Untersuchung der Realisierungsmöglichkeiten von CYBOL - Webfrontends, unter Verwendung von Konzepten des Cybernetics Oriented Programming (CYBOP)

Diplomarbeit zur Erlangung des akademischen Grades Diplominformatiker, vorgelegt der Fakultät für Informatik und Automatisierung der Technischen Universität Ilmenau

von: Rolf Holzmüller

Betreuer: Dipl.-Ing. Christian Heller

verantwortlicher Hochschullehrer: Prof. Dr.-Ing. habil. Ilka Philippow

Inventarisierungsnummer: 2005-04-01/032/JN93/2232

eingereicht am: 23. Juni 2005

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Kapitel 1

Einleitung

1.1 Ziel

In der vorliegenden Diplomarbeit wird CYBOP (Cybernetics Oriented Programming) und dessen Realisierungsmöglichkeiten einer Webanwendung untersucht. Dies basiert auf den Konzepten von CYBOP. CYBOP setzt sich aus der Beschreibungssprache CYBOL und dem Interpreter CYBOI zusammen. Der Hauptgedanke in CYBOL ist, dass die Programmiersprache nicht dem technischem Umfeld (Computer) bzw. programmtechnischen Umfeld (Programm) genügt, sondern sich an dem menschlichen Denken orientiert. Dazu ist es immer noch nötig die Mensch-Maschine-Schnittstelle zu definieren, nur dass diese Schnittstelle dem Menschen auf fundamentaler Ebene verständlich ist. Auf der Basis dieser Sprache wird ein kleines Anwendungsprogramm entwickelt, welches im Webbrowser ablaufen soll. Für die Realisierung dieser Aufgabe nähert man sich dem Problem auf zwei Wegen, der erste Weg aus der Sicht von CYBOP und der zweite aus der Sicht von Webanwendungen. Aus der Sicht von CYBOP ist zu klären, auf welche Prinzipien CYBOP basiert und wie sich die Prinzipien in CYBOL auswirken. Danach wird die aktuelle technische Umsetzung des Interpreters CYBOI, sowie dessen Designaufbau und Entwicklungsprinzipien erklärt. Für den zweiten Weg sind die Darstellungsmöglichkeiten in Webanwendungen zu untersuchen, eine Auswahl aus diesen Möglichkeiten zu treffen und die erforderlichen technischen Voraussetzungen zu klären. Am Schluss sind beide Wege zu integrieren. Dafür ist zu prüfen, welche Erweiterungen CYBOL braucht, um die Definition für die Webanwendung vollständig abzudecken, wie die Umsetzung in CYBOI eingebunden wird und welche Probleme dabei zu beachten und zu lösen sind.

1.2 Umfeld

Die Diplomarbeit wird im Rahmen des Projektes CYBOP realisiert. Dieses Open Source Projekt verfolgt das Ziel, Softwareentwicklung auf Basis von natürlichen Konzepten zu entwickeln und dies als Debian-Paket zur Verfügung zu stellen. Dazu ist es notwendig, dass die erstellten Programme und Dokumentationen unter die GNU-Lizenz gestellt sind. Weitere Informationen für das Debian-Projekt sind unter [deb05] und für das GNU-Projekt unter [gnu05] zu finden.

1.3 Praktische Aufgabe

Für die praktische Aufgabe der Diplomarbeit ist eine Anwendung auf Basis von CYBOL zu entwickeln, die die folgenden Aufgaben abdeckt:

- Darstellung von Adressdaten in einer Tabelle
- die dargestellten Adressdaten sollen löschbar und editierbar sein
- es sollen neue Adressen eingebbar sein

Als Voraussetzung für diese Webanwendung sind zu erstellen:

- Domain-Modell in CYBOL
- Anwendungsmodell in CYBOL
- Weboberflächenbeschreibung in CYBOL
- Prototyp eines Webservers mit Integration in CYBOI
- Erweiterung von CYBOI für die Bearbeitung von Web-spezifischen Anfragen

Kapitel 2

Grundlagen von CYBOP

2.1 Überblick

CYBOP steht für *Cybernetics Oriented Programming*. Laut [MD90] ist Kybernetik folgendermaßen definiert:

> Forschungsrichtung, die vergleichende Betrachtungen über Gesetzmäßigkeiten im Ablauf von Steuerungs- und Regelungsvorgängen in Technik, Biologie und Soziologie anstellt.

Kybernetik ist also eine Richtung die versucht das Verhalten und die Abstraktion von einzelnen Systemen zu betrachten und zu vergleichen. Die Programmierung von Anwendungssystemen wird von Menschen realisiert. Darum ist es nahe liegend, die Programmierung (Beschreibungssprache) dem menschlichen Denken entsprechend zu modellieren.

Laut [Hel04] gibt es folgende Prinzipien für das menschliche Denken:

> Fundamental principles of human thinking are Discrimination, Categorization and Composition. The abstractions they deliver are Item, Category and Compound.

CYBOP versucht nicht menschliches Denken nachzubilden, wie es der Begriff suggerieren könnte oder wie der Begriff auch in dem Bereich Künstliche Intelligenz verwendet wird, sondern die Beschreibungssprache soll dem menschlichen Denken entsprechen.

2.2 Discrimation/Items

Der Mensch versucht seine Umwelt zu verstehen. Zur Unterscheidung seiner Umwelt gibt es in der Psychologie den Begriff *Discrimination*. Dabei zergliedert der Mensch seine Umwelt in kleine Teile. Die Abbildungen der realen Umwelt auf diese kleinen Teile werden in CYBOP *Items* genannt. Die *Items* sind kontextabhängig. Für verschiedene Aufgaben werden auch unterschiedliche *Items* gebildet, je nachdem welche Parameter für die Aufgaben relevant sind.

Der Mensch kann nur mit solchen Abbildungen umgehen, da er nie die Umwelt in ihrer Komplexität verstehen kann und dies für die Lösung von Aufgaben auch nicht braucht. Das menschliche Denken kann nur mit Modellen der Umwelt umgehen, aber nie mit der gesamten realen Umwelt.

2.3 Categorization/Category

Unter *Categorization* versteht wir in diesem Zusammenhang die Fähigkeit des Menschen verschiedene Teile auf Grund bestimmter Merkmale zu Gruppen zusammenzufassen. In CYBOP wird dies *Category* genannt. In der *Category* wird eine „is-a-Beziehung“ beschrieben. Dies bedeutet, ein Teil gehört auf Grund spezieller Eigenschaften zu einer bestimmten Gruppe. Die Gruppierungen können unterschiedlicher Natur sein. Ein einfaches Beispiel wäre die Kategorie Mensch. Zu dieser Kategorie würden z.B. Afrikaner, Europäer usw. zählen.

2.4 Composition/Compound

Teile können zu größeren Teilen zusammengefasst werden. Dieses Prinzip nennt man *Composition* und verkörpert eine „has-a-Beziehung“. Die kleinsten nicht teilbaren Teile nennt man Items. Wir wissen aber, das Teilen in der Regel weiter zerlegt werden können. Darum verstehen wir hier unter kleinsten nicht teilbaren Teilen eine für die Aufgabe ausreichende

Zerlegung. Die Abstraktion in CYBOP von einer *Composition* ist das *Compound*. Letzten Endes kann man also sagen, die Composition (lat. compositio = Zusammengesetztes) ist ein Zusammenfügen von *Items* oder anderen *Compounds* zu etwas Größerem.

2.5 Zusammenhang der Prinzipien

In der Abbildung 2.1 wird der Zusammenhang zwischen den gerade beschriebenen Prinzipien von *Human Thinking* verdeutlicht. Dies ist von [Hel04] entnommen.

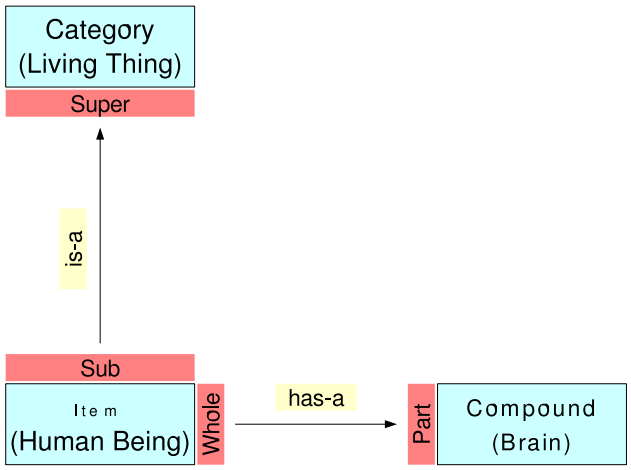

Abbildung 2.1: Human Thinking

Items können über die „has-a-Beziehung“ ein *Compound* bilden und werden über die „is-a-Beziehung“ zu einer *Category* zusammen gefasst. Wie sind diese Prinzipien in CYBOL umgesetzt? Jedes Template bzw. Laufzeitmodell vom Template repräsentiert ein diskretes *Item* bzw. ein diskretes *Compound*, was wiederum aus disketen *Items* oder *Compounds* besteht. Die Abstraktion *Category* ist in CYBOL nicht direkt als Beschreibung abgebildet, sondern muss über Operationen in CYBOL realisiert werden.

2.6 Architektur von CYBOP

CYBOP setzt sich aus verschieden Teilen zusammen. Dies beinhaltet einmal die Beschreibungssprache, dann einen Interpreter, der diese Beschreibungssprache versteht und auf unterster Ebene die Hardware, auf der die Operationen ausgeführt werden.

- Beschreibungssprache CYBOL

  Hier werden die Modelle beschrieben. Dazu gehören das Domain-Wissen, die Anwendungslogik und die Beschreibung der Oberfläche für verschieden Ausgabemedien.

- Interpreter CYBOI

  Dies ist das Herzstück von CYBOP. Hier wird die Sprache CYBOL interpretiert. Dazu werden die Modelle gelesen, verarbeitet und für die verschiedenen Ausgabemedien die Anwendung generiert.

- Hardware

  Damit eine Anwendung funktioniert muss sie mit der Hardware zusammen arbeiten. Dies muss durch den CYBOP - Interpreter gewährleistet werden.

Die Beschreibungssprache CYBOL und der Interpreter CYBOI werden in den nächsten Kapitel beschrieben. Der Abschnitt Hardware ist für die Diplomarbeit nicht relevant.

Kapitel 3

Cybernetics Oriented Language (CYBOL)

3.1 Überblick

CYBOL ist die Beschreibungssprache von CYBOP für das Anwendungswissen, die Anwendungslogik und die Oberflächenbeschreibung. CYBOL ist eine hierarchische Beschreibungssprache. Darum eignet sich für CYBOL das XML-Format, da XML genau für hierarchische Abbildungen gedacht ist. Dies hat den Vorteil, das vorhandene XML-Parser verwendet werden können. Weiterhin stehen XML-Editoren zur Erstellung und Überprüfung der CYBOL-Dateien zur Verfügung. In diesem Kapitel wird zuerst XML näher erläutert, danach wird CYBOL in der XML-Syntax beschrieben und als letztes erfolgen die formellen Beschreibungen von CYBOL in DTD, XML-Schema und EBNF.

3.2 XML - Grundlage der Beschreibung von CYBOL

3.2.1 Geschichtliches

Die Grundlage von CYBOL ist XML. XML ist ein Projekt der W3C und wird seit Juli 1996 entwickelt. Ursprünglich war XML als Alternative zur Hypertext Markup Language (HTML), der Auszeichnungssprache für Internetseiten, gedacht. Es wurde jedoch schnell erkannt, das XML als eine universelle Sprache zur Verwaltung und zum Austausch semantisch qualifizierter Daten genutzt werden kann. Im November 1996 wurde XML als Entwurf vorgestellt und hat seit Februar 1998 in der Version XML 1.0 den Status „Empfehlung des W3C“.

XML stellt eine Teilmenge der Standard Generalized Markup Language (SGML) dar.

3.2.2 Definition von XML

Hier folgt nun eine kurze Erläuterung, was XML eigentlich ist [Ull02, Seite 618].

> Der Inhalt eines XML-Dokuments besteht aus strukturierten Elementen, die hierarchisch geschachtelt sind. Dazwischen befindet sich der Inhalt, der aus weiteren Elementen (daher hierarchisch) und reinen Text bestehen kann. Die Elemente können Attribute enthalten, die zusätzliche Informationen in einem Element ablegen.

Ein weiterer Begriff in Zusammenhang mit XML ist die Wohlgeformtheit eines Dokumentes. Jedes XML-Dokument muss der Wohlgeformtheit genügen, das bedeutet dass jedes XML-Dokument einer speziellen Spezifikationen entspricht. Diese Spezifikation ist im Detail hier aufgelistet:

- Jedes Element besteht aus einem Begin-Tag und einem End-Tag.
- Es gibt genau ein Wurzelelement, unter dem sich ein hierarchischer Baum aufbaut.
- Elemente müssen sauber ineinander eingebettet sein. Es darf keine Überdeckungen geben.
- Alle Attributwerte müssen in Anführungsstrichen stehen.
- Der Name eines Attributs kommt innerhalb eines Elements nicht mehr als einmal vor.
- Ein Attributwert darf keinen Verweis auf ein externes Entity enthalten.

3.2.3 Syntax von XML

Die folgenden Ausführungen beziehen sich auf die XML-Spezifikation des W3C [w3c05a]. Dabei werden nur die Teile beschrieben, die auch in CYBOL verwendet werden.

Struktur von XML-Dokumenten

Ein XML-Dokument besteht aus einem optionalen Prolog, einem Rumpf sowie aus einem optionalen Epilog. Ein Dokument kann wohlgeformt sein, ohne einen Prolog oder Epilog zu haben.

Der Prolog kann die XML-Deklaration, die Informationen über die verwendete XML-Spezifikation, die Zeichenkodierung, sowie die verwendeten Entities enthalten. Hier ist ein Beispiel für einen Prolog, so wie er auch in CYBOP verwendet wird, zu sehen:

```
<?xml version="1.0" encoding="iso-8859-1" standalone="yes"?>
```

Der Rumpf ist der wichtigste Teil des XML-Dokumentes, da hier die eigentlichen Informationen abgelegt werden. Er besteht aus einem Element oder mehreren ineinander verschachtelten, Elementen.

Der Epilog eines Dokumentes ist der Abschnitt nach dem Rumpf und steht für Kommentare und Verarbeitungsanweisungen zur Verfügung. Die Verwendung dieses Abschnittes ist bisher nicht klar definiert, deshalb ist eine sinnvolle Nutzung nur mit eigenen Applikationen möglich.

Elemente

Elemente sind die Grundbausteine eines XML-Dokumentes. Ein Element ist ein Container für beliebige Inhalte, wie zum Beispiel verschiedene Zeichen, weitere Elemente sowie andere Informationen. Begin-Tag und End-Tag begrenzen ein Element. Sie bestehen aus dem Namen des Elementyps, der in spitzen Klammern eingeschlossen wird.

```
<Elementname> Elementinhalt </Elementname>
```

Elemente ohne Inhalt werden als leere Elemente bezeichnet und können durch eine spezielle Schreibweise angegeben werden.

```
<Leerelement />
```

Soll ein Dokument wohlgeformt sein, muss es eine baumartige Struktur aufweisen. In jedem Dokument darf es nur genau einen Baum geben, dessen Wurzel als *document root* bezeichnet wird. Dieses Element ist das Elternelement aller anderen enthaltenen Elemente

(Kinderelemente). Jede Wurzel eines Kinderelements muss mindestens in einen Teilbaum enthalten sein, dessen Wurzel das *document root* ist. Die Verschachtelung der Elemente ist strikt einzuhalten, das heißt ein Element muss seine Begin-Tags und End-Tags auf derselben Verschachtelungsebene haben.

Attribute

Attribute sind Teile von Elementen, die Informationen über deren Inhalt enthalten und nicht selbst zum Inhalt gehören. Attribute können in Begin-Tags von Elementen und Leerelementen angegeben werden und haben immer die Form:

```
Attributname="Attributwert"
```

Die Attributwerte müssen immer Text-Literale sein. Text-Literale sind ganz normale Zeichenketten, die von Begrenzungszeichen (`" oder '`) eingeschlossen werden. Dabei ist darauf zu achten, dass am Anfang und Ende dasselbe Begrenzungszeichen steht und das dieses nicht in der Zeichenkette verwendet wird.

Kommentare

Sämtliche Anmerkungen in Dokumenten, die nicht zum Inhalt gehören, werden als Kommentare gekennzeichnet. Die Syntax von XML für Kommentare ist

```
<!-- Hier steht ein Kommentar -->
```

Diese ermöglichen es, Anmerkungen aufzunehmen, Entwicklungsschritte zu dokumentieren oder verschiedene andere Informationen einzufügen, die nur für den Leser des Quelltextes bestimmt sind.

Weiter Informationen zu XML sind direkt auf den Seiten des W3C [w3c05a] bzw. in der deutschen Übersetzung [w3c05b] zu finden.

3.3 Beschreibung von CYBOL

Nachdem XML und ihre Syntax geklärt ist, kann die spezielle Syntax von CYBOL dargelegt werden. Ein wichtiger Aspekt von CYBOL ist die Einfachheit der Beschreibung. Darum werden alle Modelle nach dem gleichen Schema beschrieben, egal, ob es sich um die Anwendungslogik, die Oberflächenbeschreibung oder die Domainbeschreibung handelt. Diese CYBOL-Dateien werden auch Templates genannt, da aus diesen die Laufzeitmodelle in CYBOI aufgebaut werden.

3.3.1 Aufbau von CYBOL

Eine CYBOL-Datei ist immer ein XML-Dokument. Es besteht aus dem Prolog und dem Rumpf. Der Epilog wird in CYBOL nicht verwendet. Der Prolog setzt sich aus der XML-Deklaration und einer Beschreibung zusammen.

```
<?xml version="1.0" encoding="iso-8859-1" standalone="yes"?>
<!--
  Copyright and Description for the CYBOL file
/-->
```

Der Rumpf von CYBOL ist hierarchisch mit Elementen und den dazugehörigen Attributen aufgebaut. Ein Rumpf sieht beispielhaft folgendermaßen aus:

```
<model>
  <part name="addresses" ...">
    <property name="whole" ..." >
      <constraint name="color" ... />
      ...
    </property>
    ...
  </part>
```

```
  ...
</model>
```

3.3.2 Elemente von CYBOL

In dem Rumpf können bis zu vier Elementtypen verwendet werden. Diese Elementtypen sind hierarchisch aufgebaut.

```
model --> part --> property --> constraint
```

Die Elementtypen haben folgende Bedeutung:

Elementtyp	**Beschreibung**
model	Dieses Element ist das Wurzelelement. Alle anderen Elemente sind ihm untergeordnet.
part	Jedes Modell besteht aus einem oder mehreren *parts*. Dies kann zum Beispiel ein Datenelement sein oder auch eine Operation. Voneinander abhängige *parts* werden hierarchisch aufgebaut (siehe dazu 3.3.4).
property	Beschreibt die Eigenschaften zu den *parts*. Ein Beispiel wäre zu einer Sendeoperation die Eigenschaften des Senders und des Empfängers zu definieren.
constraint	Unterliegen die *properties* bestimmten Bedingungen, die z.B. einen eingeschränkten Wertebereich darstellen könnten, so werden diese hier definiert.

Tabelle 3.1: Elemente von CYBOL

Andere Elementtypen sind in CYBOL nicht erlaubt. Die Elementtypen *property* und *constraint* müssen nicht immer angegeben werden, sondern nur, wenn es im Kontext zu den Elementtyp *part* sinnvoll ist.

3.3.3 Attribute der Elemente von CYBOL

Für die vollständige Beschreibung wird nun auf die Attribute der Elemente eingegangen. Auch hier gibt es immer die gleichen Attribute, egal um welche Beschreibung es sich handelt. Die Attribute in den Elementen haben folgende Bedeutung:

Attribut	**Beschreibung**
name	Das Attribut *name* ist die Bezeichnung für den Eintrag. In Abhängigkeit von dem zu beschreibenden Element ist dieser frei wählbar (bei Element part) oder durch den Kontext bestimmt. Ein Beispiel für einen Name mit einer bestimmten Bedeutung wäre für das Element *property* der Name *color* für die Farbe des *part*.
channel	Dieses Attribut beschreibt, wie der Eintrag im Attribut *model* zu lesen ist. Dies könnte *inline* sein, wenn die Beschreibung im Attribut *model* enthalten ist oder es könnte *file* annehmen, wenn im Attribut *model* auf eine weitere CYBOL-Datei verwiesen wird.
abstraction	Dies entspricht im herkömmlichen Sinne dem Datentyp. Bei einfachen Datentypen wäre dies *string*, *integer*, *float* oder *vector*. Bei zusammengesetzten Datentypen würde dies *cybol* entsprechen. Für eine primitive Operation der Anwendungslogik würde der Wert *operation* heißen.
model	Dies ist abhängig von der *abstraction*. Bei einfachen Datentypen kann im Attribut *model* ein Wert zugewiesen werden. Bei Operationen wird hier die auszuführende Operation definiert. Ist die *abstraction* gleich *cybol*, so wird im *model* auf eine weitere CYBOL-Datei verwiesen.

Tabelle 3.2: Attribute der Elemente von CYBOL

Dies ist die allgemeine Beschreibung von CYBOL. Was jetzt noch fehlt, ist die Definition, welche Werte die einzelnen Attribute der Elemente annehmen können und in welcher Kombination sie sinnvoll sind. CYBOL ist noch in der Anfangsphase der Entwicklung und wird systematisch weiter ausgestaltet. Bei der Beispielanwendung, die im Rahmen der Diplomarbeit entsteht, werden diese Werte für die einzelnen Attribute der Elemente beschrieben.

3.3.4 Abbildung von Hierarchien

Nicht nur der Aufbau der Beschreibungssprache ist hierarchisch, sondern auch die Abbildungen der *part*. Ein Teil setzt sich aus mehreren Teilen zusammen. Dies wird in CYBOL über mehrere Dateien gelöst. Innerhalb des Elements part kann keine Hierarchie abgebildet werden, sondern dies geschieht über die Einbindung weiterer CYBOL-Dateien mit den Attributen `channel="file"`, `abstraction="cybol"` und `model="?model?"`, wobei `?model?` für das entsprechende hierarchisch untergeordnete CYBOL-Modell steht. Ein kleines Beispiel könnte so aussehen:

```
<model>
  <part name="table" channel="file"
        abstraction="cybol" model="part_of_table.cybol" />
</model>
```

Hier wird der *part* Tisch definiert. In der Datei part_of_table.cybol sind die parts Tischplatte und die 4 Tischbeine definiert.

```
<model>
  <part name="tabletop" channel="inline"
        abstraction="vector" model="50,50,5" />
  <part name="tableleg1" channel="inline"
        abstraction="vector" model="5,5,30" />
  ...
</model>
```

3.4 Formelle Beschreibungen von CYBOL

Für die formelle Beschreibung von CYBOL gibt es verschiedene Möglichkeiten. Da CYBOL in XML beschrieben wird, sind die Meta-Beschreibungen von XML als formelle Beschreibung möglich. Dies beinhaltet die Document Type Definition, kurz DTD, und die XML-Schemata. Eine weitere Möglichkeit für die formelle Beschreibung besteht in der Backus-Naur-Form (BNF) bzw. in der erweiterten Backus-Naur-Form (EBNF).

3.4.1 Document Type Definition (DTD)

Eine Document Type Definition, kurz DTD, beschreibt den strukturellen Aufbau und die logischen Elemente einer Klasse von Dokumenten, genannt Dokumententyp. Die DTD ist eine Mustervorlage für eine Sammlung gleichartiger Dokumente. Ein Editor kann dieses Muster schon beim Schreiben mit dem aktuellen Dokument vergleichen und auf strukturelle Fehler, z.B. fehlende Elemente, hinweisen. Ohne DTD kann kein Programm entscheiden, welche Elemente zu einem Dokument gehören und ob sie zwingend notwendig sind.

Die Angabe der DTD erfolgt in einem XML-Dokument mit der DOCTYPE-Deklaration. Dabei wird zwischen interner und externer Deklaration unterschieden. Bei einer internen Deklaration wird die DTD innerhalb der DOCTYPE-Deklaration angegeben, bei der externen Deklaration wird auf eine separate DTD verwiesen. CYBOL verwendet die DOCTYPE-Deklaration für die DTD nicht. Darum brauchen sie in CYBOL-Dateien weder intern noch extern referenziert werden.

Elemente

Elemente sind das Kernstück von XML-Dokumenten. Sie werden in der DTD durch das Tag `<!ELEMENT name (wert) >` definiert. Dabei steht 'name' für den Namen des Elements und für 'wert' können die Werte 'EMPTY', 'ANY', 'PCDATA' (als Datentyp) oder weitere untergeordnete Elemente eingetragen sein. Bei untergeordneten Elementen kann noch unterschieden werden, wie oft diese Elemente vorkommen dürfen (? einmal oder gar nicht, * keinmal, einmal oder beliebig oft, + mindestens einmal). Weiterhin besteht die Möglichkeit,

Sequenzen von Elementen (',' zwischen den Elementen) oder Alternativen von Elementen ('—' zwischen den Elementen) zu definieren.

Attribute

Attribute bieten die Möglichkeit, Elemente zu erweitern und zu modifizieren. Mit ihrer Hilfe können zum Element gehörende Informationen angegeben werden. Diese Attribute werden auch innerhalb der DTD definiert. Alle Elemente, denen Attribute zugeordnet sind, erhalten in der DTD eine Attributliste mit dem Tag `<!ATTLIST name typ vorgabewert >`. Jede Attributdefinition besteht aus einem Namen, dem Typ und einem optionalen Vorgabewert. Typ ist im einfachsten Fall CDATA, für Vorgabewerte können z.B. #REQUIRED (erforderlich) oder #IMPLIED (optional) eingetragen werden.

DTD für CYBOL

Hier folgt nur die komplette DTD für CYBOL. Da sie nicht in den CYBOL-Dateien referenziert wird, ist dies nur eine formale Beschreibung von CYBOL.

```
<!ELEMENT model (part*)>
<!ELEMENT part (property*)>
<!ELEMENT property (constraint*)>
<!ELEMENT constraint EMPTY>

<!ATTLIST part
    name CDATA #REQUIRED
    channel CDATA #REQUIRED
    abstraction CDATA #REQUIRED
    model CDATA #REQUIRED>

<!ATTLIST property
    name CDATA #REQUIRED
    channel CDATA #REQUIRED
    abstraction CDATA #REQUIRED
    model CDATA #REQUIRED>
```

```
<!ATTLIST constraint
    name CDATA #REQUIRED
    channel CDATA #REQUIRED
    abstraction CDATA #REQUIRED
    model CDATA #REQUIRED>
```

3.4.2 XML - Schema

XML-Schema ist ein neuer Ansatz zur Definition von Dokumententypen. Mit der Entwicklung von XML hatte sich zunächst die von der SGML übernommene DTD als Format zur Beschreibung konkreter Dokumententypen etabliert. Mit der zunehmenden Verbreitung von XML in der Praxis machten sich zunehmend die Grenzen und Nachteile der DTD bemerkbar. Besonders die dokumentenzentrierte Sichtweise der DTD unter Vernachlässigung von Datentypen erwies sich als Problem. So ließen DTD weder die Beschreibung bestimmter semantischer Bedingungen noch die Festlegung von Wertebereichen zu. Gerade die zunehmende Verbreitung verteilter Anwendungen erfordert es, Daten in einem einheitlichen, aber flexiblen und leicht modifizierbaren Format, das sich einfach parsen lässt, zu transportieren. Für die Lösung des Problems sollte eine XML-basierte Sprache genutzt werden. Daraus entstand das XML-Schema.

Die komplette Beschreibung von XML - Schema würde den Rahmen dieser Arbeit sprengen. Darum wird nur auf die Elemente eingegangen, die in der Beschreibung für CYBOL notwendig sind. Weitere Informationen zu XML-Schemata sind unter [xsd05] zu finden.

Namensraum

Im Wurzelelement des Schemas wird der aktuelle Namensraum für Schemata nach W3C Version 1.0, bezogen auf die dort definierten Strukturen, angegeben:

```
xmlns:xs="http://www.w3.org/2001/XMLSchema"
```

Der Namensraum wird nun durch den Alias `xs` repräsentiert. Der definierte Alias wird im Dokument dann vor allen Elementen, Attributen und Datentypen mit angegeben, so das die Beschreibung innerhalb des Namensraums eindeutig ist. **Komplexe Elementtypen**
Das Element complexType dient zur Definition eines Elementtyps, welcher weitere Unterelemente beinhalten kann. Das Attribut name definiert den Bezeichner dieses Elementtyps. Die Reihenfolge und Auswahl der Unterelemente kann durch folgende Kompositoren angegeben werden:

Kompositor	**Bedeutung**
sequence	Vorgegebene Reihenfolge de Subelemente
choice	Auswahl eines der Subelemente
all	Alle Subelemente oder keines, Reihenfolge ist egal

Tabelle 3.3: Kompositoren komplexer Elemente

Elemente

Zur Deklaration eines Elementes in einem XML-Schema reichen die Angabe eines Namens und eines Typs als Attribute des Elementes ¡element¿. Durch die Attribute minOccurs und maxOccurs kann zusätzlich angegeben werden, wie oft das Element mindestens bzw. maximal hintereinander auftreten darf.

```
<xs:element name="MyElement" type="xs:string"/>
```

Attribute

Attributdeklarationen erscheinen am Schluss einer Elementbeschreibung. Ihre Angabe erfolgt durch das Element ¡attribute¿, das die Attribute name und type hat. Mit dem Attribut use kann angeben werden, ob das entsprechende Attribut definiert werden muss. Folgende Deklaration beschreibt ein Attribut:

```
<xs:attribute name="MyAttribute" type="xs:string"/>
```

XML-Schema für CYBOL

Hier folgt nun das komplette XML-Schema für CYBOL. Da dieses Schema, wie auch die DTD nicht in den CYBOL-Dateien referenziert wird, ist dies auch nur eine formelle Beschreibung von CYBOL.

```
<xs:schema
  xmlns:xs='http://www.w3.org/2001/XMLSchema'
  targetNamespace='http://www.cybop.net'
  xmlns='http://www.cybop.net'
  elementFormDefault='qualified'>
  <xs:element name='model'>
    <xs:complexType>
      <xs:sequence>
         <xs:element ref='part' minOccurs='0' maxOccurs='unbounded'/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name='part'>
    <xs:complexType>
      <xs:sequence>
        <xs:element ref='property' minOccurs='0' maxOccurs='unbounded'/>
      </xs:sequence>
      <xs:attribute name='name' type='xs:string' use='required'/>
      <xs:attribute name='channel' type='xs:string' use='required'/>
      <xs:attribute name='abstraction' type='xs:string' use='required'/>
      <xs:attribute name='model' type='xs:string' use='required'/>
    </xs:complexType>
  </xs:element>
  <xs:element name='property'>
    <xs:complexType>
      <xs:sequence>
```

```
        <xs:element ref='constraint' minOccurs='0' maxOccurs='unbounded'/>
      </xs:sequence>
      <xs:attribute name='name' type='xs:string' use='required'/>
      <xs:attribute name='channel' type='xs:string' use='required'/>
      <xs:attribute name='abstraction' type='xs:string' use='required'/>
      <xs:attribute name='model' type='xs:string' use='required'/>
    </xs:complexType>
  </xs:element>
  <xs:element name='constraint'>
    <xs:complexType>
      <xs:attribute name='name' type='xs:string' use='required'/>
      <xs:attribute name='channel' type='xs:string' use='required'/>
      <xs:attribute name='abstraction' type='xs:string' use='required'/>
      <xs:attribute name='model' type='xs:string' use='required'/>
    </xs:complexType>
  </xs:element>
</xs:schema
```

3.4.3 Erweiterte Backus-Naur-Form (EBNF)

Die Grammatik einer formalen Sprache regelt die Rechtschreibung. Sie legt die korrekte Syntax fest. Es gibt unterschiedliche Notationen für Grammatiken. Eine einfache und ausdrucksstarke Form ist die Erweiterte Backus-Naur-Form (EBNF). Seltener verwendet wird die ursprüngliche, einfache Backus-Naur-Form (BNF). Dieser fehlen gegenüber der EBNF ein paar bequeme, abkürzende Schreibweisen. Mit beiden Formen lassen sich aber dieselben Grammatiken wiedergeben.

Mit EBNF lassen sich Grammatiken vom Typ 2 der Chomsky-Hierarchie darstellen. Dies sind kontextfreie Grammatiken, das bedeutet das die Grammatik $G = (N, \Sigma, P, S)$ mit N gleich Menge der Nichtterminalsymbole, Σ gleich Menge der Terminalsymbole, P gleich die

Menge von Regeln und S gleich das Startsymbol, folgenden Einschränkungen unterliegen:

$$\forall(w_1 \rightarrow w_2) \in P : |w_1| \leq |w_2|$$

$$\forall(w_1 \rightarrow w_2) \in P : w_1 \in N$$

CYBOL ist eine kontextfreie Grammatik und kann somit in der EBNF notiert werden.

Mit EBNF lassen sich auch Grammatiken vom Typ 3 der Chomsky-Hierachie darstellen. Dazu müssen zusätzlich die folgenden Bedingungen eingehalten werden:

$$\forall(w_1 \rightarrow w_2) \in P : w_1 \in N$$

$$w_2 \in \Sigma \cup \Sigma \cdot N \cup \{\epsilon\}$$

Schon die erste Regel in der EBNF von CYBOL widerspricht der zusätzlichen Regel.

```
CYBOL   = '<model>'
            {part}
          '</model>';
```

Auf der rechten Seite der Regel dürfen nur Terminalsymbole oder Terminalsymbole gefolgt von einem Nichtterminalsymbol stehen. Somit entspricht CYBOL keine Grammatik vom Typ 3 der Chomsky-Hierarchie.

Syntax von EBNF

Die folgende Übersicht gibt die verwendeten Beschreibungskonstrukte von EBNF [ebn05] wieder:

Zeichen	**Bedeutung**
=	Definition
;	Endezeichen
—	Logisches Oder
[...]	Option
...	Optionale Wiederholung
(...)	Gruppierung

”...”	Anführungszeichen, 1. Variante
’...’	Anführungszeichen, 2. Variante
(*...*)	Kommentar

Tabelle 3.4: Syntax von EBNF

EBNF für CYBOL

Hier folgt die vollständige Notation von CYBOL in der EBNF.

```
CYBOL   = '<model>'
              {part}
          '</model>';

part    = '<part ' all_attribute '\>' |
          '<part ' all_attribute '>'
              {property}
          '</part>';

property    = '<property ' all_attribute '\>' |
              '<property ' all_attribute '>'
                 {constraint}
              '</property>';

constraint  = '<constraint ' all_attribute '\>';

all_attribute          = attribute_name attribute_channel
                         attribute_abstraction attribute_model

attribute_name          = 'name="' name '"';
attribute_channel       = 'channel="' channel '"';
```

```
attribute_abstraction   = 'abstraction="' abstraction '"';
attribute_model         = 'model="' model '"';

name            = description_sign;
channel         = description_sign;
abstraction     = description_sign;
model           = value_sign;

description_sign    = { ( letter | number ) } ;
value_sign          = { ( letter | number | other_sign ) } ;

letter          = small_letter | big_letter;
small_letter    = 'a' | 'b' | 'c' | 'd' | 'e' | 'f' | 'g' |
                  'h' | 'i' | 'j' | 'k' | 'l' | 'm' | 'n' |
                  'o' | 'p' | 'q' | 'r' | 's' | 't' | 'u' |
                  'v' | 'w' | 'x' | 'y' | 'z';
big_letter      = 'A' | 'B' | 'C' | 'D' | 'E' | 'F' | 'G' |
                  'H' | 'I' | 'J' | 'K' | 'L' | 'M' | 'N' |
                  'O' | 'P' | 'Q' | 'R' | 'S' | 'T' | 'U' |
                  'V' | 'W' | 'X' | 'Y' | 'Z';
other_sign      = ',' | '.' | '/', '+', '-', '*';
number          = '0' | '1' | '2' | '3' | '4' |
                  '5' | '6' | '7' | '8' | '9';
```

3.5 Logik von CYBOL

Mit der Beschreibungssprache CYBOL wird das statische und dynamische Verhalten einer Anwendung dargestellt. Dazu bedarf es der folgenden Grundkonstrukte, um das Wissen und die Logik einer Anwendung zu beschreiben.

1. Sequential
2. Selection
3. Iteration

In den weiteren Abschnitten wird die Darstellung dieser Konstrukte für CYBOL beschrieben.

3.5.1 Sequential

Die Ausführung von mehreren Befehlen hintereinander wird als sequentielle Abarbeitung bezeichnet. Im ersten Schritt wird dies durch die Reihenfolge in den CYBOL-Beschreibungsdateien bestimmt. In Zukunft könnte auch die Reihenfolge über ein Property *position* innerhalb der CYBOL-Datei gesteuert werden. Dies hätte den Vorteil, dass unabhängig von der Position innerhalb der CYBOL-Datei die Abarbeitungsreihenfolge festgelegt werden kann. Für diese Flexibilität würde aber die CYBOL-Datei unübersichtlicher, da die Reihenfolge der Abarbeitung nicht der Reihenfolge der Position in der CYBOL-Datei entsprechend würde.

3.5.2 Selection

Eine weitere wichtige Funktion zur Ablaufsteuerung ist die *selection*. Dies ermöglicht Verzweigungen innerhalb der Programmausführung. Dafür gibt es in CYBOL die Operation *selection* mit den Properties *comparison*, *true_model* und *false_model*. Die Eigenschaft *comparison* ist vom Typ Boolean und dient als Vergleichs-Flag zur Unterscheidung, welches Modell ausgeführt werden soll. Eine Verzweigung wird in CYBOL folgendermaßen definiert:

```
<part name="selection" channel="inline"
      abstraction="operation" model="selection">
    <property name="comparison" channel="inline"
              abstraction="knowledge" model="domain.comparisonflag"/>
    <property name="true_model" channel="file"
```

```
            abstraction="cybol" model="true_model.cybol"/>
    <property name="false_model" channel="file"
            abstraction="cybol" model="false_model.cybol"/>
</part>
```

3.5.3 Iteration

Iteration ist eine Schleifenkonstrukt. In der prozeduralen Programmierung gibt es bis zu drei verschiedene Schleifentypen, die *repeat until*, die *for* und die *while do* Schleife. Diese Schleifentypen sind alle ineinander umwandelbar. Darum gibt es in CYBOL nur eine Iterationsform. Diese ist folgendermaßen definiert.

```
<part name="creat_table_body" channel="inline"
      abstraction="operation" model="loop">
    <property name="break" channel="inline"
            abstraction="knowledge" model="domain.loop_breakflag"/>
    <property name="index" channel="inline"
            abstraction="knowledge" model="domain.loop_index"/>
    <property name="model" channel="file"
            abstraction="cybol" model="create_table_rows.cybol"/>
</part>
```

Es gibt eine Operation *loop*, die die Eigenschaften *break*, *index* und *model* hat. Das *model* wird solange ausgeführt, bis die Eigenschaft *break* wahr ist. Das Modell für *break* ist vorher anzulegen, damit die Schleifenoperation das Breakflag auch auswerten kann. Weiterhin muss, soll keine Endlosschleife entstehen, in dem *model* die Eigenschaft *break* manipuliert werden, so dass die Schleife eine Chance hat, sich zu beenden. In der Eigenschaft *index* wird eine Zählvariable mitgezählt.

Kapitel 4

Cybernetics Oriented Interpreter (CYBOI)

Für die Beschreibungssprache CYBOL ist zur Ausführung ein Interpreter, genannt CYBOI, notwendig. Dieser Interpreter hat die Aufgabe, die CYBOL-Dateien zu lesen, zu parsen und zu verarbeiten. Der Interpreter ist in C geschrieben. In diesen Abschnitt wird der Interpreter vorgestellt, seine Arbeitsweise beschrieben und die verwendeten Software- und Architektur-Pattern erläutert.

4.1 Pattern in CYBOI

Sehr oft wiederholen sich die Probleme. Auch die Lösungsansätze für diese sind identisch. Alles spricht dafür, solche Lösungswege abstrakt zu beschreiben und zu benennen. Ein Pattern umfasst die beteiligten Komponenten, ihre Verantwortlichkeiten und Verbindungen. Die Pattern helfen somit auf vorhandenes Expertenwissen zurückzugreifen.

Christoper Alexander definierte Pattern folgendermaßen [CA77]:

> Jedes Muster beschreibt ein in unserer Umwelt beständig wiederkehrendes Problem und erläutert den Kern der Lösung für dieses Problem, so dass man diese Lösung beliebig oft anwenden kann, ohne sie jemals ein zweites Mal gleich auszuführen.

In den nächsten Abschnitten werden einige Pattern, die in CYBOI verwendet werden, beschrieben.

4.2 Pipes and Filters

Ein wichtiges Pattern in CYBOI ist *Pipes and Filters* [FB98].

> Das Pipes-and-Filters-Muster bietet eine Struktur für Systeme, die Datenströme verarbeiten. Jeder Verarbeitungsschritt ist dabei in einen Filter gekapselt. Daten werden durch Kanäle (engl. pipe) von Filter zu Filter weitergegeben. Die Filter können immer wieder neu angeordnet werden, was es ermöglicht, eine Familie von verwandten Systemen zu erstellen.

Das *Pipes and Filters* Muster wird also verwendet, wenn die Ausgabe von einem Filter gleich für die Eingabe des nächsten Filters verwendet wird. In dem Filter werden die Transformationen des Datenstromes durchgeführt.

Im CYBOI wird dieses Pattern bei der Behandlung (Erzeugen und Zerstören) von CYBOI-Modellen verwendet. Dabei ist in CYBOI die Unterscheidung von *primitive model* und *compound model* zu machen. *Primitive model* sind alle Modelle, die auf elementare Strukturen aufbauen. Dies beinhaltet alle primitiven Datentypen (Integer, Vector, ...), die in CYBOI implementiert sind und alle elementaren Operationen. *Compound model* sind dagegen alle Modelle, die sich aus weiteren Modellen zusammensetzen.

In den Abbildungen 4.1 und 4.2 sind die in CYBOI verwendeten *Pipes and Filters* zu sehen. Diese werden genutzt, um Modelle, die persistent vorliegen, in das System von CYBOI zu integrieren bzw. wieder persistent abzuspeichern.

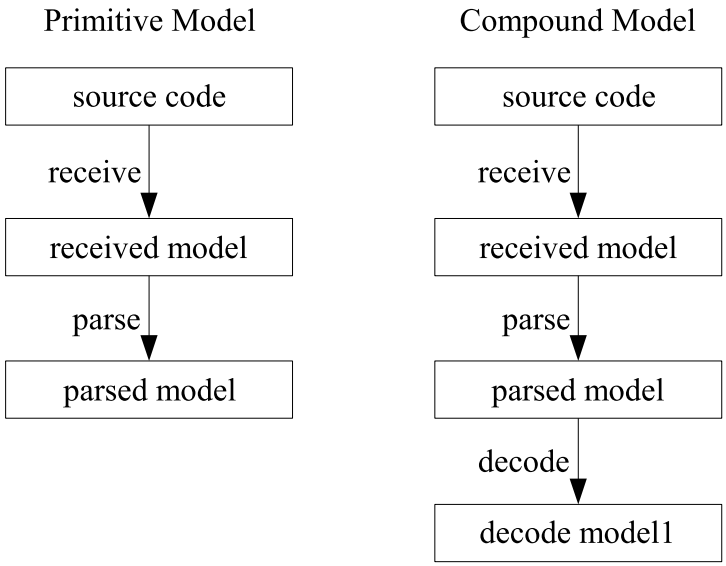

Abbildung 4.1: decode model

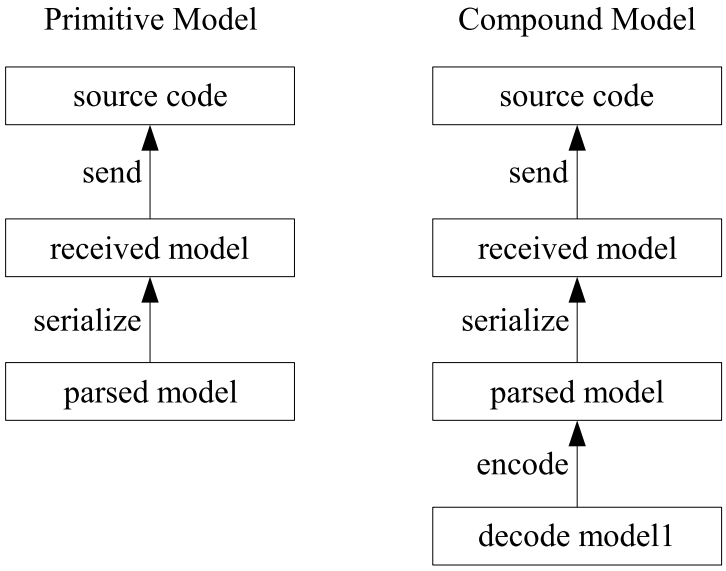

Abbildung 4.2: encode model

Der *source code*, der in einen persistenten Zustand vorliegt, wird geladen. Das Ergebnis wird einen Parser übergeben. Ist es ein *primitive model*, so ist das geparste Ergebnis unser Modell. Bei einen *compound model* wird das Ergebnis des Filters *parse* noch decodiert, damit das Modell hierarchisch aufgebaut wird. Diese Modelle liegen in einen transienten Zustand vor. Für die Abspeicherung von Modellen ist der umgekehrte Weg nötig.

4.3 Compositum

Das Compositum Muster dient zur Darstellung von Teil-Ganzes-Hierarchien. In CYBOI wird das gesamte Wissen in einer Baumstruktur gespeichert. Diese Baumstruktur setzt sich aus primitiven Typen und aus einer Container-Struktur, in CYBOI *compound* genannt, zusammen. Diese Container-Struktur kann entweder primitive Typen oder weitere Container-Strukturen enthalten. Die untersten Blätter in der Baumstruktur sind immer primitive Typen.

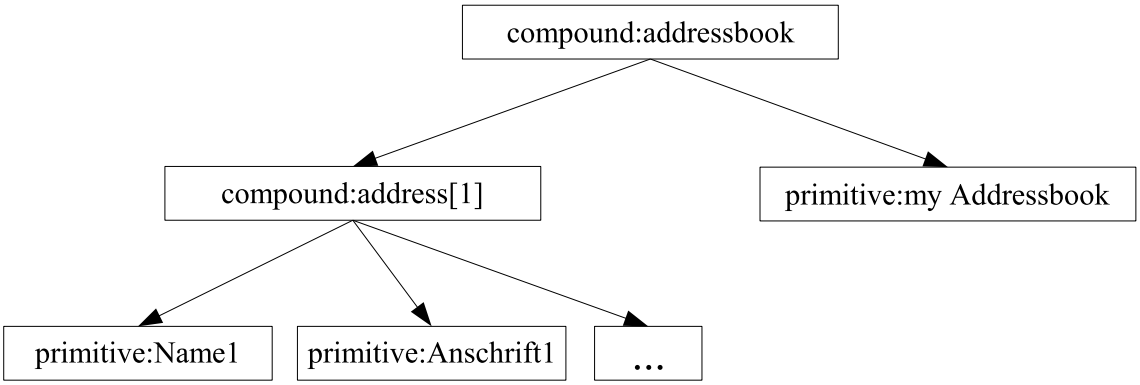

Abbildung 4.3: Adressbuch als Compositum

Nach [EG04] wird dieses Muster verwendet, wenn keine Unterscheidung zwischen den Operationen von primitiven und zusammengesetzten Objekten gemacht werden soll. Da dieses Muster aus der objektorientierten Programmierung kommt, ist dies in CYBOI nur partiell nutzbar. Das Prinzip des Patterns ist, bei der Ausführung einer Operation auf einer Komponente, diese Operation hierarchisch nach unten durchzureichen. Da in CYBOI keine direkte Verknüpfung von Operation und Daten vorliegt, ist dieser Teil des Patterns auch nicht nutzbar. Das Compositum Muster wird in CYBOI nur für die Abbildung der Beziehungen der einzelnen Komponenten verwendet, das aber konsequent für alle Daten, für alle Operation usw.

4.4 Architektur von CYBOI

CYBOI beschränkt sich in seiner Architektur im Wesentlichen auf folgende zwei Container.

- Signal Memory
- Knowledge Memory

4.4.1 Signal Memory

Der *Signal Memory* beschreibt die dynamische Komponente. Unter einen Signal versteht man die Anweisung für die Ausführung eines Befehles in CYBOI. *Signal Memory* ist eine Liste von allen noch abzuarbeitenden Signalen. In einer Endlosschleife werden diese Signale ausgeführt. Nach der Abarbeitung werden die einzelnen Signale aus dem *Signal Memory* entfernt. Die Reihenfolge der Abarbeitung von Signalen kann über Prioritäten gesteuert werden. Ein Signal setzt sich aus den Bestandteilen, wie in Tabelle 4.1 aufgelistet, zusammen.

Bestandteil	**Beschreibung**
abstraction	Dies beschreibt, in welcher Abstraktion das Modell vorliegt. Die Abstraktion bestimmt somit, wie das Modell in CYBOI verarbeitet wird.
model	Dies ist das eigentliche Signal.
detail	Manche Signale brauchen für eine erfolgreiche Abarbeitung noch weitere Parameter.
priority	Über die Priorität kann die Reihenfolge der Signalabarbeitung beeinflusst werden.
main signal id	Das Erzeugersignal bekommt eine eindeutige ID innerhalb von CYBOI. Erzeugt dieser Prozess weitere Signale, so wird auf den Erzeugerprozess verwiesen.

Tabelle 4.1: Bestandteile eines Signals

4.4.2 Knowledge Memory

Der zweite Container *Knowledge Memory* dient zur Speicherung unseres Wissens. Diesen Container kann man sich als Baumstruktur vorstellen, in dem unser ganzes Wissen enthalten ist. *Knowledge Memory* enthält das aktuelle Wissen, das CYBOI zur Verfügung steht. Eine Auswahl des Wissens ist in Abbildung 4.4 zu betrachten. Jedes Wissen wird in dieser Struktur gespeichert.

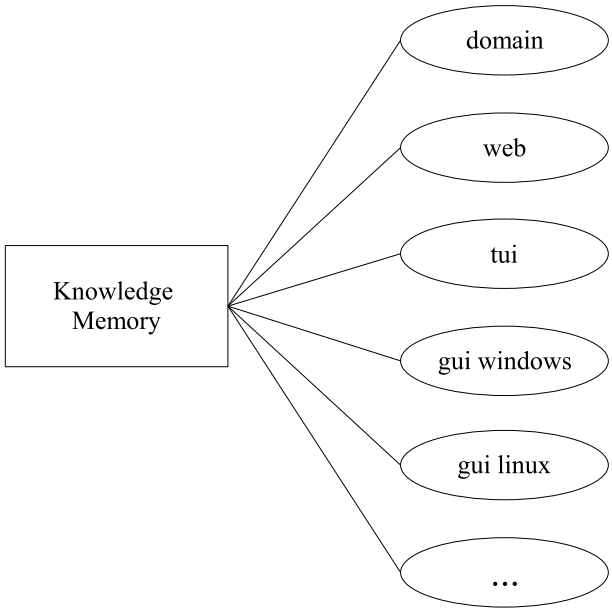

Abbildung 4.4: Knowledge Memory

4.5 Lifecycle von CYBOI

Das Grundgerüst von CYBOI ist eine einfache Endlosschleife. In dieser Schleife wird immer wieder geprüft, ob ein Signal für die Bearbeitung ansteht. Ist dies der Fall, so wird das Signal bearbeitet. Zuvor werden verschiedene Strukturen erstellt und nach Beenden der Endlosschleife werden die Strukturen zerstört. Die Endlosschleife wird beendet, wenn das Shutdown-Signal abgearbeitet wird.

Folgende Arbeitsschritte müssen dafür durchlaufen werden:

- Erstellung der globalen Variablen
- Erstellung der internen Variablen
- Erstellung des Signalspeichers
- Erstellung der statischen Struktur (Knowledge Memory)
- Erstellung des Startup-Signals
- Abarbeitung aller Signal in der Endlosschleife, bis das Shutdown-Signal erreicht wird
- Zerstörung des Startup-Signals
- Zerstörung der statischen Struktur (Knowledge Memory)
- Zerstörung des Signalspeichers
- Zerstörung der internen Variablen
- Zerstörung der globalen Variablen

Die eigentliche Kernkomponente ist die Endlosschleife. Dort werden nacheinander alle Signale abgearbeitet.

- Holen des abzuarbeitenden Signals
- Signal abarbeiten
- Löschen des Signals aus der Signalwarteschlange

Dies wird solange ausgeführt, bis das Signal mit der Operation *exit* ausgeführt wird. Liegt kein Signal vor, so wird solange die Schleife ausgeführt, bis wieder ein Signal anliegt.

Kapitel 5

Webanwendungen

5.1 Überblick

In dieser Arbeit sollen die Realisierungsmöglichkeiten von CYBOL-Webfrontends untersucht werden. Damit CYBOL über ein Webfrontend angesprochen werden kann, muss CYBOL als Webanwendung laufen. Nach [Kap03] ist eine Webanwendung folgendermaßen definiert:

> Eine Web-Anwendung ist ein Softwaresystem, das auf Spezifikationen des World Wide Web Consortium (W3C) beruht und Web-spezifische Ressourcen wie Inhalte und Dienste bereitstellt, die über eine Benutzerschnittstelle, den Web-Browser, verwendet werden.

Welche Möglichkeiten gibt es, Inhalte und Anwendungslogik als Webanwendung darzustellen. Dazu muss geklärt werden, was Webanwendungen sind, welche Arten von Webanwendungen es gibt, welche Architektur diesen zugrunde liegt und welchem Bereich die für die Diplomarbeit zu erstellende Webanwendung zugeordnet werden kann.

5.2 Architektur von Webanwendungen

Webanwendungen sind immer eine Client-Server-Architektur. Hier ist eine Definition für eine Client-Server-Architektur aus dem Buch [H.R97]

> Unter der Client-Server-Architektur (engl.: client-server architecture) versteht man eine kooperative Informationsverarbeitung, bei der die Aufgaben zwischen

Programmen auf verbundenen Rechnern aufgeteilt werden. In einem solchen Verbundsystem können Rechner aller Art zusammenarbeiten. Server (= Dienstleister; Backend) bieten über das Netz Dienstleistungen an, Clients (= Kunden; Frontend) fordern diese bei Bedarf an. Die Kommunikation zwischen einem Client-Programm und dem Server-Programm basiert auf Transaktionen, die vom Client generiert und dem Server zur Verarbeitung überstellt werden. Eine Transaktion (engl.: transaction) ist eine Folge logisch zusammengehöriger Aktionen, beispielsweise zur Verarbeitung eines Geschäftsvorfalls. Client und Server können über ein lokales Netz verbunden sein oder sie können über große Entfernungen hinweg, zum Beispiel über eine Satellitenverbindung, miteinander kommunizieren. Dabei kann es sich um Systeme jeglicher Größenordnung handeln; das Leistungsvermögen des Clients kann das des Servers also durchaus übersteigen. Grundidee der Client-Server-Architektur ist eine optimale Ausnutzung der Ressourcen der beteiligten Systeme.

Die Kommunikation von Webanwendungen erfolgt über das Http-Protokoll. Ein Client (Webbrowser) stellt eine Anfrage (request) an den Server. Der Server nimmt diese Anfrage entgegen und verarbeitet diese Anfrage. Das Ergebnis schickt der Server als Antwort (response) an den Client. Bei Webanwendungen ist dies typischerweise eine HTML-Seite bzw. eine Antwort, die der Client, in unserem Fall der Webbrowser, versteht. In der Abbildung 5.1 wird dies verdeutlicht.

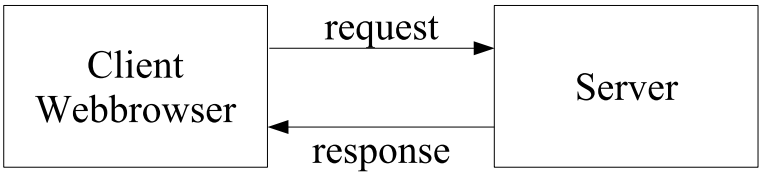

Abbildung 5.1: Request-Anfrage vom Client an Server

5.3 Kategorien von Webanwendungen

Webanwendungen besitzen unterschiedliche Komplexitäten. Sie reichen von einfachen Informationsseiten bis zu komplexen Programmabläufen. Nach [Kap03] werden folgende Kategorien von Webanwendungen unterteilt:

- Dokumentenzentriert

  Hier werden fertige Dokumente auf dem Server gelegt und bei Anfrage als Antwort ausgegeben. Beispiele sind einfache Firmenpräsentationen oder das Webradio.

- Interaktiv

  Hier hat der Anwender die Möglichkeit über einfache Interaktionen die Webanwendungen zu steuern. Dies kann z.B. durch eine einfache HTML-Formular-Technik und durch die Link-Technik umgesetzt sein.

- Transaktional

  Durch ein Datenbanksystem kann eine konsistente Datenhaltung erreicht werden. Ein Beispiel dafür ist ein Resevierungssystem. Daten sind nicht nur über eine Session bekannt, sonder werden dauerhaft gespeichert.

- Workflow-basiert

  Diese erlauben Abwicklungen von Geschäftsprozessen intern oder zwischen verschiedenen Teilnehmern. Dafür sind Web-Services mit definierten Schnittstellen notwendig. Beispiele sind Business-to-Business-Lösungen (B2B) im E-Commerce-Bereich oder E-Government-Anwendungen im Bereich der öffentlichen Verwaltung.

- Kollaborativ

  Hier werden die Webanwendung zur Kooperation bei unstrukturierten Vorgängen und hohem Kommunikationsbedarf verwendet. Beispiel dafür sind Chatrooms und gemeinschaftliche Arbeitsräume um gemeinsame Informationen generiereren, bearbeiten und verwalten zu können (z.B. Wikipedia).

- Portalorientiert

  Diese vereinen den Zugriff auf verteilte und verschiedenen Informationsquellen und Informationsdienste unter einer einheitlichen Oberfläche (Single Point of Access).

- Ubiquitär

  Stellen personalisierte Dienste zu jeder Zeit, an jedem Ort und für eine Vielzahl von Endgeräten zur Verfügung.

- Semantisches Web

  Informationen im Web sollen nicht nur für den Menschen verständlich aufbereitet, sondern auch für Maschinen automatisch verarbeitbar gemacht werden. Dadurch kann neues relevantes Wissen automatisch ermittelt werden. Dies wird bei Recommender-Systeme eingesetzt um z.B. Kunden Produkte zu empfehlen oder ihnen für ihre Kaufentscheidung hilfreiche Informationen zur Verfügung zu stellen.

In der Praxis gibt es keine so strikte Trennung für die Webanwendungen, sondern es werden verschiedene Mischformen vorhanden sein.

5.4 Darstellungsmöglichkeiten von Webanwendungen

Nachdem wir geklärt haben, was Webanwendungen sind, sollen im Folgenden die gebräuchlichsten Darstellungsmöglichkeiten von Webanwendungen beschrieben werden.

- HTML und HTML mit CSS

  HTML ist eine Seitenbeschreibungssprache, die vom W3C standardisiert wurde. Die aktuelle Version ist 4.0. Alle Browser unterstützen HTML 4.0. Leichte Darstellungsunterschiede gibt es dennoch zwischen den Browsern. Mit CSS können die Formatierungsinformationen von den Daten und die Grundstruktur der Webseite getrennt werden. Dies ist ein Schritt zur Trennung von Inhalt und Darstellung.

- XHTML

  XHTML entspricht im Wesentlichen HTML, die Dateien müssen allerdings XML-

konform sein. Dies wurde in HTML noch nicht konsequent durchgesetzt. Der Vorteil von XHTML gegenüber HTML ist, das XML-Werkzeuge zur Syntaxüberprüfung gut geeignet sind.

- HTML mit Javascript

  Javascript wird clientseitig ausgeführt, d.h. der Client muss den Befehlsatz von Javascript interpretieren können. Javascript wurde von Netscape entwickelt und als offener Standard für eine Objekt-Skriptsprache bezeichnet. Leider wurde Javascript nicht vom W3C standardisiert. Als Gegenprodukt von Microsoft wurde JScript eingeführt. Javascript und JScript sind sehr ähnlich, aber doch nicht gleich. Darum gibt es sehr viele Inkompatibilitäten zwischen den Browsern, weshalb Javascript nur bedingt in Webanwendungen verwendet werden sollte. Der Vorteil von Javascript ist, dass schon clientseitig auf Benutzereingabe reagiert werden kann, ohne den Server zu befragen und dadurch Interaktionen ohne kompletten Neuaufbau des Frames realisierbar sind.

- XML mit XSLT

  Hier wird die Trennung von Inhalt und Darstellung umgesetzt. In der XML-Datei stehen die ganzen Daten für die Darstellung und in der XSLT-Datei die ganzen Formatierungen und Strukturierungen für die Darstellung der Daten. Auf den Server werden diese zwei Dateien zusammengebracht und das Ergebnis wird zum Client (Browser) geschickt.

- Serverseitige Skriptsprache (PHP, CGI, ...)

  Diese Skriptsprachen werden auf dem Server ausgeführt und das Ergebnis wird zum Browser geschickt. In ihnen können komplexe Anwendungslogiken hinterlegt werden. Dies ist auch der Vorteil der Skriptsprachen, weil damit komplexe Webanwendungen realisierbar sind. Man greift auf die vielen vorgefertigten Bibliotheken, die in den Skriptsprachen schon enthalten sind, zurück, so dass Standardwebanwendungen relativ zügig entwickelbar sind. Für die Trennung von Darstellung und Anwendungslogik können hier Template-Systeme verwendet werden.

- Serverseitige Anwendungssprachen (Servlet, ASP, JSP, ...)

  Diese werden auf dem Server ausgeführt. Im Unterschied zu den serverseitigen Skriptsprachen sind sie in einer Programmiersprache wie Java, Visual Basic usw. geschrieben und liegen auf dem Server in einer kompilierten Form vor (Servlets) oder werden zur Laufzeit kompiliert (JSP) und danach ausgeführt.

Auch hier sind wieder Mischformen vorhanden und denkbar. Ein Beispiel dafür ist die Fehlerüberprüfung von Eingaben durch Javascript und die Ausführung der restlichen Anwendungslogik durch serverseitige Skriptsprachen oder serverseitige Anwendungssprachen.

5.5 Einordung von CYBOP

Zurzeit können dokumentenbasierte und interaktive Webanwendungen mit CYBOP erstellt werden. Ein Beispiel für eine interaktive Webanwendung ist der Prototyp, der im Rahmen der Diplomarbeit entstanden ist. Der nächste Schritt für CYBOP ist die Umsetzung von transaktionalen Webanwendungen. Dazu ist die Abspeicherung von Daten in persistenten Medien notwendig.

Als Darstellung kommen aus Architekturgründen nicht alle möglichen Formen in Frage. Alle serverseitigen fallen heraus, da CYBOP mit seinen Komponenten bereits eine serverseitige Architektur darstellt. Also sind nur clientseitige Darstellungsmöglichkeiten wie HTML, XHTML und Javascript in Betracht zu ziehen. HTML ist zwar standardisiert, hat aber noch viele Freiräume bezüglich von Schreibweisen. In dieser Beziehung ist XHTML die bessere Wahl, da hier die Wohlgeformtheit von XML-Dokumenten eingehalten werden muss. Dadurch sind die Strukturen für die Generierung besser definiert und es müssen nicht so viele Ausnahmebedingungen in CYBOI integriert werden. Javascript ist keine unabhängige Technologie, sonder von einer Firma abhängig. Es wird zwar von vielen gängigen Webbrowsern unterstützt, ist aber nicht standardisiert. Damit ist diese Technologie für CYBOP nicht zu empfehlen. Für die Umsetzung in CYBOP bleibt nur die eine sinnvolle Darstellungsmöglichkeit, XHTML, übrig.

Kapitel 6

Webserver und dessen Integration in CYBOI

6.1 Hintergrund

Zur Beantwortung einer Anfrage eines Clients an einen Server über das Http-Protokoll ist ein Webserver nötig. Dies kann durch einen vorhandenen Webserver, wie z.B. Apache, oder durch einen selbst programmierten Webserver realisiert werden. Dieser Webserver muss für CYBOL angepasst werden. Bei einem bestehenden Webserver ist eine Erweiterung z.B. durch ein CYBOI-Plugin möglich. Der Vorteil der Erweiterung eines bestehenden Webservers ist die Verwendung der ganzen Funktionalität, wie z.B. die Session-Verwaltung und die Skalierbarkeit. Das Hauptproblem bei der Integration wäre die Anpassung der verschiedenen Softwarearchitekturen (bestehender Webserver und CYBOI). Ein weiterer Nachteil ist die Abhängigkeit des Plugins zu einem bestehenden Webserver, das bedeutet dass bei Änderungen des Webservers auch das Plugin anzupassen ist. Die Kontrolle würde nicht allein beim CYBOP-Projekt liegen.

Da die Implementierung für CYBOL insgesamt als Prototyp geplant ist, reicht es aus, einen kleinen Webserver selbst zu programmieren, so dass man die gesamte Verfügungsgewalt über das Projekt und dessen Quelltext hat. Weiterhin kann im Softwaredesign eine bessere Homogenität erreicht werden. Änderungen am Webserver sind somit leichter zu realisieren. Der Nachteil ist hier, das die Standardfunktionalität von Webservern, wie z.B. die Session-Verwaltung, in CYBOI programmiert werden muss.

Die Vorraussetzungen für einen Webserver sind die Sockets. Diese stellen die Kommunikationsschnittstelle zwischen den einzelnen Komponenten dar. Wichtiger Bestandteile für

die Integration des Webserver-Prototyps in CYBOI sind Prozesse und Threads. Da bei der Implementierung der Sockets blockierende Sockets verwendet werden, ist es notwendig den Webserver in einem eigenen Prozess bzw. Thread laufen zu lassen.

6.2 Sockets

Sockets sind nach [def05] folgendermaßen definiert.

> Sockets is a method for communication between a client program and a server program in a network. A socket is defined as the endpoint in a connection. Sockets are created and used with a set of programming requests or function calls sometimes called the sockets application programming interface (API). The most common sockets API is the Berkeley UNIX C language interface for sockets. Sockets can also be used for communication between processes within the same computer.

Ein Socket ist also eine Schnittstelle zwischen einem Prozess und einem Transportprotokoll. Letzteres kann z.B. TCP oder UDP sein. Das Socket-Prinzip entspricht dem von File-Deskriptoren. Dort repräsentiert nach dem Öffnen einer Datei ein Handle die Verbindung zu dieser Datei und unter Angabe des Handles ist der Lese- oder Schreibzugriff möglich. Bei Sockets geht es jedoch nicht um physikalische Dateien sondern um Kommunikationskanäle, über die Daten gesendet und empfangen werden können.

Socket-Schnittstellen sind zwar von keiner Institution genormt, stellen aber einen de-facto- bzw. Industriestandard dar, was zum einen daran liegen dürfte, dass sie leicht verständlich sind, zum anderen fügen sie sich ausgesprochen harmonisch in die UNIX-Umwelt ein. Eine Variante der Socket-Schnittstelle wurde von Microsoft und verschiedenen anderen Firmen unter der Bezeichnung WinSock in das Schnittstellenangebot der Windows Open Service Architecture (WOSA) aufgenommen und dürfte damit auch ein etablierter Standard in der PC-Welt sein.

6.2.1 Arten von Sockets

Sockets stellen eine allgemeine Form der Kommunikation zwischen Dateien, Prozessen usw. dar. Schon allein die Verwendung unterschiedlicher Übertragungsprotokolle spezifiziert unterschiedliche Anwendungsgebiete. Folgende Arten von Sockets werden darum unterschieden:

- Stream Socket
  Stream Sockets setzen auf dem verbindungsorientierten TCP auf. Er ist langsamer als ein Datagramm Socket, aber dafür wird die Übertragungskontrolle gewährleistet.

- Datagram Socket
  Datagram Sockets setzen auf dem verbindungslos orientierten User Datagram Protocol (UDP) auf. Das große Problem dabei ist, dass die Übertragungskontrolle fehlt. Ansonsten ist die Übertragungsart sehr schnell, falls die Pakete ankommen und erzeugt kaum Overhead.

- Raw Socket
  Ein Raw Socket ist ein spezieller Socket. Er ermöglicht dem Benutzer den direkten Zugang zum Netzwerk, indem man eigene Pakete einspeisen kann. Raw Socket benutzen z.B. das Protokoll ICMP. Sie sind sehr schnell, da sie auf eine tiefere Protokollfamilie aufsetzen und damit weniger Overhead haben.

- Unix Domain Sockets
  Unix verwendet Sockets zur lokalen Interprozess-Kommunikation, sog. Unix Domain Sockets. Sie sind Teil des POSIX-Standards. Dies ist die ursprüngliche Form des von BDS entwickelten Sockets.

Für den Webserver der vorliegenden Diplomarbeit wird ein Stream Socket mit dem Protokoll TCP verwendet. Da in einer Anwendung sichergestellt werden muss, das die Datenpakete ankommen und auch in der richtigen Reihenfolge zusammengesetzt werden, ist nur ein Stream Socket mit der entsprechenden Übertragungskontrolle verwendbar.

6.2.2 Arbeitsweise von Sockets

Sockets arbeiten immer nach einen bestimmten Muster. Als erstes ist der Server für die Bearbeitung des Sockets vorzubereiten. Dazu muss er veranlasst werden über einen bestimmten Port auf die Anforderungen der Clients zu warten. Ports gehören zu der Adresskomponente einer Netzwerkkommunikation. Der Port ist mit 16 Bit kodiert. Damit kann der Port Werte von 0 bis 65535 annehmen. Die Ports bis zur 1024 sind für verschiedenen Dienste (FTP, HTTP, ...) definiert. Die Portnummern höher als 1024 stehen zur freien Verfügung, wobei sich auch hier schon für gewisse Dienste Standard-Portnummern eingebürgert haben.

In der Abbildung 6.1 ist einen typischen Ablauf eines Stream Sockets zu sehen.

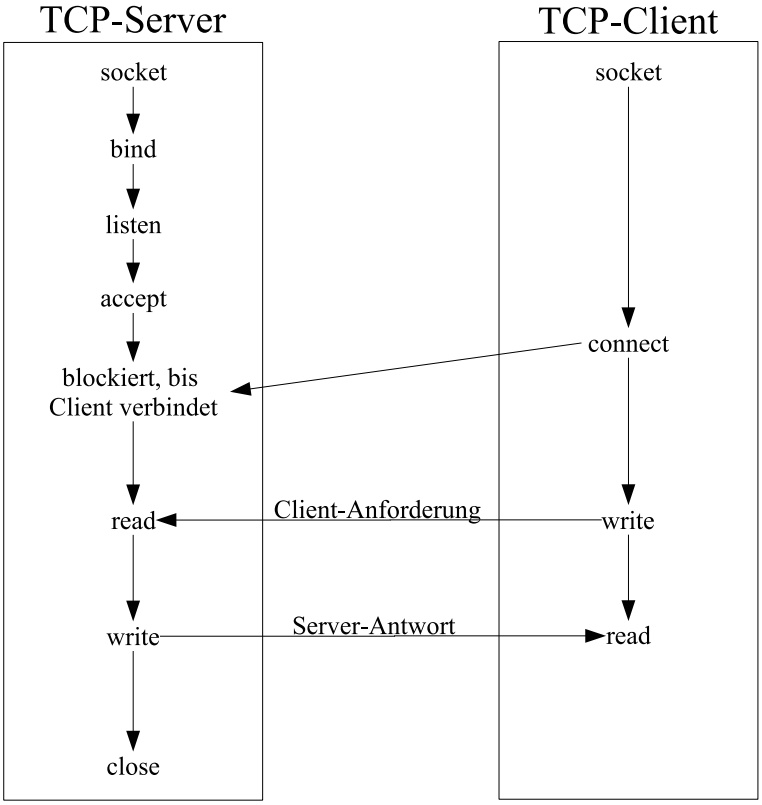

Abbildung 6.1: Ablauf TCP-Socket

Als erstes muss der Socket erstellt werden. Mit *bind* wird dieser Socket an eine bestimmte Portnummer gebunden und mit dem Befehl *listen* wird dem Socket mitgeteilt, das er auf Verbindungsversuche zu dieser Portnummer lauschen soll. Jetzt wartet der Server mit *ac-*

ceppt auf einen Verbindungsversuch des Clients. Dies ist entweder als blockierende oder nicht blockierende Funktion implementiert (siehe Abschnitt 6.2.3). Nach dem *accept* die Verbindung akzeptiert hat, wird der normale Kommunikationsprozess durchgeführt. Als erstes liest der Server (*read*) die Anfrage vom Client und schickt eine Antwort (*write*) an den Client zurück. Danach ist vom Server die Verbindung zu beenden (*close*).

6.2.3 Blockierende und nicht blockierende Sockets

Gewisse Funktionen einer Socketoperation können als blockierende oder nicht blockierende Operation ausgeführt werden. Blockierend bedeutet, dass die Funktion so lange wartet, d.h. dass das restliche Programm an dieser Stelle unterbrochen wird, bis die Aktion ausgeführt wurde. Eine typische Operation ist das Warten auf eine Anfrage vom Client. In dieser Zeit ist der Prozess stillgelegt, bis eine Anfrage auf dem Kommunikationskanal anliegt. So lange der Prozess wartet, wird keine Prozessorleistung verbraucht. Andere Prozesse auf dem Rechner können normal weiterarbeiten, da der Prozessor keine Belastung durch das Warten auf eine Anfrage hat. Wenn diese Operation als nicht blockierende Operation realisiert wäre, so würde, wenn kein Anfragesignal auf dem Kommunikationskanal anliegt, die Funktion an dieser Stelle nicht warten, sondern einfach im Programm weitermachen. Um nun eine Anfrage über das Socket zu registrieren, ist die Ausführung der Operation in einer Endlosschleife nötig. Dies verbraucht aber viele Ressourcen des Prozessors. Auf Grund der Performance des Gesamtsystems sind also blockierende Sockets vorzuziehen.

6.3 Prozesse und Threads

Da die Implementierung des Webservers mit blockierenden Sockets erfolgt, ist es erforderlich, parallele bzw. nebenläufige Programme in CYBOI zu starten, da ansonsten die blockierenden Socketoperationen den gesamten Interpreter CYBOI lahm legen. Es gibt zwei Möglichkeiten parallele bzw. nebenläufige Programme zu starten, entweder als eigenständiger Prozess oder als Thread. Für die Auswahl sind beide Varianten zu erklären und Vor- und Nachteile ge-

genüberzustellen.

6.3.1 Prozess

Ein Prozess aus Betriebssystemsicht ist ein in Ausführung befindliches Programm, welches sequentiell abgearbeitet wird. Es wird immer nur eine Instruktion zu einer bestimmten Zeit ausgeführt. Ein Prozess besteht aus dem Programm und der Programmumgebung und er verfügt über einen eigenen Adressraum. Zu der Programmumgebung gehören Programmzähler, alle Registerinhalte, Stack (temporäre Daten) und der Datenbereich (globale Variablen). Da das Betriebssystem mehrere Prozesse gleichzeitig verwaltet und es so aussieht, als ob die Prozesse parallel ablaufen, müssen diese Prozesse verschieden Zustände annehmen können. Folgendes Bild veranschaulicht die Prozesszustände und welche Wechsel zwischen den Prozesszuständen möglich sind.

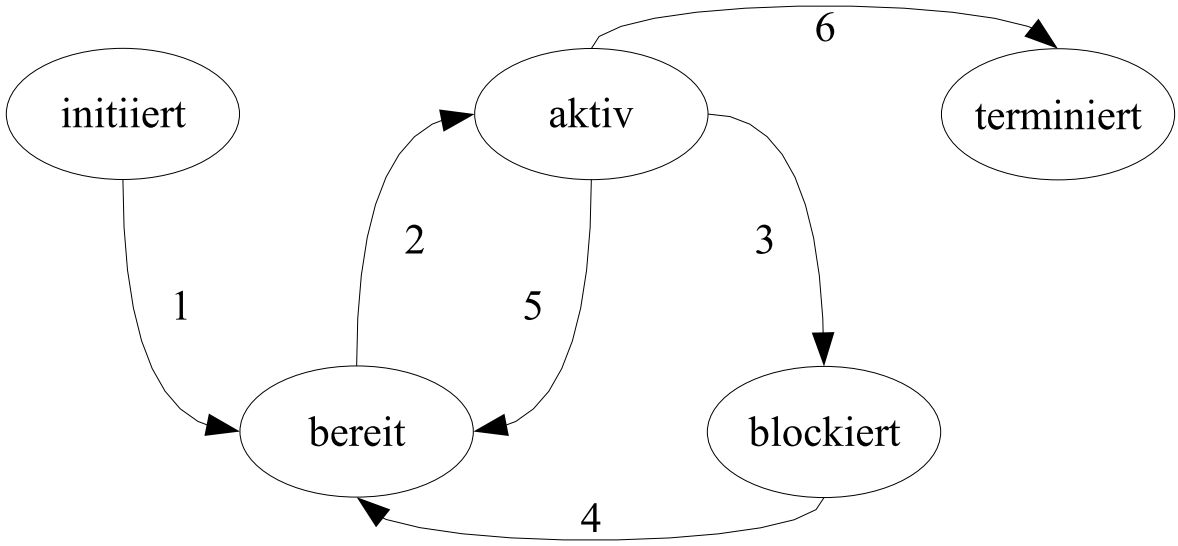

Abbildung 6.2: Prozesszustände

Als erstes wird ein Prozess initiiert. Dazu werden alle Initialbetriebsmittel alloziiert (1). Danach ist der Prozess in dem Zustand *bereit*. In diesem Zustand kann der Prozess noch nichts machen, da ihm noch keine Prozessorzeit zugeordnet ist. Dies wird durch den Prozessscheduler des Betriebssystems realisiert (2). Dieser Scheduler kann den Prozess auch wieder in den

Zustand *bereit* zurückversetzen (5). Dies kann z.B. durch Ablauf der Zeitscheibe oder durch Prioritätensteuerung passieren. Wird von einem aktiven Prozess ein Betriebsmittel angefordert und dieses ist zurzeit nicht verfügbar, so wird der Prozess in den Zustand blockiert gesetzt (3). Liegt ein Signal an, dass das erwartete Betriebsmittel zur Verfügung steht, so wechselt der Status des Prozesses in den Zustand *bereit* (4). Am Ende wird der Prozess abgeschlossen und alle Betriebsmittel und Ressourcen werden freigegeben.

Um zwischen den Prozessen umschalten zu können, müssen Informationen des Prozesses gesichert werden. Dazu dienen so genannte Prozess-Kontroll-Blöcke (PKB). Diese beinhalten folgende Informationen:

- Prozesszustand
- Zeiger auf Prozess
- Prozess-Id
- Programmumgebung
  - Programmzähler
  - Prozessorregister
  - IO Statusinformationen
  - Speicherverwaltungsinformationen

Um nun von einen zum anderen Prozess zu wechseln, wird der gesamte PKB gesichert und der PKB vom aktivierten Prozess geladen. Diese Art von Prozessen wird auch als Single-Thread-Prozessor-Modell bezeichnet.

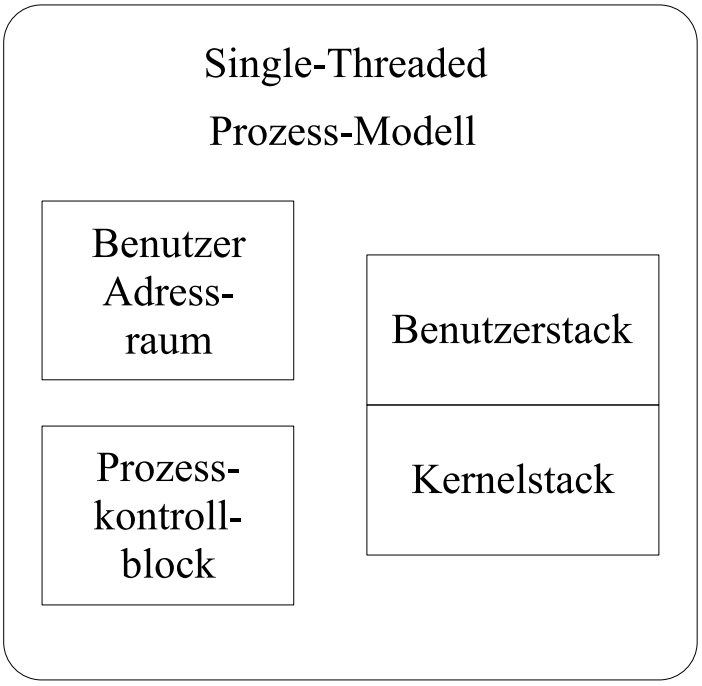

Abbildung 6.3: Single-Thread-Prozess-Modell

Ein Prozess kann weitere Unterprozesse generieren, so dass eine Baumstruktur von Prozessen und ihren Unterprozessen entsteht. Die Umschaltung zwischen den Prozessen (siehe vorheriges Kapitel) erfordert die Sicherung und das Laden des gesamten Prozess-Kontroll-Block und der Stacks.

6.3.2 Parallele und nebenläufige Prozesse

Ein Prozess wird sequentiell im Prozessor abgearbeitet. Richtige parallele Arbeitung von Prozessen gibt es nur in Mehrprozessorrechnern. In Einprozessormaschinen gibt es keine echte parallele Verarbeitung, sondern nur eine nebenläufige (quasi parallel) Abarbeitung von Prozessen. Dies wird durch das Betriebssystem intern geregelt. Da es für den Endanwender keinen Unterschied bedeutet, ob nebenläufige oder parallele Abarbeitung vorliegt, wird nur noch auf nebenläufige Prozesse eingegangen.

Hier eine Definition für nebenläufige Prozesse:

> Zwei oder mehrere Prozesse werden als nebenläufig bezeichnet, wenn sie weitgehend unabhängig vom Betriebssystem bearbeitet werden können. Dabei spielt es keine Rolle, ob sie von einem Multiprozessor-System parallel bearbeitet werden oder ihre Abarbeitung sequentialisiert werden, d.h. sie werden in einer durch das Betriebssystem bestimmten Reihenfolge abgearbeitet. Dabei ist es auch möglich, dass ein Prozess während der Bearbeitung unterbrochen wird, dann ein anderer bearbeitet und der abgebrochene erst nach einer Verweildauer fortgesetzt wird. Man muss zwischen unabhängigen nebenläufigen Prozessen, bei denen das Ergebnis nicht von der Abarbeitungsreihenfolge abhängig ist, und abhängigen nebenläufigen Prozessen, bei denen eine Synchronisation notwendig ist, unterscheiden. Betriebssysteme gehen in der Regel davon aus, dass die Prozesse unabhängig von einander sind. Sind Prozesse von einander abhängig, so müssen sie synchronisiert werden.

6.3.3 Threads

Als Thread wird ein leichtgewichtiger Prozess bezeichnet, der nur die wichtigsten Kontextinformationen mit sich führt. Im Prinzip werden innerhalb eines Prozesses mehrere Threads gehalten. Diese Threads teilen sich den Benutzer-Adressraum. Dadurch wird ein Wechsel zwischen verschiedenen Threads leichter. So können innerhalb einer Anwendung mehrere Teilprozesse nebenläufig ausgeführt werden. Die Abbildung 6.4 verdeutlicht die gemeinsame Nutzung vom Benutzer-Adressraum und das Ausführen von mehreren Threads innerhalb eines Prozesses.

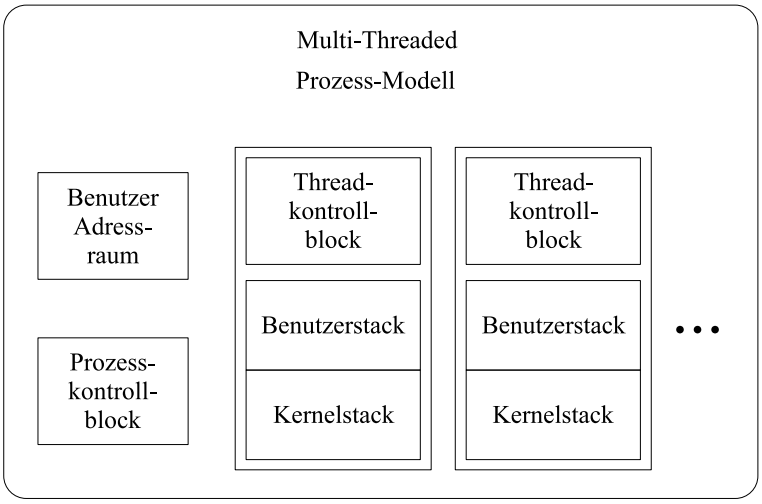

Abbildung 6.4: Multi-Thread-Prozess-Modell

Der Interpreter CYBOI wird in C geschrieben. Dieser Quelltext wird unter Linux kompiliert und ausgeführt. Eine Möglichkeit für Windows wäre *cygwin*, das die kompletten GNU-Tools und Bibliotheken auch für Windows zur Verfügung stellt. Die C-Bibliothek pthread hat sich für die Threadbehandlung etabliert. In dieser Bibliothek sind Möglichkeiten zur Erzeugung, Beendung und Synchronisation von Threads vorhanden.

6.3.4 Vergleich Threads und Prozesse

Für die Integration des Webservers in CYBOI wurden die leichtgewichtigen Prozesse (Threads) ausgewählt. Da Threads innerhalb eines Prozesses von sich aus den gleichen Adressraum verwenden, ist eine direkte Kommunikation zwischen CYBOI und den Webserver über den gleichen Adressraum möglich, womit bei Threads die Kommunikation über andere aufwendigere Kommunikationskanäle entfällt. Ein weiterer Vorteil ist die Geschwindigkeit der Operationen von Threads. Da weniger Informationen involviert sind, sind die Operationen für Threads auch schneller als bei Prozessen. Dies betrifft sowohl die normalen Operationen, wie z.B. Aktivieren, Erzeugen, Blockieren als auch den Kontextwechsel zwischen den Threads. Die Nachteile von Threads sind, dass dem Betriebssystem die Kontrolle über die Threads entzogen ist. Der Entwickler muss selbst darauf achten, dass sich diese nicht

gegenseitig blockieren. Weiterhin ist durch die Benutzung des gleichen Adressraumes nicht der Schutz vor anderen Threads gesichert. Dieser ist ebenfalls vom Entwickler sicherzustellen.

6.4 Synchronisation von Threads

Die Kommunikation von Threads erfolgt üblichweise über den gemeinsamen Benutzer-Adressraum. Da Threads nebenläufig arbeiten, müssen die Zugriffe auf gemeinsame Variablen synchronisiert werden, da es ansonsten zu Kollisionen beim gleichzeitigen Zugriff auf den gemeinsamen Speicherbereich kommen kann. Die pthread-Bibliothek stellt dafür zwei Möglichkeiten zur Verfügung, entweder über Mutex oder mit Bedingungsvariablen.

6.4.1 Threadsynchronisation mit Mutex

Mutex ist ein Kunstwort, welches sich aus den Wörtern MUTual und EXclusions (gegenseitiger Ausschluss) zusammensetzt. Sie dienen dazu, Ressourcen für Threads exklusiv zu reservieren. Dabei kann ein Mutex immer nur von einem Thread belegt sein. Versuchen weitere Threads diesen Mutex für sich zu beanspruchen, werden diese blockiert, bis der aktuelle Besitzer den Mutex wieder freigibt. Da die Belegung atomar erfolgt, ist auch sichergestellt, dass immer nur ein Thread einen Mutex zur selben Zeit belegt. Mutex liegt das Semaphoren-Konzept zu Grunde.

Semaphor im Allgemeinen bedeutet, dass ein Zugriff auf einen kritischen Bereich (Shared memory) geschützt wird. Dazu besteht ein Semaphor aus zwei Elementen, dem Semaphor-Wert s(ganzzahlig) und einer Warteschlange. Ist der Semaphoren-Wert größer Null, so bedeutet dies, dass der kritische Abschnitt betreten werden kann. Ansonsten ist der Abschnitt durch einen anderen Prozess belegt. Die Warteschlange dient zur Speicherung aller Prozesse, die nicht den kritischen Abschnitt betreten konnten. Zum weiteren sind für Semaphoren zwei Operationen P(s) und V(s) folgendermaßen definiert.

`P(s):` `s = 0` → Prozess in Warteschlange schlafen legen

`s > 0` → s um eins erniedrigen

`V(s):` → s um eins erhöhen

→ falls Warteschlange nicht leer ist, so nächsten Prozess aufwecken

Ein binärer Semaphore ist ein normaler Semaphor mit Initialisierung der Semaphoren-Variable auf eins. Mutexe stellen binäre Semaphoren dar, wobei die Implementierung ohne die Warteschlange realisiert wurde.

Das Prinzip von Mutex ist einfach. Ein Thread arbeitet mit einer globalen oder statischen Variable, die für allen anderen Threads während einer Operation von einem Mutex blockiert (gesperrt) wird. Benötigt der Thread diese Variable nicht mehr, gibt er diese frei. Durch das Setzen der Sperren im Programm liegt die Verantwortung einer Verklemmungsfreiheit beim Programmierer. Dadurch können aber bei unsauberer Programmierung Deadlocks auftreten, falls ein Thread die Sperrung nicht ordnungsgemäß freigibt oder es Ressourcen anfordert, obwohl es schon Ressourcen bekommen hat.

6.4.2 Threads synchronisieren mit Bedingungsvariablen

Die Bedingungsvariablen dienen dazu, den Eintritt bestimmter Bedingungen abzuwarten beziehungsweise deren Erfüllung anzuzeigen und können dazu genutzt werden Threads zu synchronisieren. Bedingungsvariablen sind an Mutexe gekoppelt um sicherzustellen, dass die Bedingungen immer nur von einem Thread zu einer Zeit geändert werden können. In der folgenden Abbildung ist das Prinzip der *Condition Variable* zu erkennen. Dies ist aus dem Buch [Wol04] entnommen.

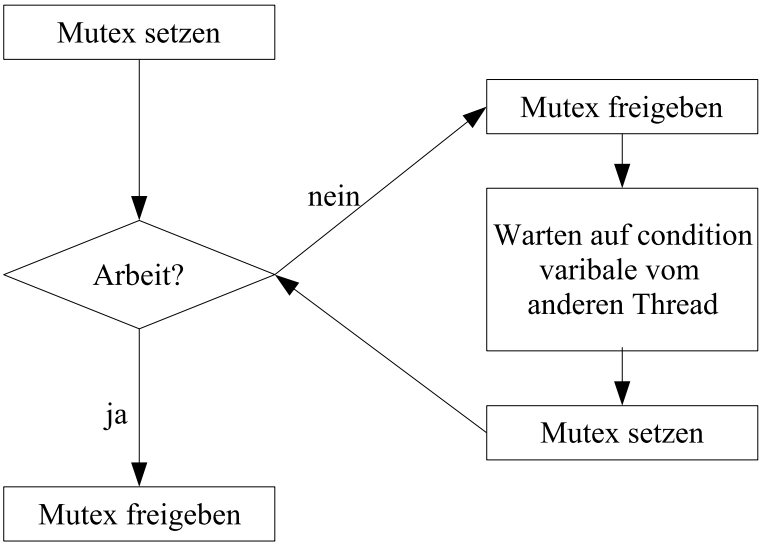

Abbildung 6.5: Condition Variable

Am Anfang wird ein Mutex für einen kritischen Bereich gesetzt. Kann der Thread nicht weiter arbeiten, weil zum Beispiel die benötigen Daten nicht anliegen, so wird dieser stillgelegt. Der Mutex wird freigegeben, es wird gewartet bis diese Bedingung (Daten vorhanden) eintrifft und danach der Mutex wieder belegt. Dies ist alles in einer Funktion der pthread-Bibliothek gekapselt. Danach kann der Thread ganz normal seinen kritischen Bereich abarbeiten und den Mutex wieder freigeben.

6.4.3 Fazit der Synchronisation

Die zwei vorgestellten Synchronisationsarten schließen sich nicht aus, sondern ergänzen sich. Mutex ist für den einfachen gegenseitigen Ausschluss da. Dieser Mechanismus wird bei den Bedingungsvariablen auch genutzt. Welche Form der Synchronisation ist für die Integration vom Webserver in CYBOI notwendig? Dazu ist die zu lösende Aufgabe näher zu betrachten. Wird im Webserver eine Anfrage vom Client gestellt, so ist ein Signal im *Signal Memeory* zu erstellen. Dieses Signal ist die Umwandlung der Anfrage vom Client in die interne Repräsentation von Signalen in CYBOI. Dazu ist es einfach nur notwendig, die Variable *Signal Memory* für den gegenseitigen Ausschluss zu sichern. Also ist es ausreichend, die Synchronisation mit Mutex zu realisieren.

Kapitel 7

Umsetzung der praktischen Aufgabe

7.1 Überblick

Als praktisches Ergebnis dieser Diplomarbeit soll ein Prototyp erstellt werden, der die Möglichkeit der Umsetzung von einer Webanwendung mit CYBOP aufzeigt. Dazu wurde das Beispiel einer kleinen Adressverwaltung gewählt, wo die Adressen editierbar, löschbar und neu anlegbar sind.

7.2 Anpassung von CYBOL

Als erstes musste im Rahmen der Diplomarbeit in Zusammenarbeit mit Christian Heller die Spezifikation der Beschreibungssprache auch hinsichtlich der Webanwendungsfähigkeit fertig gestellt werden. Dabei mussten sowohl allgemeine als auch webspezifizische Sprachkonstrukte berücksichtigt werden. Es wurde festgestellt, dass der prinzipielle Aufbau von CYBOL die Belange von webspezifischen Anforderungen ohne Probleme erfüllt. Erweiterungen mussten nur in den Ausprägungen der Werte, wie z.B. die neue Operation *url_refresh*, die zur Zeit nur im Web Sinn macht, oder verschiedene Properties, die für die Generierung von XHTML-Seiten von Bedeutung sind, vorgenommen werden.

Im Folgenden möchte ich kurz auf die Elemente von CYBOL eingehen, die für den Prototyp der Diplomarbeit relevant sind. Dazu gehören welche Arten von *channel* und *abstraction* verwendet werden und welche Operationen definiert und programmiert werden mussten, damit diese Anwendung technisch umgesetzt werden konnte.

Für *channel* werden hier nur zwei Arten verwendet. Vorgesehen sind weitere Kommunikationskanäle. Diese sind aber noch nicht umgesetzt.

channel	**Beschreibung**
inline	Es muss kein Kommunikationskanal extra aufgemacht werden, sondern der Wert im *model* repräsentiert das Modell.
file	Das Modell muss aus einer Datei gelesen werden. Der Dateiname steht in *model*.

Tabelle 7.1: Überblick: channel in CYBOL

Die *abstraction* in CYBOL beinflußt, wie das *model* behandelt wird. Neben den Grunddatentypen, wie z.B. *integer*, *string* oder *double*, gibt es noch folgende Arten:

abstraction	**Beschreibung**
knowledge	In dem *model* steht ein Verweis auf das *knowlege memory*. Der Wert wird als String gelesen und als Zugriffspfad interpretiert. Das erzeugte Modell enthält den Wert, worauf *model* verweist.
encapsulated_knowledge	Hier wird noch ein Schritt weiter als bei der *abstraction knowledge* gegangen. Im *model* steht ein Zugriffspfad auf das Knowledge Memory, der den vom Programmierer gewollten Zugriffspfad vom Knowlegde Memory enthält. Mit diesen Konstrukt lassen sich dynamische Zugriffspfade erstellen.
cybol	Hier wird im *model* auf eine externe CYBOL-Datei verwiesen, die an dieser Stelle im erzeugten Modell geparst wird. Somit lassen sich z.B. hierarchische Strukturen im Knowledge Memory aufbauen.

abstraction	**Beschreibung**
operation	Der Wert im *model* wird als Operation verstanden. Eine Operation kann z.B. *create_part* zum Erstellen eines Knoten im Knowlege Memory oder *exit* zum Verlassen des Interpreters sein. Auf Grund der Operation wird in CYBOI diese Funktion aufgerufen.

Tabelle 7.2: Überblick: abstraction in CYBOL

Für die Realisierung des Prototypen in der Diplomarbeit mussten einige Operationen (siehe Tabelle 7.3) in CYBOI implementiert werden. Die Operation *url_refresh* ist die einzigste Operation, die speziell für die Webanwendung umgesetzt wurde. Die Operation *send* musste für die TCP-Socket-Kommunikation, die die Grundlage für die Webanwendung ist, modifiziert werden. Alle anderen Operationen sind allgemein gültig und können nicht nur in Webanwendungen sondern in jeder Beschreibung relevant sein.

operation	**properties**	**Beschreibung**
create_part	whole name channel abstraction model	Erzeugt ein Model, das durch die Eigenschaften *name*, *channel*, *abstraction* und *model* definiert ist. Die Eigenschaft *whole* gibt dabei an, an welcher Stelle das Modell im Knowledge Memory hinzugefügt wird. Ist die Eigenschaft nicht angegeben, so wird das Modell in der Wurzel vom Knowledge Memory hinzugefügt.
destroy_part	name	Zerstört ein Modell im Knowledge Memory, wobei der *name* die Stelle definiert.
send	language receiver message	Es wird eine Nachricht an den Empfänger geschickt. Über welchen Kommunikationsweg die Daten geschickt werden, entscheidet die *language*.

operation	**properties**	**Beschreibung**
count_part	basisname model result	Listen werden in Knowledge Memory über einen Basisnamen und einen Index abgespeichert. Diese Operation zählt alle Listeneinträge die unter dem *model* des Knowledge Memory mit dem *basisname* vorhanden sind. Das Ergebniss wird in *result* geschrieben.
build_listname	basisname index result	Mit dieser Operation kann ein Listbezeichner dynamisch zur Laufzeit generiert werden. Dabei wird der *basisname* und der *index* zum Listbezeichner in *result* geschrieben.
add	operand_1 operand_2 result	In Abhängigkeiten von den Datentypen der Operanden werden z.B. zwei Zahlen addiert bzw. bei String-Datentypen diese aneinander gehangen und in *result* gespeichert.
set	source destination	Der Inhalt von *source* wird nach *destination* kopiert.
url_refresh	url	Die *url* wird als Ergebnis an den Client geschickt. Dies hat die gleiche Auswirkung, als wenn ich im Webbrowser die URL eingebe, nur das hier vorher noch verschiedene andere Operationen, wie z.B. das Modifizieren des Domain-Wissens aus dem Knowledge Memory ausgeführt werden können.

operation	**properties**	**Beschreibung**
set_property	source destination destination_property	Mit dieser Operation kann eine Eigenschaft zu einen Modell dynamisch zur Laufzeit geändert werden. In *source* ist der Eigenschaftswert enthalten. Diese wird in die Eigenschaft *destination_property* des Modells *destination* aus dem Knowledge Memory geschrieben.
compare	operand_1 operand_2 operator result	Vergleicht die zwei Operanden und schreibt das Ergebnis in *result*. Welcher Vergleich durchgeführt wird, steht in *operator*. Dies könnte z.B. *equal* oder *equal_or_smaller* sein.
loop	break index model	Die Schleife wird solange ausgeführt, bis *break* auf *true* gesetzt wurde. In *model* wird definiert, was innerhalb der Schleife ausgeführt werden soll und der *index* wird bei jeden Schleifendurchlauf um eins erhöht. Der *index* muss darum vom Typ Integer sein.
startup	service tcp_socket_port	Startet einen Service. Für den integrierten Webserver in CYBOI muss für *service* „tcp_socket“ eingetragen sein. Nur für diesen Service ist die zweiten Eigenschaft *tcp_socket_port* von Bedeutung, da dies ein serviceabhängiger Startparameter ist. Dort wird der Port des Webservers übergeben, auf welchen er auf Anfragen wartet.
shutdown	service	Beendet einen Service in CYBOI.

operation	**properties**	**Beschreibung**
receive	service blocking	Hier wird ein Service in Empfangsmodus gesetzt. Die Eigenschaft *blocking* gibt dabei an, ob der Service als separater Thread oder direkt in CYBOI gestartet wird.
interupt	service	Beendet die Empfangsbereitschaft eines Services.

Tabelle 7.3: Überblick: operation in CYBOL

7.3 CYBOI und Webserver

Das Hauptproblem neben der Beschreibung war die Integration des Webservers in CYBOI. Dazu ist es notwendig sich die Architektur von Webanwendungen vor Augen zu halten. Der Webserver wartet auf eine Anfrage von einem beliebigen Client. Diese Anfrage muss er bearbeiten und das Ergebnis an den Client, der diese Anfrage gestellt hat, schicken.

Folgende Aufgabenstellungen waren dafür zu lösen.

- Welche Anfragen kann der Webserver verarbeiten?
  Der jetzige Stand ist eine Verarbeitung von Aufrufen einer CYBOL-Datei, d.h. in der URL des Webbrowser werden der Server, der Port und der Zugriffspfad der CYBOL-Datei eingegeben. In dem Beispiel könnte dies so aussehen:

  ```
  http://server:3456/examples/resadmin/logic/send_name.cybol
  ```

  Dabei ist zu beachten, das die letzte Anweisung in der CYBOL-Datei eine Send-Operation für TCP Socket ist. Wird keine Send-Operation für diese Anfrage gemacht, wartet der Client auf eine Antwort, die er halt nicht bekommen kann.

- Wie übertrage ich Werte vom Client zum Server?

  Oft ist es notwendig, Benutzereingaben an den Server zu schicken, damit der Server diese verarbeiten kann. Die normale Parameterübergabe an einen Webserver erfolgt bei der Get-Operation über die URL, getrennt durch ein `'?'`, und bei der Post-Operation werden die Daten am Ende der Anfrage angehangen. Die Parameter müssen dann vom Webserver bearbeitet werden. Dafür ist es erforderlich gewisse Notationen für diese Parameter festzulegen. Als erstes braucht man eine Variable, wo der Wert gespeichert werden soll. Dafür bietet sich der Knowledge Memory von CYBOI an. Als zweites ist ein Trennungszeichen zwischen Speicherort und Wert nötig. Dafür wurde das Zeichen `'='` gewählt. Somit ist ein Übergabeparamter in der URL folgendermaßen definiert.

  ```
  <Zugriffspfad vom Knowledge Memory>=<Wert der zugewiesen werden soll>
  ```

  Natürlich können auch mehrere Übergabeparamater übergeben werden. Diese werden durch das Zeichen `'&'` voneinander getrennt. Bei der Post-Operation werden die Übergabeparameter nicht durch die URL kodiert, sondern die Eingabefelder mit ihren Inhalt werden vom Webbrowser automatisch an die Anfrage hinzugefügt. Dabei ist aber wichtig, dass der Name von den Eingabefeldern dem Zugriffspfad vom Knowledge Memory entspricht, damit die Werte auch zugeordnet werden können.

- Wie ist der Webserver in CYBOI integriert?

  CYBOI arbeitet alle Signale in einer Endlosschleife nacheinander ab. Eine entgegengenomme Anfrage des Webservers wird zu dieser Signalwarteschlange hinzugefügt. Danach hat der Webserver erstmal nichts weiter zu tun, da CYBOI intern alle Signale abarbeitet. Erst wenn CYBOI die Send-Operation für TCP Socket ausführt wird dem Webserver mitgeteilt, das dieser die angeforderte Antwort an den Client schicken soll. Jetzt muss aber der Webserver wissen, an welchen Client er die Antwort zu schicken hat, da er mehrere Clients bearbeiten kann. Dazu muss er sich die Signal-Id, die von CYBOI für jede Anfrage vergeben wird und die dazugehörige Clientsocketnummer, die bei jeder Anfrage an den Webserver vergeben wird, merken. Wird die Send-Operation für eine Signal-Id ausgeführt, so wird die dazugehörige Clientsocketnummer ermittelt und an diese wird dann das Ergebnis geschickt.

- Wie wird eine XHTML-Antwort in CYBOI erzeugt?

  Die Grundstruktur der XHTML-Antwort wird im Knowledge Memory erzeugt. Erst wenn ich die Send-Operation auslöse wird die reale XHTML-Antwort generiert. Bei der Send-Operation wird ein Part aus dem Knowledge Memory als oberstes Hierarchie-Ebene angegeben. Alles was unter dem Part hängt wird ausgegeben. Dazu werden die Properties *html_tag* und *html_tag_properties* der Parts ausgewertet. Das Property *html_tag* gibt den Tag an in dem das Modell des Parts eingeschlossen wird. Zusätzliche Informationen zu dem Tag können mit dem Property *html_tag_properties* angegeben werden. Das folgende Beispiel verdeutlicht die Generierung. Die Beschreibung aus der folgenden CYBOL-Datei wird im Knowledge Memory erzeugt und mit der Send-Operation über TCP-Socket geschickt.

```
<part name="delete_button" channel="inline"
      abstraction="string" model="delete">
  <property name="html_tag" channel="inline"
            abstraction="string" model="a"/>
  <property name="html_tag_properties" channel="inline"
            abstraction="string" model="href='delete_address.cybol'"/>
</part>
```

  Es würde folgende XHTML-Antwort daraus generiert werden:

```
<a href='delete_address.cybol'>
  delete
</a>
```

Kapitel 8

Zusammenfassung

Ziel dieser Arbeit war es, CYBOP für Webanwendungen zu erweitern. Dafür musste untersucht werden, inwieweit die Beschreibungssprache CYBOL dafür geeignet ist, welche Änderungen an dieser nötig sind, um ein Webanwendungsfähigkeit zu erreichen und mit welcher Architektur diese im Interpreter integriert werden kann. Weiterhin war für den praktischen Teil der Interpreter CYBOI anzupassen und in CYBOL eine kleiner Prototyp einer Webanwendung zu realisieren.

Als Ergebnis dieser Untersuchung stellte sich heraus, dass CYBOL die Belange einer Webanwendung abdeckt. Es mussten keine Strukturänderungen an CYBOL vorgenommen werden. Die allgemeinen Strukturen für die Beschreibung von Sachverhalten (Domain, Logik, Repräsentation) sind auch für Webanwendungen ausreichend. Als Ergänzung für die XHTML-Darstellung mussten nur zwei verschiedene Properties definiert werden. Problematisch bei der Erstellung des Prototypen war das Nichtvorhandensein von unterstützenden Editoren. XML-Editoren konnten nur die XML-Syntax prüfen, aber nicht die spezifischen Anforderungen zur Entwicklung von CYBOP-Anwendungen unterstützen. Dazu wären spezielle Editoren (siehe Ausblick) notwendig. Als Vorteil für die Entwicklung ist die einfache Beschreibung von CYBOL zu nennen. Durch den hierarchischen Ansatz können schnell verständliche Programme erzeugt werden. Der große Nachteil zurzeit ist die Unstrukturiertheit der Anwendungsentwicklung. Es ist aus dem Quelltext (CYBOL-Dateien) auf den ersten Blick nicht erkennbar, welche Hierarchie die CYBOL-Dateien aufbauen. Dieser Nachteil würde aber durch unterstützende Editoren verschwinden.

Die Integration des erstellten Webservers in CYBOI erwies sich als das Hauptproblem. Wegen der eingeschränkten Kommunikationsmöglichkeiten zwischen Client und Server einer

Webanwendung (nur GET- und POST-Operationen) musste eine Schnittstelle zu CYBOI definiert werden. Dies geschah durch die Syntax-Definition der Parameterübergabe bei diesen Operationen.

Ein weiterer wichtiger Punkt war die Entscheidung für die Verwendung von blockierenden oder nicht blockierenden Socketkommunikationen. Durch die Verwendung von Threads konnten für die Kommunikation zwischen Client und Server blockierende TCP-Sockets verwendet werden. Dies bedeutet, dass keine Ressourcen des Computers verbraucht werden, solange der Webserver auf eine Anfrage des Clients wartet. Die Kombination von Thread und blockierenden Kommunikationskanälen sind auch bei anderen Services denkbar.

Der Vorteil bei der Integration des Webserver ist die Verwendung der Hauptstruktur von CYBOI. Der Webserver nimmt nur die Anfragen an, ermittelt die Übergabeparameter der Anfrage und reiht das Setzen der Parameter und die Anfrage des Clients in die Signalwarteschlange ein. Dann bearbeitet CYBOI diese Signale und am Ende wird der Webserver veranlasst, eine Antwort an den Client zu schicken. Es werden die Strukturen von CYBOI genutzt und nur die Socket-Kommunikation ist ausgelagert.

Kapitel 9

Ausblick

In der Programmierung versucht man Redundanzen zu vermeiden. Dies geschieht durch eine sinnvolle Zerkleinerung von Funktionen, so dass man diese an mehreren Stellen verwenden kann oder durch Vererbung, wie in der Objektorientierten Programmierung. Wie kann man diese Gedanken in CYBOP umsetzten? Die Zerkleinerung von Funktionen erfolgt in CYBOP durch die Aufteilung auf mehrere Dateien. Diese können an mehreren Stellen verwendet werden. Für den zweiten Teil ist anzumerken, das Vererbung in CYBOP nur Sinn macht, wenn von einem Part aus dem Knowledge Memory vererbt wird. Dieser Part ist aber nur eine hierarchische Struktur. Vererbung würde bedeuten, dass man diese Struktur unter einen anderen Part nochmal darunter hängen möchte. Dies kann aber auch über normale Operationen, wie *copy_part* oder *move_part* realisiert werden. Darum ist eine separate Vererbungsnotation in CYBOL nicht nötig.

Ein weiterer wichtiger Schritt für die Entwicklung von CYBOL ist die Abspeicherung der Daten. Dies ist ein sehr weitläufiges Thema, da die Daten in verschiedenen Formaten abgespeichert werden können. Vorstellbar wären in Datenbanken, XML-Dateien oder auch verschiedenen Binärformate. Der Fantasie sind hier keine Grenzen gesetzt. Ein interessanter Ansatz wäre auch die Daten, in dem CYBOL-Format zu speichern. Dabei ist zu beachten, dass CYBOL ein XML-Format ist und eventuelle nicht XML konforme Datenelemente entsprechend den XML-Regeln maskiert werden müssen.

Für die Modellierung von Anwendungen gibt es verschiedene Ansätze. Am nahe liegensten, da CYBOL im XML-Format vorliegt, wäre die Verwendung eines XML-Editors. Dies würde aber nur die Syntax von CYBOL kontrollieren. Weitere Punkte, die ein Entwicklungswerkzeug für CYBOL besitzen sollte, wäre die hierarchische Modellierung der Templates sowie

die hierarchische Darstellung des Modells (Knowledge Memory). Das erste wäre zur Modellierung von CYBOL-Dateien gedacht, um die Kompositbeziehungen der Templates zu modellieren. Der zweite Teil ist zur Veranschaulichung des aktuellen Laufzeitmodells geeignet.

In CYBOI gibt es noch keine benutzerbezogene Variablen. Für die Umsetzung, auch in Hinblick von Webanwendungen, gibt es verschiedene Lösungsmöglichkeiten. Eine davon wäre die Speicherung im Knowledge Memory unter einem benutzerdefinierten Zweig. Dies wäre z.B. der Login-Name von der Betriebssystemanmeldung. Bei Webanwendungen ist dies etwas schwieriger. Normalerweise kann der Web-Client nicht auf die Betriebssystemebene zugreifen. Der Client bekommt nicht die Information des angemeldeten Benutzers. In Webanwendungen gibt es dafür sessiongebunden Variablen. Eine Session bedeutet eine zeitweise bestehende Verbindung zwischen Server und Client. Diese Sessionvariablen dienen dazu, Informationen für eine Session zu merken. Diesw sind temporäre Variablen, die nach Beenden der Session nicht mehr existieren. In Standardwebanwendungen können die Sessionvariablen auch länger über Cookies gespeichert werden. Für benutzerbezogene Daten müsste eine Sessionverwaltung in CYBOI und dessen Webserver integriert werden. Eine weitere Möglichkeit für Webanwendungen wäre die Umsetzung eines SSO (Single Sign On), wo die Anmeldung des Betriebssystems an die Webanwendung durchgereicht wird.

Ein Problem ist zurzeit noch die Robustheit des Systems. Beendet sich CYBOI aus irgendeinen Grund, sei es durch Hardwarefehler, durch Softwarefehler oder einfach durch Stromausfall, so sind die Daten seit der letzten Speicherung, da sie sich nur im Speicher befinden, einfach weg. Das System ist ist so zu erweitert, dass im Prinzip die Daten fortdauernd in einem bestimmten Format auf dauerhafte Speichermedien geschrieben werden. Bei einem Absturz können nach dem Neustart des Systems die Daten aus dem temporären Dauerspeicher wieder rekonstruiert werden.

Anhang A

Literaturverzeichnis

[CA77] Christopher Alexander, Sara Ishikawa, Murray Silverstein: *A Pattern Language*. Oxford University, 1977.

[deb05] *Debian*. www.debian.de, Juni 2005.

[def05] *Definition Socket*. http://whatis.techtarget.com/definition/0,,sid9_gci213021,00.html, Juni 2005.

[ebn05] *EBNF*. http://de.wikipedia.org/wiki/Erweiterte_Backus-Naur-Form, Juni 2005.

[EG04] Erich Gamma, Richard Helm, ...: *Entwurfsmuster*. Addision-Wesley, 2004.

[FB98] F. Buschmann, R. Meunier, ...: *Patternorientierte Software-Architektur*. Addision-Wesley, 1998.

[gnu05] *GNU*. www.gnu.org, Juni 2005.

[Hel04] Heller, Christian: *Cybernetics Oriented Language (CYBOL)*. http://cybop.berlios.de/papers/2004_cybernetics_oriented_language/paper.pdf, 2004.

[H.R97] H.R.Hansen: *Wirtschaftsinformatik I*. Gustav Fischer Verlag - Stuttgart, 7 Auflage, 1997.

[Kap03] Kappel, Pröll, ...: *Web Engineering*. Dpunkt Verlag, 2003.

[MD90] Maria Dose, Jürgen Folz, ...: *Fremwörterbuch*. In: *Duden*, Band 5. Dudenverlag Mannheim - Leipzig - Wien - Zürich, 1990.

[Ull02] Ullenboom, Christian: *Java ist auch eine Insel*. Galileo Computing, 2 Auflage, 2002.

[w3c05a] *Spezifikation XML*. http://www.w3.org/TR/2000/REC-xml-20001006, Juni 2005.

[w3c05b] *Spezifikation XML (deutsch)*. http://edition-w3c.de/TR/2000/REC-xml-20001006/, Juni 2005.

[Wol04] Wolf, Jürgen: *Linux-Unix-Programmierung*. Galileo Press, 2004.

[xsd05] *Def. XML-Schema*. http://www.w3.org/TR/xmlschema-1/, Juni 2005.

Anhang B

Thesen

These 1:
CYBOL entspricht dem XML - Standard, da dieser die hierarchische Beschreibungssprache abdeckt.

These 2:
Mit CYBOL sind die Grundoperationen, Sequential, Selection, Iteration, als Grundbestandteil einer Beschreibungssprache realisierbar.

These 3:
Die Integration eines Webservers in CYBOI und somit die Realisierbarkeit von Webanwendungen für CYBOP ist möglich.

These 4:
Die Darstellung von Webanwendungen unter CYPOP ist mit XHTML am sinnvollsten. Damit wird auch das hierachische System von Human Thinking konsequent weitergeführt.

These 5:
Durch die Verwendung von Threads und blockierenden Sockets ist die Belastung der Hardware und der damit verbundene Ressourcenverbrauch sehr gering, wodurch die Performance von CYBOI wesentlich erhöht wird.

These 6:

Die Grundstruktur von Threads und blockierenden Kommunikationskanal, wie sie bei dem Webserver eingesetzt wird, ist analog auf andere Services übertragbar.

These 7:

Erweiterungen von Operationen sind ohne Anpassungen in CYBOL möglich. Diese sind nur in CYBOI zu implementieren.

These 8:

Durch Verwendung von speziellen Editoren, die die spezifischen Besonderheiten von CYBOP berücksichtigen (Knowledge Memory, Hierarchiedarstellung der Templates), ist die Programmentwicklung effizienter zu gestalten.

Ilmenau, den 23. Juni 2005 _____

Rolf Holzmüller

Anhang C

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Diplomarbeit selbständig und nur unter Verwendung der angegebenen Hilfsmittel geschrieben zu haben.

Ilmenau, den 23. Juni 2005

Rolf Holzmüller