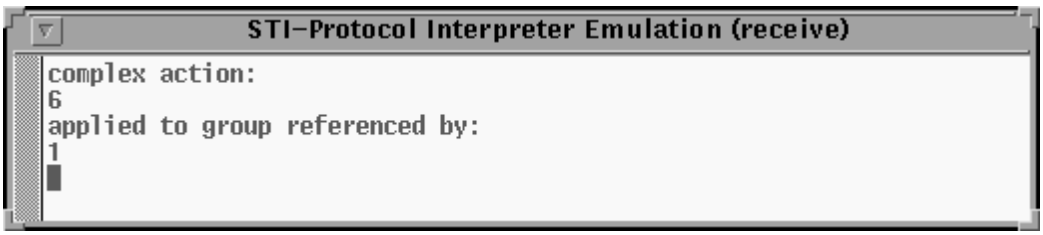

Abbildung 7.11: “Receive” Fenster der STI–Protokoll–Interpreter

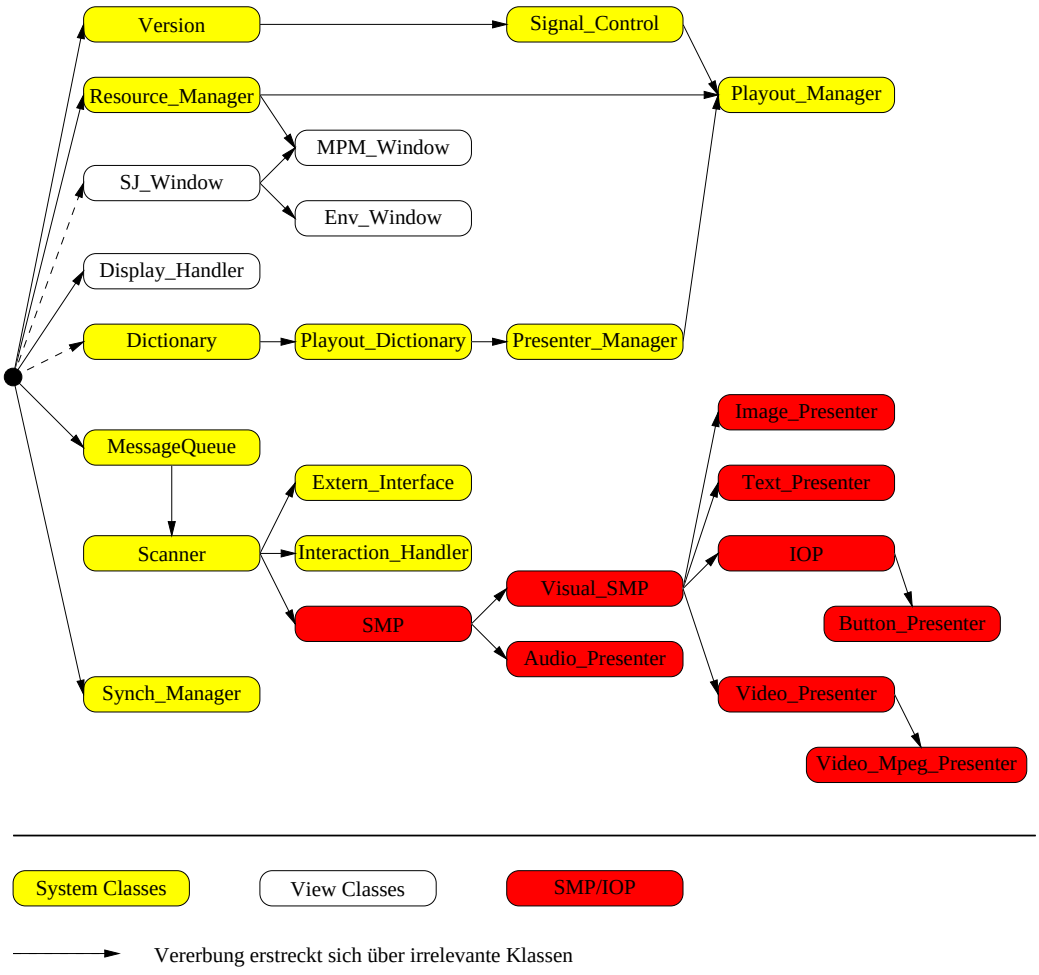

Abbildung 7.12: C++ Klassen

Teil III

Kapitel 8

Ausblick

Der MPM als zentrales Modul der Client–Umgebung des AMOS–Systems wurde als erstes Modul realisiert. Im Hinblick auf den MPM gibt es eine Reihe Aspekte, die Gegenstand weiterer Forschung sein können. Neben weiteren konzeptionellen Überlegungen wird es auch notwendig sein, die bereits existierende Funktionalität zu erweitern und dafür ein allgemeines Konzept zur Verfügung zu stellen. Weiter werden Anwendungsmöglichkeiten für eine "stand–alone" Version des MPM gezeigt. Zum Schluß wird in diesem Kapitel auf mögliche Anwendungsszenarien für das AMOS–System hingewiesen (siehe Abschnitt 8).

Reactive Playout Management

Multimediale Präsentation stellen hohe Anforderungen an das Präsentationssystem. Es ist nicht vermeidbar, daß zu nicht vorhersagbaren Zeitpunkten die Systemperformanz für eine gewünschte Präsentation nicht ausreichend ist. Dies führt zu Störungen, deren Auswirkung unterschiedlich stark sein kann. So kann sie vom Betrachter kaum bemerkt werden oder aber auch das gesamte System in einen undefinierten Zustand bringen.

Ein Lösungsansatz für dieses Problem wird in [TK95b] beschrieben. Hierbei wird jedem Betrachter ein privates Profil zugeordnet, in dem u.a. definiert ist, welche Defizite er während der Präsentation toleriert. Das sog. **Reactive Playout Management**, nutzt diese Information, um auftretende Konflikte

zu behandeln. Dabei werden drei Phasen unterschieden:

detection phase: In dieser Phase werden die explizit definierten und präsentationsimmanenten Restriktionen überwacht. Falls die Notwendigkeit besteht, zu adaptieren, wird zur nächsten Phase übergegangen.

reaction preparation phase: Hierbei wird die optimale Adaptionsstrategie ermittelt.

schedule generation and interpretation phase: Anwendung der in der reaction preparation phase ausgewählten Strategie auf die Präsentation.

Mit diesem Verfahren soll es möglich sein, externe Restriktionen, die mit der multimedialen Präsentation gespeichert wurden, zu erfüllen.

Synchronisationsmanagement

Derzeit werden die einzelnen Medien untereinander lediglich grob synchronisiert, was bedeutet, daß sie gleichzeitig gestartet und beendet werden können. Ein allgemeines Konzept für die Feinsynchronisation wird derzeit in einer auf den MPM aufbauenden Diplomarbeit entwickelt und realisiert.

Erweiterung der Präsentationsfunktionalität durch Integration weiterer SMPs/IOPs

Neben SMPs und IOPs für die bekannten Standardmedien (Bilder, Video, Audio, ...) ist es denkbar, daß applikationsspezifische SMPs und IOPs, die für bestimmte Applikationsdomänen sinnvoll sind, realisiert werden. Neben dem im Rahmen dieser Diplomarbeit entwickelten Schach–SMP sind unzählige Erweiterungen denkbar, von denen einige hier aufzählen:

- HTML[1]–SMP

[1]Hypertext Markup Language

- Statistik–SMP (Darstellung von Tortendiagrammen,...)

- SMP zur Anzeige von CAD[2]–Zeichnungen

Dynamische Erweiterbarkeit des MPM

Im Moment sind die bereits existierenden SMPs fest in das System integriert. Es wäre jedoch wünschenswert, einen Mechanismus bereitzustellen, mit dem SMPs dynamisch zur Laufzeit in das System eingebunden werden können. Der MPM kann dann während der Präsentation die angeforderten SMPs verfügbar machen, indem die entsprechende Funktionen aus einer dynamisch linkbaren SMP–Bibliothek geladen wird. Neben einer vereinfachten Erweiterbarkeit trägt dies auch zur Minimierung der Komplexität des Gesamtsystems bei.

Für ein solches Konzept ist eine Spezifikation der Fähigkeiten der SMPs notwendig, sodaß bei dynamischer Integration neuer SMPs Abhängigkeiten zu den bereits existierenden adäquat berücksichtigt werden können.

Stand–alone Version des MPM

Eine vom AMOS–System abgekoppelte Stand–alone Version des MPM (z.B. unter Verwendung des UNIX–Filesystems als einfaches Speichersystem) kann nutzvoll eingesetzt werden, wie zum Beispiel als

- generischer Viewer für beliebige Medien

- generische Oberfläche für Applikationen

[2] Computer Aided Design

Integration des AMOS–Systems in bereits bestehende Dienste

Als ein denkbares Anwendungsszenario für das AMOS–System bietet sich die Integration in das WWW an.

Wie bereits in der Einleitung erwähnt, weist die gegenwärtige Implementierung des WWW Defizite auf. Vor allem die Verwaltung der multimedialen Daten sowie deren homogene Darstellung sind eher pragmatisch nicht aber effizient realisiert.

Durch die effizientere Realisierung des AMOS–Systems sowie dessen mächtigere Modellierungsmöglichkeiten, kann das AMOS–System vorteilhaft in das WWW integriert werden.

Anhang A

Abkürzungen

AMOS Active Media Object Stores

CAD Computer Aided Design

CODU Continuous Object Data Unit

COM Continuous Object Manager

COPU Continuous Object Presentation Unit

DBVS Datenbank–Verwaltungssystem

GIF Graphics Interchange Format

GMD GMD – Forschungszentrum Informationstechnik GmbH

HTML Hypertext Markup Language

HyTime Hypermedia/Time–Based Structuring Language

IFF Image File Format

ILBM Interleaved Bitmap

IOP Interaction Object Presenter

IPC Interprozeß–Kommunikation

IPSI Integrated Publication and Information Systems Institute

JPEG Joint Photographic Experts Group

LWP Lightweight–Prozeß

MHEG Multimedia and Hypermedia Information Coding Experts Group

MIDI Musical Instrument Digital Interface

MMDBVS Multimedia–Datenbank–Verwaltungssystem

MMPR Multimedia Presenter

MPEG Moving Pictures Experts Group

MPM Multimedia Playout Manager

OSF Open Software Foundation

PBM Portable Bitmap File Format

PD Public Domain

RAPM Reactive Playout Management

QoS Quality of Service

SMP Single Media Presenter

STI Spatial Temporal Interaction

TIFF Tagged Image File Format

VML VODAK Manipulation Language

VODAK Verteiltes Objektorientiertes Datenbanksystem

VQL VODAK Query Language

VRAPI VODAK Remote Application Programming Interface

WORM Write Once Read Multiple

WWW World Wide Web

XMS External Media Server

Literaturverzeichnis

[BLCG+92] T. J. Berners-Lee, R. Cailliau, J.-F. Groff, B. Pollermann, and CERN, Electronic Networking: Research, Applications and Policy, vol. 2, ch. World-Wide Web: The Information Universe, pp. 52–58, Meckler Publishing, Westport, CT, USA, 1992, pp. 52–58.

[Fan93] X. Fan, *Latency–Directed Multithreaded Computation and Its Architectural Support*, Ph.D. thesis, Fachbereich Informatik der Universität Hamburg, 1993.

[GMD94] GMD/IPSI, *Open object–oriented database systems (vodak)*, 1994, URL: http://este.darmstadt.gmd.de:5000/vodak/.

[GSS+94] T. Görlich, B. Schuster, K. Spatzek, J. Spieker, and A. Stärke, *Entwicklung und Integration der VML–Klasse Video in das DBMS VODAK*, Praktikum Fachbereich Informatik, Technische Hochschule Darmstadt, 1994.

[Har94] R. Hartmann, *Audio*, Master's thesis, Fachbereich Informatik der Technischen Hochschule Darmstadt, 1994.

[Hel91] Dan Heller, *Motif programming manual*, O'Reilly, 1991.

[Jak93] S. Jakob, *Entwicklung des Datentyps VIDEO für das objektorientierte Datenbanksystem VODAK*, Studienarbeit Fachbereich Informatik, Technische Hochschule Darmstadt, 1993.

[Koh95] D. Kohlmeyer, *Michail tal memorial*, Schach Magazin 64 / Schach Echo (1995), 233.

[Kra94] A. Kraiß, *Ein Objektmanager zur Verwaltung kontinuierlicher Objekte für das OODBMS VODAK*, Master's thesis, Fachbereich Informatik der Technischen Hochschule Darmstadt, 1994, GMD–Studie.

[Lip89] Stanley B. Lippman, *C++ primer*, Addison Wesley, 1989.

[LS87] P. C. Lockemann and J. W. Schmidt, *Datenbank–Handbuch*, Springer Verlag, 1987.

[MW91] K. Meyer-Wegener, *Multimedia datenbanken*, Leitfäden der angewandten Informatik, Teubner Stuttgart, 1991.

[OQL88] Tim O'Reilly, Valerie Quercia, and Linda Lamb, *X window system user's guide*, O'Reilly, 1988.

[RL94] T. Rakow and M. Löhr, *Das Ende der Sprachlosigkeit - Auf dem Weg zum multimedialen Datenbanksystem*, GMD-Jahresbericht 1993/94, GMD, Sankt Augustin, 1994.

[RM93] T. Rakow and P. Muth, *The V3 Video Server - Managing Analog and Digital Video Clips*, Proc. SIGMOD '93, May 1993, pp. 556–557.

[Ste93] R. Steinmetz, *Multimedia-technologie: Einführung und grundlagen*, Springer Berlin, 1993.

[Sun93a] SunSoft, *Solaris user level threads — an exercise in application design*, 1993.

[Sun93b] SunSoft, *Sunos5.3 guide to multithreading programming*, 1993.

[TK95a] H. Thimm and W. Klas, *Playout Management — An Integrated Service of a Multimedia Database Management System*, FirstInternational Workshop on Multi–Media Database Management Systems, Blue Mountain Lake, NY, USA, IEEE Computer Society Press, August 1995, pp. 28–30.

[TK95b] H. Thimm and W. Klas, *Reactive Playout Management Adapting Multimedia Presentations to Contradictory Constraints*, Arbeitspapier, GMD/IPSI, May 1995.

[WM94] R. Winacott and R. Marejka, *Thread Extension Library*, Sun Microsystems of Canda, 7 1994, URL: ftp://opcom.sun.ca/pub/threads.