Studienarbeit

Ein XML-basiertes Konfigurationswerkzeug für Betriebssysteme am Beispiel EPOS

Sascha Römke
Otto-von-Guericke-Universität Magdeburg
Fakultät für Informatik

Aufgabenstellung: Prof. Dr. Wolfgang Schröder-Preikschat
Betreuung: Antônio Augusto Fröhlich
Dipl. Inf. Danilo Beuche

Magdeburg, 16. Februar 2001

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Motivation 1
1.2 Problemdefinition 2
1.3 Gliederung 3

2 Konfigurierung **4**
2.1 Allgemeines 4
2.1.1 Konfiguierung vs. Konfiguration 4
2.1.2 Konfigurationsregeln 5
2.1.3 Konfigurierungskomplexität 5
2.1.4 Statische und dynamische Adaption 6
2.1.5 Effizienz 7
2.2 Statische Adaption 7
2.3 Dynamische Adaption 9

3 EPOS und dessen Konfigurierung **11**
3.1 Was ist EPOS ? 11
3.2 Das Design von EPOS 12
3.2.1 Systemabstraktionen 12
3.2.2 Szenarioadaptoren 13
3.2.3 EPOS-Schnittstellen 13
3.2.4 Exklusive und partielle Realisierungsbeziehungen . . . 14
3.3 Die Umsetzung von EPOS 15
3.4 Einordnung des neuen Werkzeugs 16

4 Die eXtensible Markup Language **18**
4.1 Warum XML ? 18
4.1.1 Die Vision 18
4.1.2 Die Entwicklung 19
4.2 Die Sprache 20
4.2.1 Das Sprachkonzept 20
4.2.2 Document Type Definition 20
4.2.3 XML-Dokumente 22

4.2.4 Document Object Model 23

5 Entwurf **24**

5.1 Anforderungsanalyse 24

5.1.1 Funktionale Anforderungen 24

5.1.2 Qualitätsanforderungen 24

5.1.3 Anforderungen zur Systemrealisierung 25

5.2 Systeminformationen 25

5.3 Die Schnittstellen 26

5.3.1 Die Eingabesprache 26

5.3.2 Die Ausgaben 31

5.3.3 Weitere Schnittstellen 32

5.4 Die graphische Oberfläche 33

5.4.1 Die Oberflächenobjekte zur Darstellung der Komponenten und Variablen 33

5.4.2 Das Oberflächenmodell 34

5.5 Das interne Verhalten - die Konfigurationsregeln 36

5.5.1 Die Wissensbasis 36

5.5.2 Testen der Vorbedingung 36

5.5.3 Testen der Nachbedingung 36

5.6 Die Programmiersprache 37

6 Implementation **39**

6.1 Systemvoraussetzungen 39

6.1.1 Neue visuelle Elemente 40

6.1.2 Der Parser 40

6.1.3 Starten und Übersetzen von XConf 40

6.2 Der Ablauf 41

6.3 XConf für andere Betriebssysteme nutzen 42

6.3.1 Neue externe Werkzeuge 42

6.3.2 Neue Systeminformationen 42

7 Zusammenfassung **44**

7.1 Diskussion 44

7.2 Aussichten 45

Literaturverzeichnis **47**

Anhang **48**

A Die Konfigurierungsbeschreibung **48**
A.1 Die DTD-Grammatik 48
A.2 Ausschnitt der Epos-Konfigurierungsbeschreibung in Xml 50

B Screenshots **51**

C Die Ausgaben **52**
C.1 Eine Konfigurationsbeschreibung für Epos 52
C.2 Ausschnitt der Epos-Konfigurationsinformationen 53
C.3 Die alternativen Gruppen 54

Kapitel 1

Einleitung

1.1 Motivation

Ein Betriebssystem ist eine Software, welche die Ressourcen eines Computers[1] verwaltet, und die Ausführung von Benutzerprogrammen steuert. Es vermittelt somit zwischen der Hardware (Computerkomponenten) und den Anwendungen, die auf dem Computer ablaufen.
Die Konfigurierung eines Betriebssystems besteht in der Anpassung des Systems an die Hardware und in der Auswahl der gewünschten Funktionalität. Eine komponentenbasierte Konfigurierung setzt sich im wesentlichen aus drei Schritten zusammen. Als erstes erfolgt die Selektion der Komponenten[2], welche die gewünschte Funktionalität erbringen. Danach werden diese Komponenten über ihre Parameter konfiguriert. Im letzten Schritt erfolgt die Systemgenerierung, indem die Komponenten zum System zusammengesetzt werden.

Die Konfigurierbarkeit, als eine wichtige Eigenschaft von Betriebsystemen, ist durch die Vielgestaltigkeit der Hardware, die Spezialisierung bezüglich konkreter Aufgaben und dem Wunsch nach Effizienz begründet. Ein System kann sich durch seine Anpaßbarkeit an die Hardware auszeichnen. Die sogenannten *special purpose systems* sind Systeme, welche speziell für eine Aufgabe gefertigt wurden. Da ein Betriebssystem die Ressourcen des Rechners verwaltet, sollte es so wenig wie möglich selber davon beanspruchen, also effizient bezüglich der Ressourcen sein.
Es gibt Hardware, die durch ihr Umfeld, etwa in der physikalischen Größe oder im Energieverbrauch eingeschränkt ist. Damit sind auch die Ressourcen einer solchen Hardware begrenzt. So kann zum Beispiel der Arbeitsspeicher auf nur wenige Bytes begrenzt sein. Software für diese Hardware muß sich an diese Beschränkungen anpassen, also an die Hardware angepaßt sein.

[1] z.B.: Zeit, CPU, Speicher, Energie
[2] Was eine Komponente ist, wird in *Kapitel 2* erläutert.

Durch die Auswahl verschiedener Komponenten, bzw. deren Einstellungen können somit ganz unterschiedliche Betriebssysteme aus demselben Quellcode entstehen. Je mehr Komponenten auswählbar sind, und je mehr Einstellungsmöglichkeiten sie in sich bieten, desto komplexer wird die Konfigurierung. Diese wird noch komplexer durch Abhängigkeiten, die zwischen den Komponenten des Systems bestehen. Es ist zum Beispiel möglich, daß die Funktionalität eines Systems durch die Auswahl von Komponenten verändert wird. Dabei ist es schwer nachvollziehbar, welche Auswirkungen das Aktivieren bzw. das Deaktivieren bestimmter Implementierungsteile hat, so daß nicht immer klar ersichtlich ist, welche Konsequenzen dies auf andere Teile des Systems hat.
So kann die Zusammenstellung einer gültigen Konfiguration für ein System, so komplex werden, das nicht der Anwender das System konfigurieren kann, sondern typische Konfigurationen, aus denen der Anwender dann auswählen kann, durch den Systementwickler bereitgestellt werden. Dies bedeutet aber, daß keine dieser Konfigurationen nur den von dem Anwender benötigten Funktionsumfang beinhaltet, sondern auch nicht verwendete Teile, und somit nicht optimal ist.
Eine bessere Lösung ist die Unterstützung der Konfigurierung durch Werkzeuge. Diese sollten dem Nutzer die auswählbaren Komponenten und deren Parameter übersichtlich darbieten. Desweiteren können sie die Einhaltung der Abhängigkeiten zwischen den Komponenten automatisch überprüfen. Damit verringern sie die Komplexität der Konfigurierung erheblich, beziehungsweise ermöglichen die Konfigurierung größerer Systeme erst.

1.2 Problemdefinition

Im Rahmen dieser Studienarbeit erfolgte die Beschäftigung im Projekt EPOS, dessen Ziel die automatische Generierung von anwendungsorientierten Betriebssystemen ist. EPOS ist eine Erweiterung des PURE-Betriebssystems, und definiert ein Framework, welches die Konstruktion eines Betriebssystems, in Form des Zusammensetzens von Systemabstraktionen, unterstützt. Zur Unterstützung der automatischen Generierung von Betriebssystemen, existieren Werkzeuge, beispielsweise zur Analyse der Anwendung, das einen Systembauplan liefert, der zur Übersetzung des anwendungsorientierten Betriebssystems benutzt wird.
Die Aufgabe bestand in der Implementierung einer graphische Schnittstelle zu bereits bestehenden Konfigurationswerkzeugen für das Betriebssystem EPOS. Diese Schnittstelle sollte es dem Nutzer ermöglichen, Konfigurationen zu überprüfen und zu verändern. Da es eine graphische Schnittstelle ist, macht sie die Konfigurierung nutzerfreundlicher, insbesondere auch für "normale" Nutzer.

1.3 Gliederung

In dieser Arbeit wird ein graphisches Werkzeug zur Vereinfachung der Konfigurierung von Betriebssystemen entworfen. Die Entwicklung erfolgte am Betriebssystem EPOS.

Der weiter Inhalt der Arbeit gliedert sich wie folgt:

Kapitel 2 beschäftigt sich mit dem Thema der Konfigurierung. Dabei wird speziell auf die statische und dynamische Konfigurierung von Betriebssystemen eingegangen.

Kapitel 3 In diesem Kapitel erfolgt eine kurze Beschreibung des Betriebssystems EPOS, und dessen Konfigurierung. Dabei wird das zu implementierende Werkzeug in diesen Prozeß eingeordnet.

Kapitel 4 Dieses Kapitel gibt eine kurze Einführung in die *eXtensible Markup Language*, da diese Sprache noch relativ jung ist. Sie wird zur Beschreibung der Eingabedaten für das Werkzeug verwendet.

Kapitel 5 In diesem Kapitel wird der Entwurf des graphischen Werzeugs erläutert. Dabei wird besonders auf die Ein- und Ausgaben und die graphische Umsetzung eingegangen.

Kapitel 6 beschreibt die Nutzungsbedingungen für das Werkzeug. Weiterhin wird beschrieben, was zu beachten ist, falls es für die Konfigurierung anderer Betriebssysteme, außer EPOS, verwendet werden soll.

Kapitel 7 faßt die gewonnenen Erkenntnisse zusammen und gibt einen Ausblick auf enventuelle zukünftige Weiterentwicklungen bzw. Erweiterungen des implementierten Werkzeugs.

Mein Dank gilt allen, die mich bei der Erstellung dieser Arbeit unterstützt haben, Antônio Augusto Fröhlich für die Betreuung während des Praktikums an der GMD FIRST *und Danilo Beuche für die Betreuung während des Schreibens dieser Studienarbeit. Besonderer Dank geht an Dagmar Schönewolf, die als "Lektorin" den Tippteufel vertrieb und durch zahlreiche Anmerkungen geholfen hat, diese Arbeit besser lesbar und verständlicher zu gestalten.*

Kapitel 2

Konfigurierung

2.1 Allgemeines

Da ein Betriebssystem im Sinne der Konfigurierung einem normalen Softwaresystem entspricht, wird es in diesem Kapitel neutral als System bezeichnet.

2.1.1 Konfiguierung vs. Konfiguration

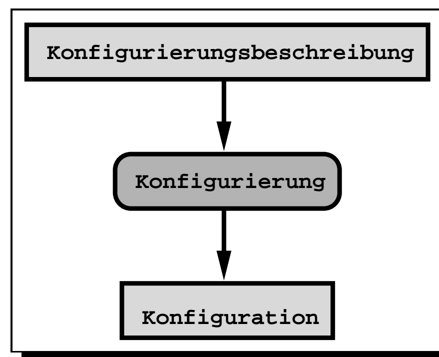


Bild 2.1: Die Konfigurierung

Die *Konfigurierung* eines Systems ist der Prozeß der Anpassung (*Adaption*) dieses Systems an die tatsächlich existierenden Anforderungen, zum Beispiel an die Hardwareplattform. Da genaue Kenntnisse über diese Anforderungen meist erst dem Nutzer, und nicht schon dem Entwickler, bekannt sind, sollte eine Konfigurierung so spät wie möglich erfolgen, um eine optimale Anpassung zu ermöglichen.

Als Eingabe dient hierbei die *Konfigurierungsbeschreibung*. Diese besteht aus einer Menge von Informationen über auswählbare Systemkomponenten, deren Eigenschaften und Abhängigkeiten sowie deren Quellcode. Dazu können auch die Anforderungen des Nutzers an das System gehören.

Die Ausgabe des Konfigurierungsprozesses ist die *Konfiguration* in Form

- eines fertigen, ausführbaren Systems
- einer *Konfigurationbeschreibung*, die zur
  - Übersetzungszeit (statische Adaption, siehe 2.2)
  - Ladezeit
  - Laufzeit (dynamische Adaption, siehe 2.3)

  des Systems ausgewertet wird.

2.1.2 Konfigurationsregeln

Die *Konfigurationsregeln* enthalten Aussagen über die Relationen zwischen den Komponenten. Sie sind Bestandteil der Konfigurierungsbeschreibung. Im wesentlichen können vier Formen (nach [CE]) der Abhängigkeiten zwischen Komponenten unterschieden werden:

- *mandatory*
  Bei dieser Form der Abhängigkeit impliziert die Auswahl einer Komponente die Auswahl einer oder mehrerer weiterer Komponenten.
- *alternativ*
  Stehen sich Komponenten alternativ gegenüber, darf und muß aus dieser Menge von Komponenten genau eine ausgewählt werden.
- *optional*
  Eine Komponenten, die dem System optional zur Verfügung steht, kann entweder dem System hinzugefügt werden oder nicht.
- *optionalalternativ*
  Hierbei muß aus einer Menge von Komponenten mindestens eine, es dürfen aber auch mehrere, ausgewählt werden.

Erfüllt eine Konfiguration alle vorgegebenen Abhängigkeiten zwischen den Komponenten, wird diese als *gültige Konfiguration* bezeichnet.

2.1.3 Konfigurierungskomplexität

Die Granularität eines Systems ist in erster Linie abhängig von dessen erbrachten Funktionalität, genauer gesagt der Varianz dieser Funktionalität. Je variabler die Funktionalität ist, desto feingranularer ist das System. Da die Funktionalität durch Komponenten erbracht wird, sind die Variationsmöglichkeiten der Komponenten genau so zu betrachten, wie die Varianz innerhalb jeder einzelnen Komponente.

Es existiert leider noch keine einheitliche Definition einer Komponente. Eine Definition ist zum Beispiel in [Eis97] zu finden. Angelehnt an diese Definition ist eine *Komponente* in dieser Arbeit eine funktional in sich geschlossene Einheit, die getrennt von ihrer Umgebung und anderen Komponeneten betrachtet wird. Das bedeutet, daß die Implementierung intern gehalten wird, und die Kommunikation über deren Schnittstellen erfolgt. Eine Komponente kann zum Beispiel nur eine einzige Klasse oder ein komplettes Modul, welches die Funktionalität mehrerer Klassen in sich vereint und nach außen anbietet, sein.

Die Komplexität der Konfigurierung eines Systems wird durch dessen Granularität bestimmt, da mit der Variabilität der Funktionalität auch deren Kombinationsvielfalt steigt. Diese wird zwar noch durch die Konfigurationsregeln eingeschränkt, dennoch ist eine manuelle Konfigurierung durch den Nutzer, also das Aussuchen und Anpassen der Komponenten, somit bei komplexeren Systemen fast unmöglich. Weiterhin erfolgt bei der manuellen Konfigurierung keine Unterstützung bei der Kontrolle der Konfigurationsregeln. Doch genau dies läßt sich durch die Nutzung von Werkzeugen erleichtern, welche die Einhaltung der Regeln überprüfen. Diese Form der Adaption wird als werkzeugunterstützte Konfigurierung bezeichnet. Durch weitere Werzeuge können auch Teile des Systems automatisch konfiguriert werden, womit die Konfigurierung komplexerer System einfacher bzw. erst ermöglicht wird.
Aus diesem Grunde sollte alles, was automatisch konfigurierbar ist, auch automatisch konfiguriert werden.

2.1.4 Statische und dynamische Adaption

Die Vorraussetzung für eine Adaption ist eine offene und verständliche Struktur des Systems. Diese erlaubt die Identifizierung der zu adaptierenden Systemkomponenten und ihr Zusammenwirken mit dem restlichen System. Dabei muß man nicht das gesamte System detailliert kennen, es genügt eine geeignete Gliederung in immer weiter abstrahierende Schichten. Diese ermöglicht die Konzentration auf die Dienste oder Betriebsmittel.

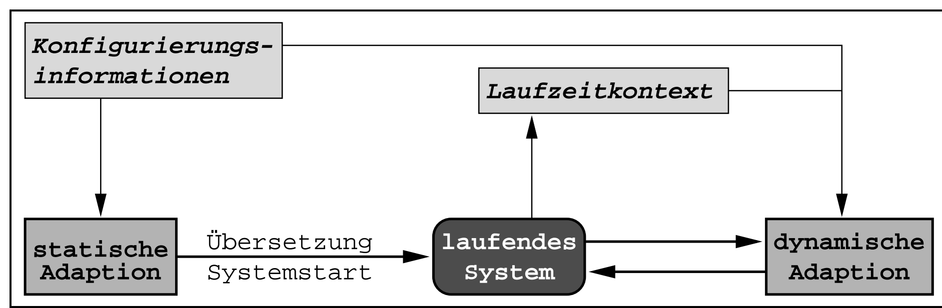


Bild 2.2: Zusammenhang - statische und dynamische Adaption

Sind alle adaptierbaren Komponenten ermittelt, muß zuerst eine statische Adaption des Systems durchgeführt werden, welche auch als initiale Konfiguration des System bezeichnet werden kann. Dabei werden alle bis dahin bekannten Anforderungen an das System berücksichtigt. Die statische Konfigurierbarkeit eines Systems ist Voraussetzung für die dynamische Adaption. Diese erfolgt zur Laufzeit des Systems, welches schon statisch konfiguriert ist, womit man sie auch als Rekonfigurierung bezeichnen kann. Diese Adaptionsform berücksichtigt zusätzlich zu den Systeminformationen den aktuellen Laufzeitkontext des Systems. Dieser Zusammenhang zwischen statischer und dynamischer Adaption wurde im Bild 2.2 dargestellt.
Der Unterschied zwischen statischer und dynamischer Adaption besteht im Zeitpunkt der Konfigurierung. Die adaptierbaren Komponenten sind dabei durch die Systemstruktur vorgegeben. Beispielsweise kann ein Konfigurationspunkt bei der statischen Adaption des Systems sein, ob es skalierbar werden soll oder nicht, also ob es zur Laufzeit rekonfigurierbar sein soll.

2.1.5 Effizienz

Einer der wichtigsten Aspekte einer Konfiguration ist dessen Effizienz. Denn neben der Anpassung an gewisse Hardware, bzw. die "beliebige" Auswahl von Komponenten, ist es das Ziel, ein optimales System zu bauen, um die vorhandenen Ressourcen so optimal wie möglich auszunutzen. Unbenutzte Funktionalität erzeugt in diesem Sinne einen Mehraufwand (*overhead*), welcher unerwünscht ist, da er Ressources verschwendet, also unnötig Speicher belegt, und im schlechtesten Fall sogar zur Verschlechterung der Laufzeiteffizenz führen kann.

2.2 Statische Adaption

Die Konfigurierung eines Systems vor dessen Start, wird als *statische Konfigurierung* oder *statische Adaption* bezeichnet, da der dynamische Aspekt des Systems zur Laufzeit, also die Menge der laufenden Prozesse und deren Zustände, noch nicht berücksichtigt werden kann. Damit beschränkt sich der Adaptionsvorgang auf die Systembeschreibung, die Konfigurierungsinformationen und den Quelltext.

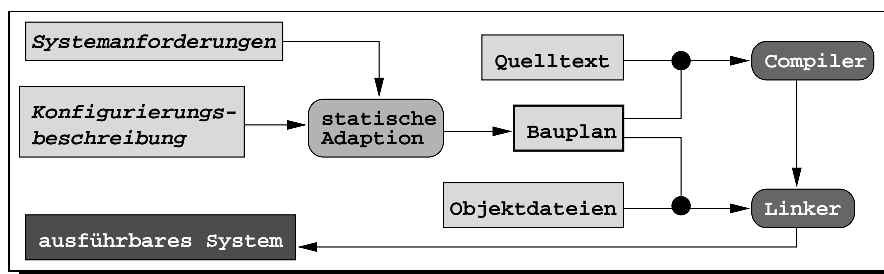


Bild 2.3: Der Weg zum ausführbaren System

Wie im Bild 2.3 beschrieben, werden durch die statische Adaption die Anforderungen an das System, unter Berücksichtigung der Konfigurationsregeln, in eine Konfiguration in Form eines *Bauplans* transformiert. Dieser Bauplan enthält die spezifischen Systeminformationen, zum Beispiel in Form von Präprozessoranweisungen oder eines Makefiles, die dann vor oder während der Übersetzung des Systems durch ein Werkzeug, zum Beispiel den Präprozessor oder `make`, ausgewertet werden müssen.

Wird der Bauplan in Form eines Makefiles erzeugt, kann dieses Informationen für den Linker enthalten, welcher (schon übersetzte) Objektdateien zu einem System zusammenbindet.

Der C-Präprozessor wird zum Beispiel bei in C++ programmierten Systemen angewendet. Er ist kein Bestandteil der Sprache, sondern verrichtet seine Arbeit vor dem eigentlichen Compiler. Dabei nimmt er eine Reihe von lexikalischen Veränderungen im Quelltext vor. Er kann sogenannte Macros zur Textersetzung (`#define`) definieren und liest zusätzlich benötigte Quelldateien ein (`#include`). Über Direktiven zur bedingten Übersetzung können, abhängig von Werten oder dem Vorhandensein bestimmter Symbole (`#ifdef`), Teile des Quelltextes übersetzt oder ignoriert werden.
Die Anwendung eines Präprozessors ist einerseits einfach, aber andererseits auch sehr fehleranfällig und verschlechtert die Wartbarkeit des Systems. Einer der Gründe hierfür ist, daß er kein Bestandteil des Compilers ist, und somit keine Typüberprüfung durchführen kann.

Eine Alternative bietet die Konfigurierung mittels parametrisierter Typen, wie zum Beispiel Templates in C++. Templates sind Klassen- oder Funktionsschablonen. Erst beim Gebrauch einer solchen generischen Funktion oder Klasse kann der Compiler anhand der verwendeten Typen erkennen, welche bestimmte Instanz der Klassen- oder Funktionsschablone benötigt wird. Ist diese zu dem Zeitpunkt noch nicht vorhanden, wird der erforderliche Code vom Compiler generiert. Also muß für jeden neuen Typ ein neuer Code generiert werden.

Die Anforderungen an das entstehende System sollten zur Übersetzungs- bzw. Linkzeit so weit wie möglich bekannt sein, damit nur die geforderte Funktionalität in das System integriert wird. Je mehr Eigenschaften des zukünftigen Systems bekannt sind, desto besser wird es an diese angepaßt sein. Dann kann ein System entstehen, für welches, bezogen auf die Funktionalität, der benötigte Speicherplatz so gering und die Laufzeiteffizienz so hoch wie möglich sind.

Bei einigen Systemen ist dieser Aspekt der optimalen Nutzung der vorhandenen Ressourcen so wichtig, daß sie ausschließlich statisch konfigurierbar sind, da der Aufwand für die Konfigurierung zur Laufzeit nicht tragbar wäre. Solche Systeme besitzen in der Regel eine:

- Zeitgarantie (Systeme mit harten Echtzeitanforderungen)
  Der wichtigste Aspekt der Echtzeitsysteme ist die Zeitgarantie. Konstruiert man solche Systeme dynamisch rekonfigurierbar, sind der Konfigurationszeitpunkt und der dann aktuelle Systemstatus nicht bekannt. Damit ist die Komplexität der Adaption, in dem Fall deren Zeitverbrauch, nicht konkret bestimmbar, was einer Abschwächung der Zeitgarantie entspricht. Bei "normalen" Echtzeitsystemen ist das noch tragbar, doch bei Systemen mit einer harten Echtzeitanforderung nicht. Beispiele für harte Echtzeitanforderungen sind in Sicherheitssystemen, wie z.B. ABS-Systemen, zu finden. Bei diesen Systemen ist die Reaktionszeitzeit des Systems so gering wie möglich zu halten, was ein schnelles und deterministisches System voraussetzt.

- begrenzte Speicherkapazität (Eingebettete Systemen)
  Bei diesen Systemen existiert meist eine obere Grenze für den Speicherbedarf, welche durch die Hardware vorgegeben ist. Die Hardware ist oft durch ihre Umgebung begrenzt. So zum Beispiel in den räumlichen Abmessungen, oder auch im Energieverbrauch.

2.3 Dynamische Adaption

Wie schon angedeutet, kann die dynamische Adaption eines Systems als Erweiterung der statischen Adaption gesehen werden. Sie wird dynamisch genannt, da sie die dynamischen Aspekte des Systems zur Laufzeit berücksichtigt. Sie beinhaltet im wesentlichen die folgenden zwei Aktionen.

- Das *Hinzufügen von Funktionalität* ist die meist verwendete Form der dynamischen Adaption. Dabei werden Module bei einer Dienstanforderung in den Speicher nachgeladen. Das Problem der meisten Module ist, daß sie nur als Ganzes geladen werden können, obwohl oft nur ein Teil ihrer Funktionalität benötigt wird. Dadurch wird unnötig Speicher belegt, im schlechtesten Fall kann dies sogar zur Verschlechterung der Laufzeiteffizenz führen.

- Durch das *Entfernen von Funktionalität* wird das Laufzeitsystem von nicht mehr benötigter Funktionalität befreit.

Die dynamische Adaption findet Anwendung in den Gebieten, in denen das System auf Veränderungen der Anforderungen zur Laufzeit reagieren können muß, und sich somit der aktuellen Situation anpaßt. Verschieden Beispiele sind:

- komplexe Betriebssysteme
  Bei vielen konventionellen Betriebssystemen, wie zum Beispiel Microsoft Windows oder Unix, wäre eine statische Konfigurierbarkeit allein nicht durchsetzbar, da sonst beispielsweise bei jeder neu hinzugefügten Hardwarekomponente das System neu übersetzt werden müßte, um den Gerätetreiber dieser Komponente einzubinden. Durch die dynamische Rekonfigurierung wird die Systemfunktionalität erweitert, indem der Gerätetreiber nachgeladen wird.

- Systeme, deren Dienste einer hohen Verfügbarkeit bedingen
  Würde ein solches System nur statisch konfigurierbar sein, so würde eine Neukonfigurierung Dienstausfall bedingen, der beispielsweise bei Servern nicht vertretbar wäre. Der weitaus schwerwiegendere Aspekt wäre die Wiederherstellung des, vor der Rekonfigurierung bestehenden Systemzustandes, sprich die Menge der laufenden Prozesse und deren Zustände.

- die Entwicklung von Betriebssystemen
  Diese wird als inkrementeller Vorgang betrachtet, in dem sich Entwurf, Implementation und Test zyklisch wiederholen. Bei bestimmten Betriebssystemen ergeben sich daraus erhebliche *Turn-Around-Zeiten*. Diese können durch die dynamische Adaption teilweise reduziert werden, indem die zu testende Funktionalität in das Laufzeitsystem nachgeladen wird, anstatt das gesamte System mit der neuen Funktionalität neu zu übersetzen.

- Kleinstsysteme
  Vorstellbar sind hier eingebettete Systeme, welche einen extrem geringen Arbeitsspeicher zur Verfügung haben, so daß die benötigte Funktionalität jeweils zum Zeitpunkt des Bedarfs nachgeladen werden muß. Dabei muß meist eine zu dem Zeitpunkt nicht benötigte Funktionalität aus dem Speicher entfernt werden, um den nötigen Platz frei zu geben.

Eine besondere Form der dynamisch konfigurierbaren Systeme sind die *reflektiven Systeme*. Diese müssen nicht nur dynamisch konfigurierbar, und damit auch statisch konfigurierbar sein, sondern auch gewisse Systemparameter überwachen. Diese Überwachung der Parameter und die Berechnungen auf diesen Parametern werden als *Kosten der Reflektion* bezeichnet, da diese Laufzeit benötigen und keine Funktionalität erbringen. Wird der Grenzwert eines Parameters unter- bzw. überschritten kann es zu einer Rekonfigurierung des Systems kommen. Weitere Informationen sind in [GJ90] zu finden.

Kapitel 3

EPOS und dessen Konfigurierung

Das im Rahmen dieser Studienarbeit implementierte Werkzeug wurde für die Konfigurierung des Betriebssystems EPOS getestet. Daher wird in diesem Kapitel eine kurze Beschreibung dieses Systems beziehungsweise der Art und Weise dessen Konfigurierung gegeben. Dabei wird die Einordnung des zu implementierenden Werkzeugs in diesen Konfigurierungsprozeß deutlich.

3.1 Was ist EPOS ?

EPOS ist das Akronym für *Embedded Parallel Operating System*. Eingebettete parallele Anwendungen arbeiten meist unter extremen Ressourceneinschränkungen. Zusammen mit der Erkenntnis, daß jede (parallele) Anwendung differierende Anforderungen an ihr Laufzeitsystem hat, ergibt sich der Anspruch einer jeden Anwendung auf ein für sie spezialisiertes Laufzeitsystem.
Der Entwurf eines Betriebssystems, welches auf eine gegebene Anwendung zugeschnitten werden kann, beginnt mit der Erzeugung der "Betriebssystembausteine", aus welchen das System später zusammengesetzt werden soll. In der Objektorientierung werden diese wiederverwendbaren Bausteine durch Klassen implementiert und in sogenannten *Repositories* (z.B. Bibliothek) gespeichert.
Daraus ergibt sich das Problem des Zusammensetzens dieser Bausteine, da eine Differenz zwischen dem besteht, was dieses Repository bietet, also einer Menge von anpaßbaren Klassen, und dem, was für einen Anwendungsprogrammierer wichtig ist, also einer abstrakten Beschreibung seiner Anforderungen an das System. Je umfangreicher das System wird, also je mehr Komponenten im Repository sind, und je komplexer diese sind, desto größer wird auch diese Differenz. Damit wird es für den Anwendungsprogrammierer immer schwieriger, dieses Repository zu durchsuchen, und diejenigen

Klassen auszuwählen und anzupassen, welche die Anforderungen seiner Anwendung an ein Betriebssystem erfüllen, und nach Möglichkeit auch nicht viel mehr Funktionalität bieten, so daß das gesamte Betriebssystem nur die Funktionalität bietet, welche von der Anwendung benötigt wird. Genau das ist das Hauptziel von EPOS, das Auswählen und Anpassen der Bausteine, um ein anwendungsorientiertes Betriebssystem zu generieren.

Die in EPOS verwendeten Systembausteine werden durch PURE [1]bereitgestellt, womit EPOS als eine Art Erweiterung von PURE gesehen werden kann. In PURE wird das Betriebssystem als eine *Programmfamilie* [Par79] verstanden. Die Bausteine sind so entworfen, daß sie so wenig wie möglich unnütze Funktionalität im Sinne ihrer Aufgabe beinhalten, und portabel sind. Damit kann aus ihnen jede Art von Betriebssystem konstruiert werden. Sie sind als C++ Klassen implementiert.
PURE ist jedoch nicht für den Anwendungsprogrammierer gedacht, sondern für den Betriebsystembauer. EPOS macht die Funktionalität von PURE auch dem Anwendungsprogrammierer leicht zugänglich.

3.2 Das Design von EPOS

Wie schon erwähnt, wurde EPOS entworfen, um die Lücke zwischen den PURE-Bausteinen und den parallelen Anwendungen zu reduzieren. Dabei folgt EPOS den Grundlagen des objekt-orientierten Designs. Die 3 Hauptelemente sind die *Systemabstraktionen*, die *Szenarioadaptoren* und die EPOS-*Schnittstellen*. Durch das "Verstecken" der PURE-Bausteine und durch die Möglichkeit neue Systemabstraktionen automatisch zu generieren, verkleinern die ersten beiden Elemente die Lücke zwischen der Anwendung und den Bausteinen. Durch das dritte Element wird das EPOS-Repository dem Anwendungsprogrammierer zur Nutzung angeboten.

3.2.1 Systemabstraktionen

Die Systemabstraktionen sind anwendungsfertige Klassen, die von den Systemprogrammierern konstruiert und konfiguriert wurden. Die Menge aller dieser Klassen bildet das EPOS-Repository.
EPOS ermöglicht die Konfigurierung auf einer höheren Abstraktionsebene als PURE, indem dem Anwendungsprogrammierer, die für ihn "unwichtigen[2]" Komponenten verborgen bleiben. Damit besitzt das EPOS-Repositoy weitaus weniger (sichtbare) Komponenten als das PURE-Repository.
Die Systemabstraktionen sind so weit wie möglich unabhängig von den

[1] *Portable Universal Runtime Executive*: siehe [SSPSS98]

[2] Unwichtige Komponenten (im Sinne der Konfigurierung) für den Anwendungsprogrammierer sind Komponenten, die er nicht selber selektieren bzw. konfigurieren kann, da sie eventuell automatisch ausgewählt und konfiguriert werden.

Ausführungsszenarien implementiert. Eine Abstraktion, die dieselbe Funktionalität in verschiedenen Ausführungsszenarien anbieten soll, kann an die entsprechenden Szenarien angepaßt werden. Verschiedene Abstraktionen, die dasselbe Szenario unterstützen sollen, unterscheiden sich meist nach einem Muster voneinander. Zum Beispiel weisen zwei *thread*-Abstraktionen (eine beschreibt eine *single task* und die andere eine *multi-task* Umgebung) mehrere Gemeinsamkeiten auf. Ebenfalls können eine *thread*-Abstraktion und eine *mailbox*-Abstraktion für ein Multi-Prozessor-Szenario dieselben Synchronisationsmechanismen besitzen. Damit ist die automatische Generierung von neuen Systemabstraktionen möglich.
Das Zusammensetzen dieser anpaßbaren, szenario-unabhängigen Systemabstraktionen erfolgt durch die *Szenarioadaptoren.*

3.2.2 Szenarioadaptoren

Die Aufgabe der Szenarioadaptoren ist die Anpassung der existierenden Systemabstraktionen an das für sie spezifische Ausführungsszenario.
Da die Szenarioadaptoren die Anforderungen der Anwendung an das Betriebssystem von denen der Laufzeitumgebung trennt, können mit deren Hilfe leicht Aspekte eines Systems modelliert werden, welche dann einer szenario-unabhängigen Systemabstraktion entsprechen.
So kann beispielsweise eine *thread*-Abstraktion, die an die Benutzung im *single-* oder *multiple* Adressraum angepaßbar ist, zu einer Anwendung gebunden oder in einen μ-Kern integriert werden, der dann entweder lokale Aufrufe oder Fernaufrufe unterstützt.

3.2.3 EPOS-Schnittstellen

Zur Konstruktion von Betriebssystemen auf der Stufe eines Anwendungsprogrammierers existieren in EPOS automatische Werkzeuge. Dadurch ist der Anwendungsprogrammierer nicht mehr gezwungen das Repository zu durchsuchen und Klassen zu kombinieren oder zu spezialisieren. Diese Werkzeuge werden durch die sogenannten *Inflated Interfaces* ermöglicht.
Eine solche Schnittstelle für eine Systemabstraktion enthält nicht nur *eine* Darstellung für diese Abstraktion, sondern eine Menge der gebräuchlichsten Darstellungen. Daher werden die EPOS-Schnittstellen auch als "aufgebläht" bezeichnet. Beispiele für diese Schnittstellen sind *thread*, *task* oder *address space*. Die EPOS-Schnittstellen für die *thread*-Abstraktion besitzen verschiedene Sichten eines *Threads*, beispielsweise *pthreads* und *native* PURE *threads*. Mehrere Schnittstellen für eine Abstraktion existieren nur, falls unzusammenhängende Sichten exportiert werden müssen.
Die EPOS-Schnittstellen stammen aus klassischen Informatikbüchern und Systemhandbüchern, dennoch können und sollen sie vom Nutzer verändert und erweitert werden.

Die Nutzung dieser Schnittstellen ermöglicht es dem Anwendungsprogrammierer, seine Erwartungen an das Betriebssystem, durch das Aufschreiben bekannter System(-Objekt-)Aufrufe im Quellcode der Anwendung, leicht auszudrücken. Die EPOS-Schnittstellen sind nur Werkzeuge, um die Systemabstraktionen nach außen anzubieten. Dabei entsprechen sie nicht nur einer Klasse, sondern einem Satz von szenario-spezifischen Klassen. Die Systemkonfigurierung erfolgt dann, indem jede Schnittstelle durch ein Werkzeug an eine ihrer szenario-spezifischen Implementationen gebunden wird.

3.2.4 Exklusive und partielle Realisierungsbeziehungen

Zur Unterstützung der Systementwicklung mittels der EPOS-Schnittstellen wurden in EPOS zwei neue objektorientierte Entwurfsnotationen eingeführt: *partiell realisieren* und *exklusiv realisieren*. Diese stellen eine Beziehung zwischen einer EPOS-Schnittstelle und der Klasse dar, durch welche diese realisiert werden sollen.
Eine Klasse, die zur Schnittstelle in einer Beziehung *partiell realisiert* steht, implementiert nur einen speziellen Teil dieser Schnittstelle. *Exklusive Realisierung* bedeutet in diesem Sinne, daß nur eine von mehreren verschiedenen Realisierungen zu einen Zeitpunkt mit der Schnittstelle verbunden ist. Beide Notationen sind im Bild 3.1 dargestellt.

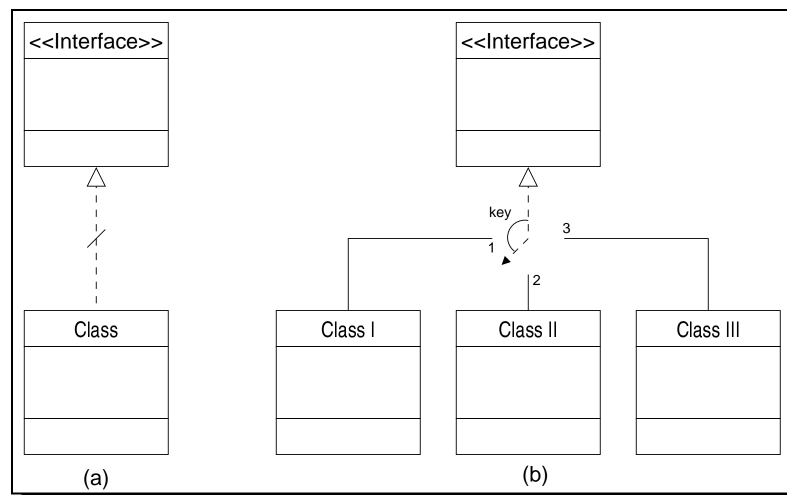


Bild 3.1: *partiell realisieren* (a) und *exklusiv realisieren* (b) Beziehungen

Jede Klasse, die sich in einer *exklusiv realisieren*-Beziehung befindet, wird mit einem Schlüssel versehen. Indem diesem Schlüssel ein Wert gegeben wird, wird eine spezielle, normalerweise *partielle*, Realisierung für diese Schnittstelle ausgewählt. Dies erfolgt zum Zeitpunkt der Erstellung des Systems durch den Betriebsystemgenerator, und nicht schon zum Entwicklungszeitpunkt.

Die Konfigurierung des Betriebssystems erfolgt durch die einfache Auswahl eines Wertes für diesen Schlüssel. Diese Schlüssel sind in eine Konfigurationsdatei eingetragen und werden zur Übersetzungszeit des Systems beachtet.

3.3 Die Umsetzung von Epos

Die automatische Generierung eines anwendungsorientierten Betriebssystems erfolgt in Epos *top-down*. Diese Strategie beginnt bei der Anwendung, in welcher der Programmierer deren Anforderungen an das Betriebssystem, mittels der Epos-Schnittstellen, spezifiziert hat. Wurde eine Anwendung in dieser Art entworfen und implementiert, kann sie an ein Analysewerkzeug übergeben werden, das syntaktische und Datenfluß-Untersuchungen durchführt. Dabei wird bestimmt, welche Systemabstraktionen wirklich zur Unterstützung der Anwendung benötigt werden, und wie diese aufgerufen werden. Die Ausgabe dieses Analysators ist ein "Bauplan" für das zu konstruierende Betriebssystem (siehe Bild 3.2). Dieser besteht aus einer Menge von Epos-Schnittstellen.
Leider ist dieser nicht komplett, da gewisse Aspekte noch nicht berücksichtigt sind. Dies sind Aspekte, welche nicht aus der Analyse der Anwendung geschlußfolgert werden können, da sie in der Anwendung nicht enthalten sind, wie beispielsweise die Anzahl der verfügbaren Prozessoren oder die Zielarchitektur. Für die Konstruktion eines möglichst genau auf seine Aufgabe zugeschnittenen Betriebssystems sind diese Faktoren aber fundamental.

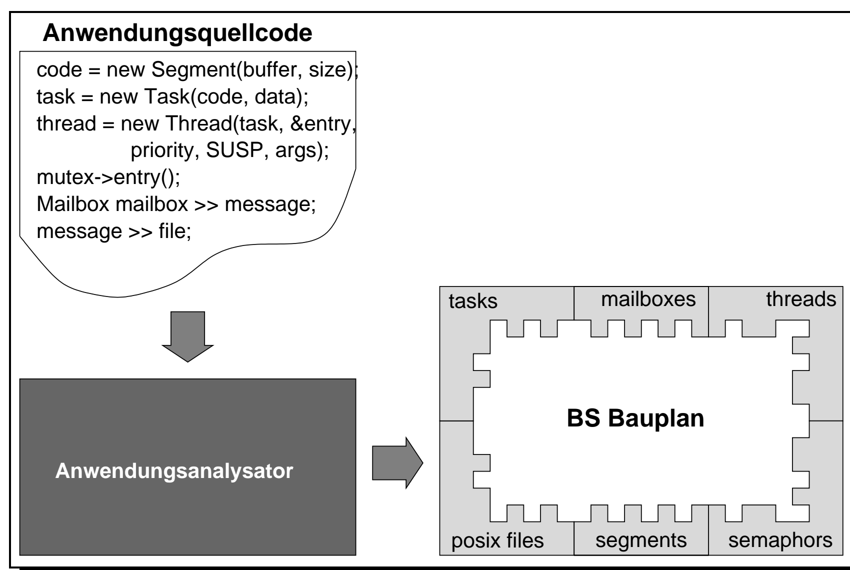


Bild 3.2: Erstellung des Bauplans

Um Angaben zu diesen Aspekten zu erhalten ist also eine Interaktion mit dem Nutzers notwendig. Somit erfolgt die Beschreibung der verfügbaren Ressourcen durch die Betriebssystementwickler und die Beschreibung des Ausführungsszenarios für die Anwendung durch den Nutzer via visueller Werkzeuge.
Mittels dieser Informationen seitens des Nutzers über das Ausführungsszenario erfolgt dann eine Verfeinerung des Systembauplans durch eine Abhängigkeitsanalyse. Der verfeinerte Bauplan wird in einer Konfigurationsdatei, in Form von Werten für die Schlüssel der exklusiven Realisierungen (siehe 3.2.4), abgelegt. Durch die Definition dieser Schlüssel werden die EPOS-Schnittstellen, welche durch den Anwendungsprogrammierer angegeben wurden, an szenario-spezifische Implementationen gebunden. Nach der Übersetzung der Systems steht am Ende dieses Prozesses das anwendungsorientierte EPOS, wie es im Bild 3.3 dargestellt ist.

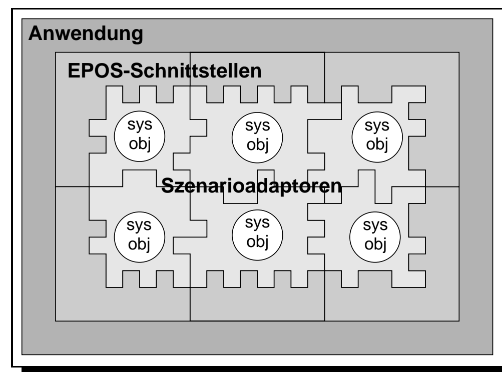


Bild 3.3: anwendungsorientiertes EPOS

3.4 Einordnung des neuen Werkzeugs

Wie im letzten Abschnitt schon erkennbar war, bietet sich der Einsatz eines Werkzeuges zur Verfeinerung des vom Anwendungsanalysator gelieferten Bauplans an. Genau an dieser Stelle wird das neue Werkzeug eingesetzt werden. Das heißt, es dient zur Beschreibung des Anwendungsszenarios und zur eventuellen manuellen Veränderung des Bauplans. Die manuelle Auswahl einer Systemabstraktion für eine EPOS-Schnittstelle kann notwendig werden, falls die Abhängigkeitsanalyse keine eindeutige Zuweisung treffen kann.
Damit besetzt das zu implementierende Werkzeug eine zentrale Stelle der Konfigurierung zwischen den EPOS-Schnittstellen und den Systemabstraktionen.

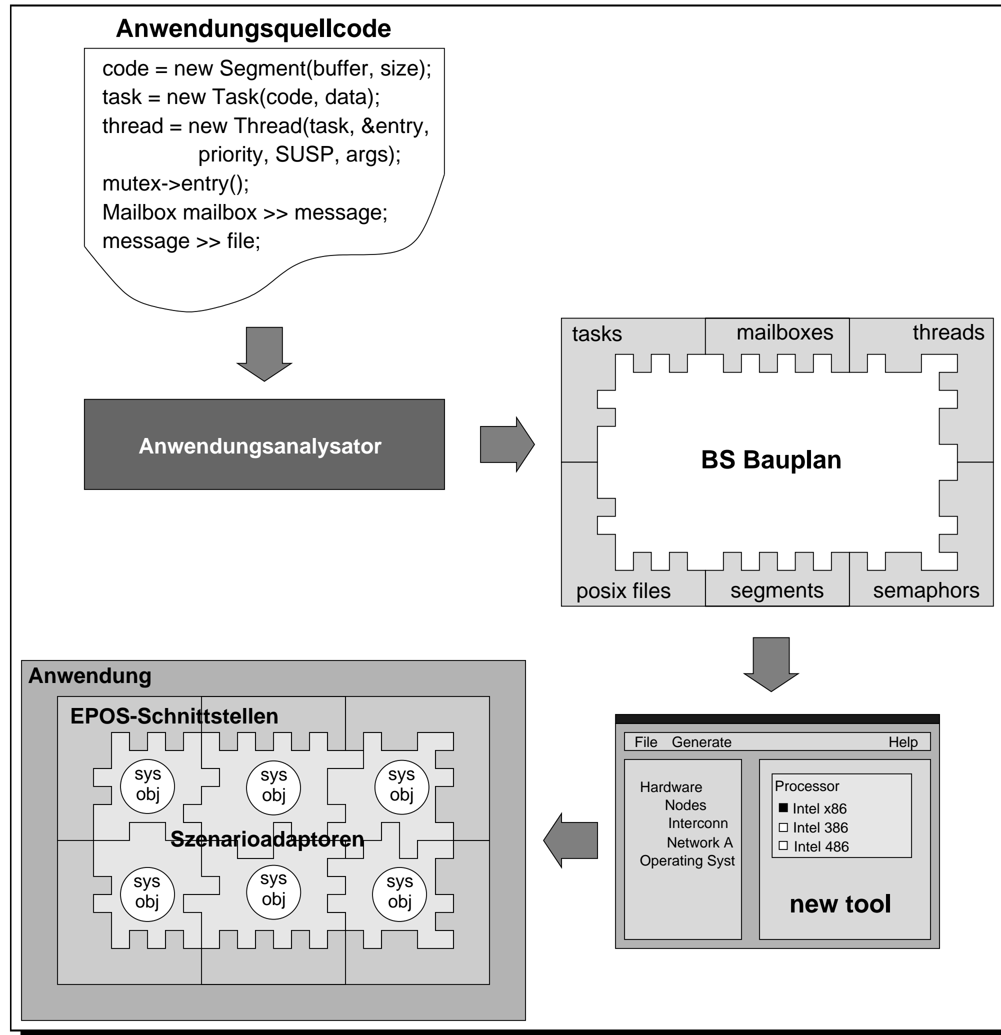


Bild 3.4: EPOS-Konfigurierungsprozeß

Kapitel 4

Die eXtensible Markup Language

Da XML zur Definition der Eingabesprache für XCONF dienen soll, werden in diesem Kapitel die grundsätzliche Funktionalität und Philosophie von XML beschrieben. Weiter Informationen sind in [GP98] und im WWW unter [Con98] zu finden.

4.1 Warum XML ?

4.1.1 Die Vision

Die *eXtensible Markup Language* ist derzeit in aller Munde. Sie bietet die Möglichkeit, Daten allgemeinverständlich zu beschreiben, und stellt somit ein universelles Datenformat dar, was bis dato fehlte. Die erste Version der XML-Spezifikation wurde im Februar 1998 durch das World Wide Web Consortium vorgelegt. Seitdem steigt die Verbreitung dieses neuen Datenformats stetig, nicht zuletzt auch, oder besonders, da viele große Unternehmen, wie z.B. SAP und Microsoft, volle Unterstützung dafür anbieten, und damit dessen Bekanntheit und Akzeptanz fördern.

XML ist "extensible", da es sich nicht um ein festes Format, wie zum Beispiel HTML , im Sinne einer bestimmten Auszeichnungssprache handelt. Es handelt sich vielmehr um eine textbasierte Metasprache[1], welche die Darstellung und die Manipulation von strukturierten Daten erlaubt, so daß diese von einer Vielzahl von Anwendungen genutzt werden können. XML stellt Regeln bereit, um eine Vielzahl konkreter Auszeichnungssprachen für die verschiedenen Dokumentarten zu erstellen.

[1] Eine Metasprache macht Aussagen über Objekte, wogegen in einer Sprache die Aussagen die Objekte sind.

4.1.2 Die Entwicklung

Das HTML-Dilemma

Die *HyperText Markup Language* wurde 1989 vorgestellt. Das primäre Entwicklungsziel war es, den Austausch von wissenschaftlichen Dokumenten mit der Möglichkeit des Verweisens auf andere Dokumente zu unterstützen. Der Vorteil von HTML liegt in deren Einfachheit, welche aber andererseits auch ihren Preis hat. So ist HTML nicht für anspruchsvollere Aufgaben als zur Darstellung von Daten einsetzbar, da sie die folgenden Defizite aufweist:

1. Erweiterbarkeit: In HTML ist es nicht möglich, eigene Tags[2] oder Attribute zur semantischen Auszeichnung der Daten zu definieren. Damit ist HTML eine reine Präsentationssprache, was bedeutet, daß HTML-Dokumente keine Informationen über die Semantik des Inhalts enthalten sondern nur Formatinformationen zur Darstellung der Daten.

2. Struktur: Durch die fehlenden Datenstrukturen in HTML ist es nicht möglich, den Zusammenhang der Daten zu beschreiben.

3. Validierung: HTML fehlen Sprachspezifikationen, welche Anwendungen, die HTML-codierte Daten verarbeiten sollen, die Überprüfung der strukturellen Validät ermöglichen.

Diese Defizite von HTML werden häufig als “HTML-Dilemma” bezeichnet.

SGML

Die *Standard Generalized Markup Language* ist seit über 10 Jahren als internationaler Standard anerkannt. Als Metasprache stellt SGML Vorschriften zur formalen Definition von Auszeichnungsprachen zur Verfügung.
SGML dient zur Definition, Identifikation und Benutzung der Struktur und des Inhalts von Dokumenten. Ihre äußerst flexible Architektur ermöglicht die Aufbereitung von Dokumenten für beliebige Medien, ohne die Struktur der Daten zu verlieren.
So ist zum Beispiel HTML eine Anwendung von SGML, da alle bisherigen HTML-Versionen in SGML definiert wurden.
Das Problem von SGML ist die Komplexität, durch welche die Entwicklung von SGML-Anwendungen teuer und kompliziert ist.

XML zwischen SGML und HTML

Während HTML aus dem Grund der fehlenden Erweiterbarkeit für komplexere Anwendungen ungeeignet ist, erweist sich SGML wegen der hohen Komplexität als nur begrenzt einsetzbar, besonders im Internet. Hier setzt XML

[2]Ein Tag ist ein Markup, also eine Markierung, zwischen < und >.

an, welche die Funktionalität von SGML für das Internet anbietet, allerdings mit wesentlich geringerer Komplexität. Die Reduzierung der Komplexität erfolgte durch das Weglassen von für das Internet überflüssigen, zu wenig genutzten oder als zu komplex angesehenen SGML-Eigenschaften. Somit ist XML eine Teilmenge von SGML und wird häufig auch als "SGML *lite*" [Mic99] bezeichnet.

4.2 Die Sprache

4.2.1 Das Sprachkonzept

Das Grundkonzept von XML besteht in der Trennung von Inhalt, Struktur und Präsentation, wie im Bild 4.2.1 dargestellt. Der Inhalt wird durch sogenannte Tags ausgezeichnet. Diese können frei benannt werden und in einer beliebigen Anzahl auftreten. In ihrer Eigenschaft als Auszeichner enthalten sie Informationen über den Inhalt. So wird durch die Verschachtelung der Tags die Struktur der Daten abgebildet.

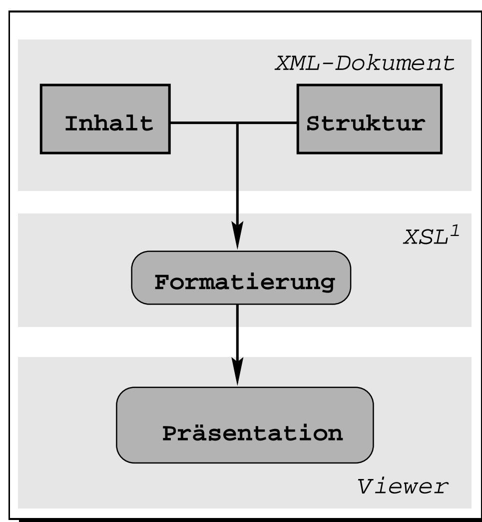


Bild 4.2.1: Das XML-Sprachkonzept
([1]XSL = eXtensible Style Language)

XML-Dokumente können, müssen aber nicht, eine formale Beschreibung ihrer Grammatik enthalten. Näheres dazu wird in 4.2.2 beschrieben.

Um ein XML-Dokument zu beschreiben, benötigt man Markup (Tags), Verarbeitungsanweisungen und eventuell Dokumenttyp-Definitionen.

4.2.2 Document Type Definition

Neben dem eigentlichen XML-Dokument, das die darzustellenden Informationen enthält, braucht (fast) jede XML-Anwendung eine *Document Type*

Definition (DTD), die vorzugsweise in einer eigenen Datei abgelegt wird. Die DTD ist die Layoutsprache zu XML. Sie beschreibt die Syntax und die logische Struktur, die für ein Dokument gelten soll, welches diese DTD benutzt. In der DTD kann der Nutzer sich seine eigenen Tags mit eigenen Attributen und deren Semantik und Abhängigkeiten zu anderen Tags definieren, mit denen er sein Dokument auszeichnen bzw. strukturieren will.
Die DTD ist vergleichbar mit einer Grammatik, durch die eine Sprache definiert wird. So bezieht sich die DTD nicht nur auf die syntaktische Richtigkeit von einzelnen Tags und deren Attributen, sondern es wird auch die Reihenfolge der auftretenden Tags berücksichtigt und überprüft.
Mit Hilfe der DTD ist es Programmen möglich, ein XML-Dokument auf strukturelle Fehler zu überprüfen oder neue Instanzen dieses Dokumenttypen zu bilden. Somit erweist sich die DTD als vorteilhaft, falls mehrere verschiedene Instanzen angelegt werden sollen. Die DTD dient dabei als Muster, das Angaben darüber enthalten kann, welche Elemente durch eine Instanz benutzt werden müssen bzw. welche benutzt werden dürfen.

Die Element-Definition

Jedes XML-Dokument enthält ein oder mehrere Elemente. Jedes dieser Elemente hat einen Typ und kann eine Menge von Attributspezifikationen haben. Die allgemeine Form einer Element-Definition lautet:

```
'<!ELEMENT' name content '>'
```

`name` steht hier für den Namen dieses Elements, und `content` kann eine der folgenden Formen haben:

- Das Schlüsselwort `'EMPTY'` zeigt an, daß dieses Element leer ist, also keine Zeichenkette und keine Kindelemente enthalten darf.
- Mittels der Angabe von `'ANY'` kann dieses Element beliebige Elemente enthalten, die in der DTD definiert sind.
- `'PCDATA'` steht für "Parsed Character Data" und zeigt an, daß dieses Element nur einfachen Text enthalten darf.
- Für `content` kann auch eine Kombination von Kindelementen stehen.
- Letztendlich ist auch eine Kombination von Kindelementen und Zeichenketten möglich.

Das folgende Beispiel zeigt die Definition einer Liste, die eine beliebige Anzahl von Listenelementen enthalten darf:

```
<!ELEMENT List (ListElement)*>
```

Attribut-Definition

Durch die Angabe von Attributen können Elemente mit zusätzlichen Informationen versehen werden. Die allgemeine Form einer Attributlistendefinition ist:

```
'<!ATTLIST' elementname
  attname1 type default
  ...
  attnameN type default '>'
```

Die Attributliste definiert die für das Element `elementname` möglichen oder notwendigen Attribute. Der Name des jeweiligen Attributes wird durch `attname` repräsentiert. Für jedes definierte Attribut ist die Angabe eines Typs notwendig. Dies kann einer der Folgenden sein:

- `ID`: Dieses Attribut dient zur eindeutigen Kennung für das Element.
- `IDREF` verweist auf ein Element im Dokument, indem der Wert des Attributes die ID eines Elements ist.
- `CDATA` erlaubt die Angabe einer beliebigen Zeichenkette.
- `NMTOKEN`: Der Wert des Attributes ist eine Zeichenkette, die aus Buchstaben, Ziffern und den Zeichen '.', ':', '-', '_' bestehen kann.

Die Deklaration einer Vorbelegung (`default`) eines jeden Attributes kann einen der folgenden Werte haben:

- `#REQUIRED`: Das Attribut muß immer vorhanden sein.
- `#IMPLIED`: Das Attribut ist optional.
- `#FIXED`: Das Attribut hat immer denselben Wert, ist also eine Konstante. Der Wert wird direkt nach dem Schlüsselwort in einfachen Anführungszeichen angegeben.

4.2.3 XML-Dokumente

XML-Dokumente sind eine durch XML beschriebene Klasse von Datenobjekten. Es wird zwischen zwei Typen von XML-Dokumenten unterschieden.
Dokumente, denen eine DTD zugeordnet ist, werden als gültig (*valid*) bezeichnet, falls sie gegen keine der in der DTD definierten Regeln verstoßen. Ein XML-Dokument muß aber nicht DTD-konform sein. Erfüllt dieses die grundlegenden Anforderungen an die XML-Syntax, wird es als wohlgeformt (*well-formed*) bezeichnet. Diese grundlegende Sprachspezifikation, die für alle XML-Dokumente gelten muß, besteht im wesentlichen aus folgenden Punkten:

- jeder geöffnete Tag muß explizit geschlossen werden
- Tags ohne Inhalt müssen explizit geschlossen werden oder mit `/>` enden
- Attributwerte müssen in doppelte Anführungzeichen gesetzt werden
- das Markup muß streng hierarchisch gegliedert sein
- es dürfen keine Markup-Zeichen (`<` oder `&`) im Text vorkommen

Zusätzlich sollte am Anfang des Dokumentes der Hinweis auf die verwendete XML-Version stehen. Weiterhin ist auch die Angabe der Art der Kodierung sinnvoll. Diese Verarbeitungsanweisungen werden zwischen `<?` und `?>` gestellt. Sie dienen den Programmen, die XML-Dokumente verarbeiten.

Die logische Struktur eines XML-Dokumentes ist durch die Anordnung der Tags gegeben. In Hinsicht auf die physische Struktur kann ein Dokument aus einer beliebigen Anzahl physischer Einheiten bestehen, unabhängig von der logischen Struktur.

4.2.4 Document Object Model

Ein XML-Dokument ist also eine Textdatei, die neben dem Inhalt (reine Daten) auch deren Struktur (Metadaten) enthält. Diese können von einem XML-Parser an verschiedene Anwendungen übergeben werden.

Das *Document Object Model* (DOM) wurde als ein standardisiertes Datenzugriffsmodell entwickelt. Es dient als plattform- und sprachunabhängige Schnittstelle, die Programmen den dynamischen Zugriff auf den Inhalt und die Struktur von XML-Dokumenten ermöglicht.

Das DOM bildet die Struktur eines XML-Documentes in einer Baumstruktur ab. Diese ist die Arbeitsgrundlage für alle DOM-basierten[3] XML-Parser. Ein solcher XML-Parser trennt die Tags von den enthaltenen Informationen und bildet die Schachtelung der Tags als eine Hierarchie ab. Damit stehen die Informationen der maschinellen Weiterverarbeitung zur Verfügung. Alle Tags und ihre Inhalte sowie deren Relationen untereinander, werden als Objekte definiert und in einer Objekthierarchie abgebildet. Auf diese Objekte kann, zum Beispiel mittels Java, zugegriffen werden.

[3]Eine zweite oft genutzte Schnittstelle für die XML-Programmierung ist SAX (*Simple API for XML*). SAX-basierte Parser nutzen kein DOM.

Kapitel 5

Entwurf

5.1 Anforderungsanalyse

5.1.1 Funktionale Anforderungen

Das zu implementierende Werkzeug soll der Konfigurierung von Betriebssystemen dienen. Die dazu notwendigen Informationen über das System werden dem Programm in einer Datei zur Verfügung gestellt. Diese Konfigurierungsbeschreibung besteht aus den Angaben über die auswählbaren Komponenten und deren Abhängigkeiten untereinander, also den Konfigurationsregeln.

Alle diese Informationen sind mittels geeigneter Elemente grafisch darzustellen. Während der Nutzer das System durch die Interaktion mit der grafischen Oberfläche konfiguriert, soll die Einhaltung der Konfigurationsregeln automatisch durch das Werkzeug kontrolliert werden.

Eine durch die Konfigurierung entstandene Konfigurationsbeschreibung soll in Form von Präprozessoranweisungen in einer Datei gespeichert werden können. Damit ist die Existenz mehrerer Konfigurationen zu einem System möglich. Weiterhin soll die Möglichkeit bestehen, bereits vorhandene Konfigurationen zu laden und zu verändern.

Um dem Werkzeug eine zentrale Stellung im Konfigurierungsprozeß zu sichern, ist es wichtig, aus dem Werkzeug heraus auch weitere Werkzeuge aufrufen zu können. So kann zum Beispiel mit der Möglichkeit `make` aufzurufen, durch dieses Werkzeug aus einer Konfigurierungsbeschreibung indirekt ein "fertiges" System erzeugt werden.

5.1.2 Qualitätsanforderungen

Das Hauptziel dieses Werkzeuges ist es, dem Nutzer eine einfache und übersichtliche Konfigurierung von Betriebssystemen zu ermöglichen. Dies soll durch die grafische Oberfläche und die automatische Kontrolle der Einhaltung der Konfigurationsregeln erreicht werden. Bei der Verletzung ei-

ner dieser Regeln wird der Nutzer darüber informiert werden, wobei ihm auch der Grund der Verletzung mitgeteilt wird. Dadurch soll es dem Nutzer ermöglicht werden, durch das Ändern anderer Parameter die Änderung des aktuellen Parameters, ohne eine Verletzung der Regeln zu ermöglichen.
Da das Werkzeug verschiedene Konfigurierungsbeschreibungen darstellen können soll, und dazu viele unterschiedliche Konfigurationsbeschreibungen existieren können, muß beim Laden einer Konfigurationsbeschreibung deren Gültigkeit geprüft werden. So ist eine Konfigurationsbeschreibung gültig, falls sie zur aktuellen Konfigurierungsbeschreibungen paßt, also alle ihre Parameter auch in der Konfigurierungsbeschreibungen enthalten sind, und diese Parameter die Konfigurationsregeln einhalten.
Das Werkzeug wird zwar speziell für die Konfigurierung des Betriebssystems EPOS entworfen, soll aber auch für die Konfigurierung anderer Systeme genutzt werden können. Eventuell dazu notwendige Erweiterungen bzw. Änderungen sollen so minimal wie möglich sein.

5.1.3 Anforderungen zur Systemrealisierung

Der Einsatz dieses Werkzeuges soll weitestgehend unabhängig von der Hardware und dem Betriebssystem möglich sein. Abhängigkeiten werden durch die Schnittstellen zu anderen Werkzeugen bestehen, welche aus diesem Werkzeug aufgerufen werden können.

5.2 Systeminformationen

Grundlegend für den Entwurf eines Programms, ist das Wissen über die zu verwaltenden Informationen. In diesem Fall sind dies Informationen, die ein Betriebssystem beschreiben.
Dieses System besteht aus *Systemkomponenten*. Daher ist dies die grundlegende Informationseinheit. Zu einer solchen Komponente gehören Informationen wie:

- der Name der Komponente
- die Schnittstelle der Komponente
  Diese entspricht dem Namen einer Klasse, welche diese Komponente repräsentiert. Eine Komponente kann intern auch durch mehr als eine Klasse implementiert sein.
- Abhängigkeiten gegenüber anderen Komponenten
  So kann zum Beispiel die Selektion einer Komponente die Selektion einer weiteren Komponente implizieren oder verbieten.

Weiterhin ist es sinnvoll, diese Komponenten in gewisse Gruppen einzuteilen. So können beispielsweise mehrere Komponenten für eine Aufgabe

implementiert sein. Zwischen den Komponenten einer Gruppe bestehen oft bestimmte Abhängigkeiten. Meist stehen sich diese Komponenten alternativ gegenüber. Das heißt, daß die Auswahl einer Komponente die Auswahl einer zweiten Komponente aus derselben Gruppe verbietet. Also darf und muß in einer solchen *alternativen Gruppe* immer genau eine Komponente ausgewählt sein. In einer *optionalen Gruppe* hingegen dürfen beliebig viele Komponenten, auch keine, selektiert sein.
Die Auswahl beziehungsweise Nicht-Auswahl einer Komponente entspricht einer *qualitativen Einstellung* am System. Neben diesen existieren auch *quantitative Parameter* des Systems. Einer kann zum Bespiel die Anzahl der Prozessoren der Plattform sein. Diese quantitativen Parameter entsprechen eher einer Variablen als einer Komponente, und dienen den Einstellungen innerhalb der Komponeten.

Mittels dieser Informationen läßt sich ein Betriebsystem beschreiben, was die Vorraussetzung für eine leichte Konfigurierung ist.

5.3 Die Schnittstellen

5.3.1 Die Eingabesprache

Wie schon erwähnt, dient die Konfigurierungsbeschreibung als Eingabe für das Werkzeug. Um diese auszudrücken, ist der Entwurf einer Sprache notwendig. Diese Sprache soll die einfache strukturierte Angabe von Daten und deren Bedeutung ermöglichen. Somit genügt eine Metasprache wie XML.

Warum XML?

XML bietet gegenüber einer selbst definierten Sprache, z.B. einer mittels `lex` und `yacc` entworfenen, den großen Vorteil, daß sie ein Standard ist. Mit Hilfe der *Document Type Definition* (siehe 4.2.2) kann eine eigene Sprache beschrieben werden. Es können somit eigene "Befehle" und deren möglichen Verschachtelungen definiert werden.
Da die "Befehle" oder Steuerzeichen wie in HTML in Form von Tags angegeben werden, ist eine leichte Erstellung von XML-Dokumenten in Texteditoren möglich.
Weiterhin existieren für die Nutzung von XML freie Parser, Editoren und Konverter.

Die Grammatik

Beim Entwurf der Grammatik der Eingabesprache sind natürlich die in 5.2 beschriebenen Informationen zu berücksichtigen. Da das Werzeug im Projekt EPOS entwickelt wurde, spiegelt sich dies an Teilen der Grammatik wieder,

besonders in der Auswahl der Tagnamen. Falls für andere Betriebssysteme andere Tagnamen benutzt werden sollen, kann entweder die vorhandene Grammatik leicht verändert werden, oder eine eigene Grammatik entworfen werden. XML-Dokumente, die einer anderen Grammatik unterliegen, können durch einen Konverter in die hier benutzte Grammatik überführt werden.
Wie in 5.2 schon angedeutet, sind Elemente zur Repräsentation einer Komponente, Elemente zur Darstellung von Gruppen und letztlich Elemente für die quantitativen Parameter zu entwerfen.
Alle Elemente können minimal 3 Parameter haben. Dies sind:

- `name`: Der Wert dieses Attributes entspricht dem programminternen Namen der Komponente, der Gruppe oder der Variable.
  Dieses Attribut muß angegeben werden, und der Wert muß eindeutig sein, darf also nur einmal in der Konfigurierungsbeschreibung erscheinen.

- `text` gibt den Text an, der auf der graphischen Oberfläche für dieses Element verwendet werden soll, und ist somit für jedes Element anzugeben.

- `help` kann Informationen über das Element enthalten. Es ist auch die Angabe einer URL möglich, welche die WWW-Adresse eines Dokuments mit Informationen zu diesem Element enthält. Die Angabe dieses Attributes ist nicht zwingend.

Eine KOMPONENTE wird durch ein Tag namens `<realisation>` repräsentiert. Dieses Tag besitzt neben den oben genannten Attributen noch die folgenden:

- `class` gibt den Namen der Klasse an, welche die Schnittstelle dieser Komponente implementiert.

- `header` enthält den Namen der Header-Datei der Klasse und deren Pfad, welche die Komponente implementiert.

Die QUALITATIVEN EINSTELLUNGSMÖGLICHKEITEN werden durch die Tags `<interface>` für alternative Gruppen und `<multiselect>` für optionale Gruppen realisiert. Da eine solche Gruppe eine Menge von Komponenten ist, sind deren Kindelemente `<realisation>`-Tags. Eine Gruppe muß mindestens eine Komponente enthalten.
Diese zwei Tags zur Gruppierung der Komponenten können neben den oben schon genannten Attributen `name`, `text` und `help` noch die folgenden Attribute besitzen:

- `default` enthält die Namen der Komponenten, die aus der Gruppe als Vorgabewert ausgewählt sein sollen. In einer alternativen Gruppe (`<interface>`-Tag), muß der Name genau einer Komponente angegeben sein. Innerhalb einer optionalen Gruppe (`<multiselect>`-Tag) entspricht der Wert einer Liste von Komponentennamen. Die angegebenen Komponenten müssen natürlich auch in dieser Gruppe vorhanden sein.

- Das `class`-Attribut ist nur für alternative Gruppen verfügbar. Der Wert dieses Attributes ist der Name der Klasse, welche die EPOS-Schnittstelle (siehe 3.2.3) repräsentiert. Diese wird später mittels `typedef` zum Synonym für die Klasse der ausgewählten Realisierung (siehe Anhang C.2).

Das Attribut `name` einer alternativen Gruppe realisiert im Sinne von EPOS eine der EPOS-Schnittstellen (siehe 3.2.3). Diese Struktur repräsentiert die schon im Kapitel 3.2.4 erwähnte *exklusiv realisieren*-Beziehung, die sehr wichtig für die EPOS-Konfigurierung ist.
Optionale Gruppen kommen während der EPOS-Konfigurierung nicht zum Einsatz, können aber bei der Konfigurierung anderer Systeme durchaus wichtig sein.
Das folgende Beispiel zeigt eine alternative Gruppe von Komponenten.

```
<interface name="NETWORK" text="Network"
           default="ETHERNET" class="Network">
    <realisation name="ETHERNET" text="Ethernet"
                 class="Ethernet"
                 header="system/header/ethernet.h"/>
    <realisation name="FAST_ETHERNET" text="Fast-Ethernet"
               class="Fast_Ethernet"
               header="system/header/fast_ethernet.h"/>
</interface>
```

Soll einer alternativen Gruppe auch keine “echte” Realisierung zugeordnet werden können, wird dies mit dem `<notneeded>`-Tag innerhalb der Gruppe angegeben. Im Sinne von EPOS kann die Schnittstelle dann auch durch eine *Dummy*-Komponente realisiert werden. Ein `<notneeded>`-Tag entspricht einem `<interface>`-Tag mit einem festen Namen (`NOT_NEEDED`) und einem festen Text (`Not needed`). Also müssen diese Attribute nicht mehr explizit angegeben werden.

Die für die Konfigurierung von Systemen ebenfalls wichtigen QUANTITATIVEN EINSTELLUNGSMÖGLICHKEITEN werden durch das Tag `<input>` realisiert. Dieses besitzt zusätzlich zu den schon angegeben Attributen die folgenden:

- `type` muß entweder den Wert `number` oder den Wert `text` haben. Es wird also angegeben, ob der Parameter beziehungsweise die Variable eine Zahl oder ein Text ist.

- `range` dient zur Angabe eines Wertebereiches einer Variablen, beispielsweise `range=''1..500''`. Damit ist die Angabe eines solchen Attributes nur für Zahlenwerte möglich, also falls `type=''number''`. Es kann auch nur ein minimaler Wert angegeben werden, zum Beispiel `range=''1..''`. Dann bleibt die obere Grenze offen.

- `length` gibt die Länge des Eingabefeldes an. Dieses Attribut ist eigentlich für die Verwendung bei Zeichenketten (`type=''text''`) gedacht, kann aber auch für Zahleneingaben verwendet werden. Bei diesen wird die Länge des Eingabefeldes sonst auf 5 gesetzt.

- `default` gibt einen Vorgabewert an, mit dem dieser Parameter initialisiert wird. Bei Zahleneingaben muß sich dieser an die durch `range` gemachten Einschränkungen halten.

Ein solches Tag zur einfachen Angabe von quantitativen Parametern, ihren Werten und Grenzen besitzt keine Kindelemente.

Bisher können nun fast alle in 5.2 angegebenen Informationen zu einem System ausgedrück werden, außer die Abhängigkeiten zwischen den Komponenten. Diese sind teilweise schon durch die Gruppen darstellbar, jedoch existieren weitere Abhängigkeiten, die explizit angegben werden müssen. So kann die Auswahl einer Komponente zum Beispiel die Auswahl einer weiteren Komponente implizieren oder die Auswahl einer Komponente einer anderen Gruppe verbieten.
Um dies ausdrücken zu können, werden zwei weitere Attribute eingeführt:

- `pre` enthält eine beziehungsweise mehrere Bedingungen, die gelten müssen, damit die Komponente ausgewählt werden kann, also eine *Vorbedingung*. Auch für Gruppen ist dieses Attribut verfügbar. Ist bei diesen die Vorbedingung ungültig, ist keines der Gruppenelemente auswählbar. Ist die Vorbedingung eines Eingabefeldes nicht erfüllt, können keine Eingaben getätigt werden.

- `pos` gibt die Bedingungen an, die nach der Auswahl einer Komponente oder nach dem Eintrag in ein Eingabefeld erfüllt sein müssen. Dieses Attribut entspricht somit einer *Nachbedingung*. Es ist auch für Gruppen verfügbar, womit die Nachbedingung für alle Komponenten dieser Gruppe gilt, und nicht mehr für jede Komponente einzeln angegeben werden muß.

In diesen Regeln kann angegeben werden, ob in einer bestimmten Gruppe eine bestimmte Komponente ausgewählt sein muß oder nicht ausgewählt

sein darf. Andererseits kann auch ein quantitativer Parameter mit einem Wert verglichen werden. Es existieren die Vergleichoperationen auf Gleichheit (`==`) und Ungleichheit (`!=`), und für numerische Werte `<` und `>`. Zum Ausdrücken komplizierterer Regeln können einzelne Regeln mittels "Und" (`&&`) bzw. "Oder" (`||`) verknüpft werden. Beispiele für Regeln sind:

```
pre="(TASK == TASK_MULTI) || (THREAD == THREAD_MULTI)"
pre="ADDRESS_SPACE == ADDRESS_SPACE_SINGLE"
pos="PROCESS_MODEL == PROCESS_SINGLE"
```

Damit können die wichtigsten Informationen zur Konfigurierung eines Systems ausgedrückt werden. Um dem Nutzer bei der Angabe der Informationen eine besseren Übersicht zu ermöglichen, wurden zwei weitere Tags zur Strukturierung der Konfigurierung in Teilbereiche entworfen. Diese Teilbereiche sollten eine logische Einheit ausdrücken.
Die Konfigurierungsbeschreibung kann grob in *Domänen*, wie beispielsweise Hardware und Betriebssystem, unterteilt werden. Diese werden durch das `<domain>`-Tag gekennzeichnet. Um diese Domänen nochmals zu untergliedern, existiert das `<section>`-Tag. Dieses Tag kann zum Beispiel die Domäne Betriebssystem nochmals in Abschnitte wie Speicher und Prozess unterteilen. Zur weiteren Gliederung der Informationen können in diese Abschnitte nochmals Abschnitte verschachtelt werden.
Ein Abschnitt enthält dann die zu ihm gehörenden Einstellungsmöglichkeiten wie `<interface>`, `<multiselect>` oder `<input>`.
Beide Tags besitzen die Attribute `name`, `text` und `type`. Das `<section>`-Tag besitzt zusätzlich noch das Attribut `pre`. Ist diese Vorbedingung nicht erfüllt, können im gesamten Abschnitt keine Einstellungen getätigt werden.

Die Konfigurierungsbeschreibung kann nun grob in Domänen unterteilt werden. Da jedes Xml-Dokument aber nur ein Wurzelelement (*root-element*) besitzen darf, und eine Konfigurierungsbeschreibung möglicherweise aus mehr als einer Domäne besteht, muß ein weiteres Tag entworfen werden, welches diese `<domain>`-Tags umschließt, und somit das Root-Element darstellt. Dieses Tag hat den Namen `<configuration>`. Es besitzt außer den oben schon erwähnten Attributen `name` und `text` die folgenden:

- `keys_file` ist der Name der Datei, in der die Schlüssel der Konfigurationsbeschreibung gespeichert werden sollen.
- `config_file` enthält den Namen einer Datei, in der weitere Konfigurationsinformationen, wie beispielsweise die Dateinamen der Klassen, gespeichert werden sollen.

Diese beiden Ausgabeformate werden in 5.3.2 beschrieben.

Die komplette Grammatik befindet sich im Anhang A.1. Im Anhang A.2 wird ein Ausschnitt einer für EPOS benutzten Konfigurierungsbeschreibung angegeben.

5.3.2 Die Ausgaben

Die Konfigurationsbeschreibung

Die Konfigurationsbeschreibung entspricht dem Bauplan für das Betriebssystem. Sie enthält die Schlüssel der Konfiguration. Im Sinne von EPOS bedeutet dies, daß hier die Realisierungen zu den EPOS-Schnittstellen definiert werden. Damit sind alle exklusiven Realisierungen durch partielle ersetzt (siehe 3.2.4). Weiterhin enthält die Konfigurationsbeschreibung alle quantitativen Einstellungen.
Für eine Konfigurierungsbeschreibung können mehrere dieser Konfigurationsbeschreibungen existieren. Die Ausgabe erfolgt in Form von Präprozessordirektiven in eine Header-Datei. Diese entsprechen der Macrodefinition:

```
#define identifier replacement
```

Eine solche Definition bewirkt, daß bei jedem Auftreten des Bezeichners `identifier` statt dessen der Ersetzungstext `replacement` eingesetzt wird. Diese Datei wird dann vor dem Übersetzen des Systems durch den Präprozessor abgearbeitet, womit durch textuelle Ersetzungen im Quellcode das System "zugeschnitten" wird.
Die `identifier` sind die in der Konfigurierungsbeschreibung, dem XML-Dokument, angegebenen Werte des Attributes `name`. Für eine EPOS-Schnittstelle (`<interface>`) entspricht der Ersetzungstext dem Namen einer der dafür verfügbaren Realisierungen. Bei einer optionalen Auswahl (`<multiselect>`) wird jede Komponente dieser Gruppe mit dem Ersetzungstext `1` oder `0` ausgegeben, je nachdem ob sie ausgewählt ist oder nicht. Der Ersetzungstext für eine quantitative Einstellung (`<input>`) ist ihr Wert.
Eine Konfigurationsbeschreibung hat somit die Form:

```
#ifndef filename_h
#define filename_h 1
......
#define INTERFACE           REALISATION
#define COMPONENT_X         1
#define COMPONENT_Y         0
#define VARIABLE            VALUE
......
#endif
```

Hier entspricht `filename` dem Namen der Datei, in welcher sich diese Angaben befinden. Ein Beispiel für eine Konfigurationsbeschreibung für EPOS befindet sich im Anhang C.1.

Die Konfigurationsinformationen

Diese Ausgabe enthält fast alle Informationen über das System, die auch schon im XML-Eingabedokument enthalten sind. Das bedeutet, es enthält Informationen über alle Komponenten, also deren Namen, Namen der Klasse, welche diese Komponente implementiert und Angaben darüber, wo sich der Quellcode dieser Klasse befindet. Auch in diesem Dokument sind diese Klassen gruppiert. Die Ausgabe erfolgt in Form von Präprozessordirektiven. In dieser Datei befindet sich ein Verweis auf die Datei, die die Schlüssel der Konfiguration enthält, also die Konfigurationsbeschreibung. Diese wird mittels einer `include` Direktive durch den Präprozessor eingelesen und dann abgearbeitet. Damit sind auch die Variablen und ihre Werte in dieser Datei mit den Konfigurationsinformationen verfügbar.

Also enthält die Konfigurationsbeschreibung alle Schlüssel und Werte, und diese Datei alle dazugehörigen Informationen. Ein Ausschnitt einer solchen Datei befindet sich im Anhang C.2.

Die alternativen Gruppen

Dieses Werkzeug soll auch während der Entwicklung von Betriebssystemen einsetzbar sein. In dieser Phase vollziehen sich meist mehrere Veränderungen, auch in der Konfigurierungsbeschreibung. Einen schnellen Überblick über die meist genutzten alternativen Gruppen soll diese Ausgabe bieten. Sie enthält zu jeder alternativen Gruppe deren Namen und alle auswählbaren Komponenten. Zu jeder Komponente sind deren Namen, der Name der Klasse, durch welche diese implementiert wird, und die dazugehörige Header-Datei enthalten.
Für EPOS bedeutet dies die Ausgabe aller EPOS-Schnittstellen und deren jeweiligen Realisierungen. Ein Beispiel hierfür ist im Anhang C.3 aufgeführt.

5.3.3 Weitere Schnittstellen

Um die Konfigurierung eines Systems nur durch die Nutzung dieses Werkzeuges zu ermöglichen, muß der Aufruf weiterer Werkzeuge aus diesem möglich sein. Welche Werzeuge das sind, hängt natürlich von dem zu konfigurierenden Betriebssystem ab.
Für EPOS ist zum Beispiel der Aufruf des Werkzeuges wichtig, welches eine Anwendung analysiert und dann einen vorläufigen Bauplan des notwendigen Systems liefert. Dieser kann dann durch das Werkzeug graphisch dargestellt und durch den Nutzer verfeinert werden.

Da die fertige Konfiguration in Form von Präprozessoranweisungen ausgegeben wird, ist der Aufruf des Werkzeuges `make` sinnvoll. Dieses kann mit Hilfe der Konfigurationsbeschreibung, den weiteren Konfigurationsinformationen (siehe 5.3.2) und dem Quellcode das Betriebssystem zusammenschneidern und liefert das "fertige", ausführbare System.

5.4 Die graphische Oberfläche

Dem Programm wird durch die XML-Eingabedatei die Konfigurierungsbeschreibung bereitgestellt. Diese soll graphisch dargestellt werden, um dem Nutzer eine einfache und übersichtliche Konfigurierung zu ermöglichen.

5.4.1 Die Oberflächenobjekte zur Darstellung der Komponenten und Variablen

Als erstes wird hier die Auswahl der graphischen Elemente, zur Darstellung der einzelnen Informationen, beschrieben. Diese Informationen bestehen aus Komponenten beziehungsweise Komponentengruppen und Variablen, und den dazu gehörigen Informationen. Die "erweiterten" Konfigurationsregeln, also alle außer die alternativen Gruppen, werden nicht graphisch dargestellt sondern intern gehalten, wie in 5.5 beschrieben.

Alle Komponenten gehören einer Gruppe an, somit werden sie auch nur innerhalb von Gruppen dargestellt. Da die Anzahl der Komponenten pro Gruppe meist 6 nicht übersteigt werden die Gruppen nicht durch Listen, sondern durch Radio- bzw. Checkbuttons dargestellt.
Radiobuttons sind Buttons, von denen der Nutzer immer nur einen markieren kann. Dabei ist ein Button bereits ausgewählt. Sie repräsentieren eine exklusive Auswahl und eignen sich somit zur Darstellung alternativer Gruppen. Die Komponenten in einer optionalen Gruppe können durch *Checkbuttons* repräsentiert werden. Dies sind Buttons, von denen der Nutzer einen oder mehrere auswählen kann. Damit implementieren sie eine nicht exklusive Auswahl. Sie werden "angekreuzt".
Eine Gruppe von Komponenten soll deutlich von anderen graphischen Elementen abgegrenzt sein, und den Gruppennamen enthalten. Die einzelnen Buttons einer Gruppe werden mit den Namen der jeweiligen Komponenten beschriftet. Das Aussehen der Radio- und Checkbuttons ist abhängig von der Systemplattform. Unter Windows 98 wird eine alternative und eine optionale Gruppe wie im Bild 5.4.1 dargestellt.

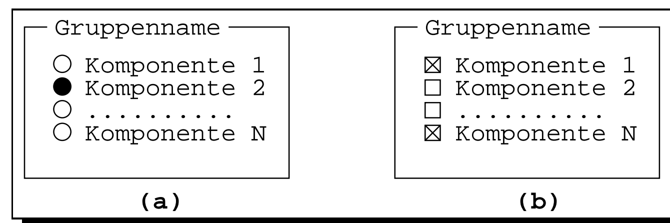


Bild 5.4.1: Radiobuttons (a) und Checkbuttons (b)

Qualitative Einstellungen erfolgen durch die Interaktion mit diesen graphischen Elementen. Die Veränderung quantitativer Eigenschaften des Systems wird durch einzeilige Texteingabefelder ermöglicht. Diese haben eine voreingestellte Länge, welche aber durch die Angaben des Attributes `length` in der Eingabedatei verändert werden kann. Ein solches Elemente wird im Bild 5.4.2 dargestellt.

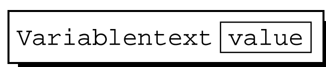


Bild 5.4.2: Ein Eingabefeld

Der Variablentext entspricht hier der Zeichenkette des Attributes `text` des `<input>`-Tags in der Eingabedatei. Der aktuelle Wert (*value*) der Variablen kann entweder ein String (`type=''text''`) oder eine ganze Zahl (`type=''number''`) sein. Nach der Eingabe eines Wertes wird dessen Typ und Gültigkeit überprüft. Eine Zahleneingabe ist gültig, falls die eingegebene Zeichenkette wirklich einer Zahl entspricht, und diese Element des in der Eingabedatei festgelegten Wertebereiches (`range`) ist. Tritt hierbei ein Fehler auf, wird dieser dem Nutzer durch eine entsprechende Meldung mitgeteilt.
Der Wert einer Variable vom Typ Zeichenkette ist immer gültig.

Damit können alle Komponenten und Variablen graphisch dargestellt werden. Im folgenden wird geklärt, wie diese graphischen Elemente in der Gesamtoberfläche organisiert werden, und welche weitere graphischen Elemente existieren.

5.4.2 Das Oberflächenmodell

Eine "gute" Organisation der Oberfläche ist wesentlich für die effiziente Anwendbarkeit eines graphischen Werkzeuges. Dies bedeutet, daß der Nutzer in dieser Oberfläche, bzw. mit deren graphischen Elementen, intuitiv interagieren kann. Dazu ist es notwendig einfache graphische Elemente zu benutzen, und den Nutzer nicht durch neue Elemente zu "überraschen". Weiterhin wird sich an Vorhandenem orientiert, und auf die Einhaltung der Regeln zur Gestaltung graphischer Oberflächen geachtet.

Die graphische Oberfläche wird in einem Fenster angelegt. In diesem Fenster befindet sich unterhalb des oberen Randes ein Menü zur Interaktion mit den Schnittstellen. Dazu gehört das Speichern und Laden von Konfigurationen und das Speichern der in 5.3.2 beschriebenen Informationen. Weiterhin können über das Menü externe Programme aufgerufen werden.
Das restliche Fenster ist vertikal in zwei Bereiche geteilt. Im linken Bereich erfolgt die Navigation durch die Informationen, und im rechten Teil werden die Informationen angezeigt. Hierbei wird sich an bereits vorhandenen Werkzeugen, wie dem Microsoft Explorer, orientiert.
Diese Aufteilung der Oberfläche wird im Bild 5.4.3 dargestellt.

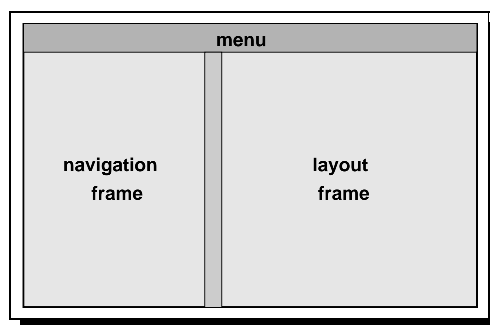


Bild 5.4.3: Die Oberflächenaufteilung

Die schon in der Eingabedatei vorhandenen Strukturierung der Konfigurierung in Teilbereiche wird in der graphischen Oberfläche in Form eines Baumes dargestellt. Ein Baum ermöglicht ein schnelles Navigieren durch die Konfigurierungsinformationen und bietet eine gute Übersicht. In diesem Baum werden die Domänen und Sektionen als Knoten dargestellt. Hinter diesen Knoten befinden sich die zur Konfigurierung wichtigen Informationen. Damit wird durch die Nutzung eines Baumes die Komplexität der graphischen Darstellung reduziert.
In diesem Baum kann durch die verschiedenen Domänen und Sektionen der Konfigurierungsbeschreibung schnell navigiert werden. Durch die Auswahl eines Knotens im Baum wird dessen Inhalt, also die ihm zugehörigen Komponentengruppen und Variablen, in Form von Radio- und Checkbuttons bzw. Eingabefeldern, im rechten Teil des Fensters angezeigt.

Die Informationen zu den Komponenten, Gruppen, Variablen, Sektionen bzw. Domänen werden in einem separaten Fenster bzw. Dialog angezeigt. Dieser Dialog kann beispielsweise durch das Anklicken des jeweiligen graphischen Elements mit der rechten Maustaste aufgerufen werden. Ist die Information eine URL, sollte das sich darunter befindende Dokument angezeigt werden.

5.5 Das interne Verhalten - die Konfigurationsregeln

Wie schon erwähnt, bestehen zwischen den Komponenten beziehungsweise zwischen Komponenten und Variablen bestimmte Abhängigkeiten. Die Abhängigkeiten zwischen Komponenten können oft durch deren Gruppierung ausgedrückt werden (alternativ, optional), und die Einschränkungen der Variablen durch die Angabe von Wertebereichen.
Es bestehen aber auch Abhängigkeiten zwischen Komponenten, welche sich nicht innerhalb einer Gruppe befinden, oder auch zwischen Komponenten und Variablen. So ist zum Beispiel für ein System, welches nur aus einem Prozess besteht, die Auswahl der Anzahl der Prozessoren unnötig.

5.5.1 Die Wissensbasis

Die Abhängigkeiten werden in der XML-Eingabedatei durch die Attribute `pre` und `pos` der jeweiligen Tags ausgedrückt. Beim Einlesen der Konfigurierungsbeschreibung ist eine Konvertierung der Nachbedingungen (`pos`) in Prologregeln naheliegend. Prolog ist die meist verbreitete und älteste Sprache zur Darstellung von strukturiertem Wissen. Deshalb bietet sie eine riesige Menge von Möglichkeiten zur Umsetzung von Expertensystemen.
Zu diesen Prologregeln kämen dann noch weitere, welche aus den Fakten, also aus der vorgegebenen (Default-)Konfiguration gebildet werden. Dies sind die Variablen und deren Werte, die alternativen Gruppen und die jeweils ausgewählten Komponenten, und die Komponenten der optionalen Gruppen nebst ihrer Werte (true, false). Diese Menge der Regeln wird die Wissensbasis bilden.

5.5.2 Testen der Vorbedingung

Ist die Vorbedingung einer Komponente, Gruppe, Sektion oder Variable nicht erfüllt, so soll diese nicht veränderbar oder auswählbar sein. Damit ist deren graphische Darstellung eigentlich unnötig.
Die Vorbedingung eines jeden Elements wird vor dessen Darstellung überprüft werden, indem sie als Anfrage an die Wissensbasis gestellt wird. Ist diese gültig, kann das Element dargestellt werden. Andererseits wird es deaktiviert dargestellt, um dem Nutzer dennoch zu zeigen, daß dieses Element existiert.

5.5.3 Testen der Nachbedingung

Die Gültigkeit der Nachbedingung wird nach jeder Änderung in einer Gruppe, Selektion oder Deselektion einer Komponente, bzw. Änderung eines Wer-

tes[1] getestet werden. Dazu ist es notwendig den vorher gesetzte Wert zu sichern, und dann in der Wissensbasis durch den neuen Wert zu überschreiben. Danach kann Prolog testen, ob noch alle Regeln in der Wissenbasis erfüllt sind. Ist dies nicht der Fall, sollen die Fehlermeldungen (die verletzten Regeln) an den Nutzer ausgegeben werden, und er wird entscheiden können, ob versucht werden soll, die Gültigkeit automatisch herzustellen. Falls ja, soll die Abarbeitung der Fehlermeldungen so erfolgen, daß die jeweiligen Werte so gesetzt werden, daß die Abhängigkeiten erfüllt sind, und der Fehler verschwindet. Dabei wird für jeden versuchsweise neu gesetzten Wert wiederum die Validität der Wissenbasis bezüglich der Regeln getestet werden müssen. Wünscht der Nutzer keine automatische "Korrektur" der Werte ist die Änderung ungültig und der alte Wert wird wieder gesetzt.

5.6 Die Programmiersprache

Nun gilt es für das bisher entworfene Werkzeug die Programmiersprache zu finden, in welcher es implementiert werden soll.
Wie in 5.1.3 erwähnt, soll der Einsatz des Werkzeuges plattformunabhängig sein. Dieser Aspekt wurde schon bei der Auswahl der Eingabesprache (XML) beachtet und wird auch bei der Auswahl der Programmiersprache wichtig sein.
Zum Einlesen der Eingabedatei wird ein XML-Parser benötigt. Dieser sollte in das Programm leicht integrierbar sein. So kann er zum Beispiel in Form einer Bibliothek vorliegen. Dieser validierende Parser muß den Zugriff auf das Document Object Model (siehe 4.2.4) ermöglichen.
Aus der Eingabedatei wird eine graphische Oberfläche erzeugt. Daher ist es von großem Vorteil, wenn die Programmiersprache eine einfache Erzeugung graphischer Elemente (z.B. Fenter, Menüs, Buttons, ...) unterstützt. Desweiteren muß die Sprache den Aufruf externer Programme ermöglichen. Dazu gehört auch eine Prolog-Schnittstelle, welche zum Beispiel über Pipes realisiert werden könnte.
Da das Werkzeug auch für die Konfigurierung anderer Betriebssysteme als EPOS genutzt werden soll, und dazu eventuell kleine Änderungen oder Erweiterungen notwendig sind, wird es objektorientiert programmiert werden.

Die Wahl der Programmiersprache fiel auf JAVA. JAVA gehört zu der Gruppe von Sprachen, bei denen der Übersetzer kein ausführbares Programm für den Zielprozessor erzeugt, sondern ein Programm für eine virtuelle Maschine. Die Instruktionen der virtuellen Maschine werden Bytecode genannt. Zur Ausführung eines solchen Programms wird ein Bytecode-Interpreter

[1] Da die Behandlung einer Komponente und eines Wertes hier gleich ist, werden beide mit Wert bezeichnet.

benötigt, der die virtuelle Maschine emuliert. Der Einsatz von Bytecode hat einige Vorteile, die vor allem im Bereich Portabilität, Programmgröße und Sicherheit liegen, aber auch den Nachteil der geringeren Geschwindigkeit. Die Emulation der virtuellen Maschine verlängert die Laufzeit eines Programms gegenüber der Ausführung des entsprechenden Maschinenprogramms um das 2 bis 50-fache.
Viele der heutigen Bytecode-Interpreter für JAVA setzen deshalb bereits eine Technik ein, die sich Just-In-Time Übersetzung nennt. Dabei wird ein Stück JAVA Bytecode, z.B. eine Methode, wenn es gebraucht wird, in Maschinencode übersetzt.

JAVA bietet die Möglichkeit einer schnellen Softwareentwicklung, durch die Vielzahl der verfügbaren Packages[2]. So existiert ein Paket, welches die Darstellung von graphischen Oberflächen mit all seinen Elementen unterstützt. Weiterhin sind für JAVA verschiedene XML-Parser frei erhältlich.
Mittels der JAVA-Datenstreams ist eine einfache Kommunikation mit externen Programmen, z.B. Prolog, möglich.
Der Nachteil der Laufzeit eines Programms ist natürlich nicht von der Hand zu weisen. Da dieses Werkzeug jedoch einen größeren Wert auf die Plattformunabhängigkeit als auf die Laufzeit legt, ist dies kein wesentlicher Nachteil.

[2] zu deutsch: Pakete

Kapitel 6

Implementation

In diesem Kapitel werden technische Aspekte des fertig implementierten Werkzeuges beschrieben. Dazu gehören Fakten wie dessen Installation und Erweiterung.
Da das Werkzeug nicht ausschließlich für die Konfigurierung des Betriebssystems EPOS entworfen und implementiert wurde, sondern auch für andere Systeme anwendbar sein soll, ist die Wahl des Namens auf XCONF gefallen, und nicht etwa auf EPOSCONF.

Der XCONF-Quellcode und die extra benötigten JAVA-Pakete, die im folgenden auch kurz beschrieben werden, befinden sich auf der beiliegenden CD. Diese enthält weiterhin die XCONF-Dokumentation.

6.1 Systemvoraussetzungen

Durch gewisse Entscheidungen im Entwurf werden bestimmte Anforderungen an das System, auf welchem XCONF ausgeführt werden soll, impliziert. Genauer gesagt, bedingen sie das Vorhandensein bestimmter Software.
Die Programme bzw. Werkzeuge, die aus XCONF heraus aufgerufen werden können, sollten sich auf dem Laufzeitsystem befindet. Dazu zählt auch Prolog, welches zum Überprüfen der Konfigurationsregeln benötigt wird[1].
Da XCONF in JAVA implementiert ist, kann es, nachdem es in den plattformunabhängigen Bytecode übersetzt wurde, auf allen Plattformen ausgeführt werden, auf welchen eine *Java Virtual Machine* installiert ist. Weiterhin müssen neben denen zum Java-Standard gehörenden Paketen[2] zwei weitere installiert sein, die im Folgenden kurz beschrieben werden.

[1]Da die Prologschnittstelle, und damit die Überprüfung der Konfigurationsregeln noch nicht implementiert ist, ist Prolog noch keine Systemvoraussetzung.

[2]Pakete sind die Java-Bibliotheken. So existiert beispielsweise ein Paket *AWT (Abstract Windowing Toolkit)* welches Methoden zur Darstellung visueller Elemente bereitstellt.

6.1.1 Neue visuelle Elemente

Die Struktur der Konfigurierungsbeschreibung wird in Form eines Baumes dargestellt. Da dieses visuelle Element in den JAVA Standardpaketen nicht vorhanden ist, muß ein extra Paket installiert sein, das JAVA-Swing-Paket. Das *Swing-Projekt* ist Teil der JFC[3]-Software, die das Standardpaket AWT durch eine umfassende Menge neuer visueller Komponenten erweitert. Diese zeichnen sich durch ein "pluggable look and feel" aus, wodurch die graphische Nutzerschnittstelle automatisch das Aussehen und Verhalten der aktuellen Betriebssystemplattform (z.B. MS Windows, Solaris, Linux) besitzt.
Ab JAVA 2 ist das Swing-Paket standardmäßig enthalten. Für ältere Javaversionen, ist das Swing-Paket im Internet unter "`http://java.sun.com`" frei erhältlich. Die Datei `swingall.jar` enthält das komplette Paket.

6.1.2 Der Parser

Zum Einlesen und Validieren der Eingabedatei wird die Bibliothek *Java Project X* von *Sun Microsystems Inc.* in der Version *Technology Release 2* benötigt. Diese ist im Internet unter "`http://java.sun.com/xml`" erhältlich. Das komplette Paket ist im Archiv `xml.jar` enthalten.

6.1.3 Starten und Übersetzen von XCONF

Sowohl zum Starten und zum Übersetzen müssen die Standardpakete von Java, das Swing-Paket und der Parser installiert sein. Wo sich diese Archive oder Pakete befinden muß dem Compiler bzw. der JVM mitgeteilt werden. Dazu wurde im *Makefile* eine Variable namens `CPATH` wie folgt definiert:

```
CPATH=.:xml.jar:swingall.jar:/lib/classes.zip
```

Diese Definition ist gültig, falls sich das Parser-Paket und das Swing-Paket im aktuellen Pfad befinden.
Der allgemeine Befehl zum Übersetzen von XCONF lautet:

```
JavaCompiler -classpath ``$(CPATH)'' XConf.java
```

JavaCompiler gibt an, wo sich der Java-Compiler befindet, zum Beispiel `/usr/local/jdk1.2/bin/javac`. XCONF wurde bisher mit dem Compiler der JAVA-Version 1.1.8 übersetzt. Für höhere Compilerversionen sollte der Übersetzungsvorgang ohne Probleme ablaufen.
Gestartet wird XCONF mit folgendem Befehl:

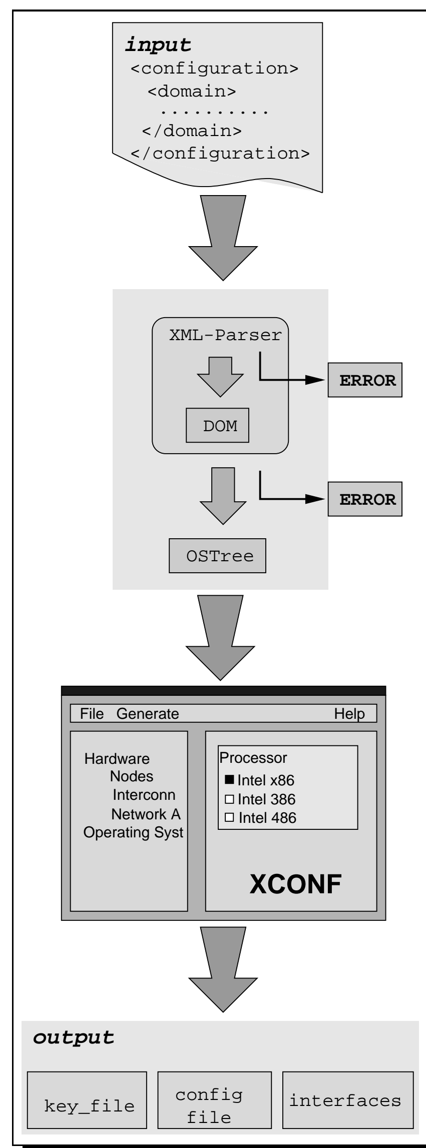
```
JVM -classpath ``$(CPATH)'' XConf xml-file
```

[3] Java^{TM} Foundation Classes

Dabei gibt *JVM* an, wo sich die Java Virtual Machine befindet, beispielsweise `/usr/local/jdk1.2/bin/java`. Die älteste JAVA-Version, unter welcher XCONF gestartet wurde, ist 1.1.3.
Als Parameter *xml-file* wird XCONF der Name der Datei übergeben, welche die Konfigurierungsbeschreibung für das zu konfigurierende System enthält.

6.2 Der Ablauf

Nachdem XCONF wie in 6.1.3 beschrieben gestartet wurde, wird die XML-Datei an den eingebundenen XML-Parser übergeben, der diese liest, validiert, und falls sie den Einschränkungen der DTD genügt, ein *Document Object Model* (DOM, siehe 4.2.4) bereitstellt. Anderenfalls werden die Fehlermeldungen ausgegeben und das Programm beendet.
Dieses DOM wird nun durchlaufen und in eine eigene Datenstruktur, den `OSTree` (Operating System Tree) übertragen. Dabei wird getestet, ob zum Beispiel die Werte der Variablen innerhalb ihrer Grenzen liegen oder sich der Vorgabewert einer alternativen Gruppe auch wirklich in dieser Gruppe befindet. Treten hierbei Fehler auf, ist die Konfigurierungsbeschreibung falsch und das Programm wird beendet.
Andernfalls wird die graphische Oberfläche dargestellt. Durch die Interaktion mit den visuellen Elementen wird der Prozess der Konfigurierung absolviert. Die Elemente werden wie gewohnt bedient.
Zum Anzeigen der Hilfe zu einer Komponente, Gruppe, Variable, Sektion oder Domäne kann diese mit der rechten Maustaste angeklickt werden.



Beispiele für die graphische Oberfläche bei der EPOS-Konfigurierung sind im Anhang B zu sehen.

Über den Menüpunkt `Generate` können die externen Werkzeuge aufgerufen werden. Bisher sind leider noch keine Werkzeuge referenziert. Das Speichern oder Laden von Konfigurationen und das Speichern der anderer Ausgabedateien erfolgt unter dem Menüpunkt `File`. Es können die drei in 5.3.2 beschriebenen Dateiformate gespeichert werden.

6.3 XCONF für andere Betriebssysteme nutzen

Um XCONF auch zur Konfigurierung anderer Betriebssysteme außer EPOS zu nutzen, müssen eventuell ein paar Erweiterungen oder Änderungen in der Grammatik der Eingabesprache und im Programm selber vorgenommen werden. Die Erweiterungen lassen sich grob in die zwei Bereiche 6.3.1 und 6.3.2 unterteilen.

6.3.1 Neue externe Werkzeuge

Die Schnittstellen zu den externen Werzeugen werden in der Datei `XConf.java` definiert. Um ein neues Werzeug einzufügen, muß vorerst das Menü um diesen Punkt erweitert werden, und dann eine neue Methode eingefügt werden, welche bei der Auswahl des Menüpunktes ausgeführt werden soll. In dieser Methode kann dann die Interaktion mit dem externen Werkzeug erfolgen. Dabei kann sich an den schon vorhandenen Aufrufen orientiert werden.

6.3.2 Neue Systeminformationen

Zum Hinzufügen neuer Informationstypen über das zu konfigurierende System müssen zuerst neue Tags in der Eingabesprache definiert werden.
Nachdem die neuen Tagnamen in der Eingabesprache existieren kann XCONF ohne weiteres gestartet werden. Die neuen Informationen werden zwar schon vom XML-Parser überprüft und mit in das DOM eingetragen, aber noch nicht mit in die Datenstruktur `OSTree` übernommen. Diese wird durch die Methoden der Klasse `JScanner` aufgebaut. Dort wird das DOM durchlaufen und der `OSTree` durch die den Tagnamen entsprechenden Objekte erweitert. Hier müssen neue Methoden eingeführt werden, die die entsprechenden Objekte für die neuen Informationstypen in den `OSTree` einfügen. Zur Zeit existieren die folgenden fünf Klassen zur Verwaltung der Systeminformationen:

- `OptionSet` enthält alle Informationen zu einer Komponente.
- `SingleSelect` repräsentiert eine alternative Gruppe.

- `MultiSelect` stellt eine optionale Gruppe dar.
- `IntInput` dient der Verwaltung von Informationen über eine Variable mit numerischem Wert.
- `StringInput` ist eine Variable mit einem String als Wert.

Um einen neuen Informationstypen in den `OSTree` aufnehmen zu können muß also eine neue Klasse programmiert werden, welche diese Informationen repräsentiert und auswertet. Diese muß von der Klasse `VirDataSet` abgeleitet sein, und somit Methoden zum zeichnen, setzen und abfragen des Wertes besitzen. Von dieser Klasse wird dann ein Objekt angelegt und in den `OSTree` eingefügt.
Soll das Verhalten der aktuellen visuellen Elemente verändert oder erweitert werden, sind die Veränderungen in den fünf oben genannten Klassen vorzunehmen.

Für weitere Informationen existiert eine kurze Beschreibung der Datenstruktur und weitere Informationen zum Quellcode von XConf auf der beiliegenden CD.

Kapitel 7

Zusammenfassung

7.1 Diskussion

Das im Rahmen dieser Studienarbeit entworfene und implementierte graphische Werkzeug XConf dient der Konfigurierung von Software, im Besonderen von Betriebssystemen. Die Entwicklung des Werkzeuges erfolgte im Projekt Epos. Das Ziel dieses Projektes ist die Erstellung von anwendungsorientierten Betriebssystemen. Dazu werden Anwendungen analysiert, was eine vorläufige Konfiguration liefert. Es fehlen darin aber Informationen über die Zielplattform, wie beispielsweise die Anzahl der Prozessoren. Zur Eingabe solcher Systemparameter ist die Interaktion mit dem Nutzers notwendig. Genau an dieser Stelle setzt XConf an. Es stellt die vorläufigen Konfiguration durch geeignete visuelle Elemente graphisch dar. Der Nutzer kann dann die Konfiguration durch die Interaktion mit der Oberfläche verfeinern beziehungsweise vervollständigen.

XConf ist so entworfen, daß es nicht nur zur Konfigurierung von Epos eingesetzt werden kann, sondern auch für andere Betriebssysteme oder Software. Ein erster Schritt in diese Richtung war die Auswahl der Sprache Xml , in welcher die Konfigurierungsbeschreibung ausgedrückt wird. Sie ist eine Metasprache, welche die einfache strukturierte Angabe von Daten ermöglicht. Ein weiterer wichtiger Aspekt dieser Sprache ist, daß sie ein Standard ist, und ihre Verbreitung stetig steigt.
Da die Konfigurierungsbeschreibung für das jeweilige Betriebssystem durch XConf erst eingelesen wird, ist es zur Konfigurierung vieler Systeme anwendbar.

Die Konfigurierungsinformationen werden graphisch dargestellt. Damit ist eine relativ übersichtliche Konfigurierung garantiert, da nicht irgendwelche Parameter per Hand in Dateien geändert, und dabei alle Abhängigkeiten zwischen den Parametern vom Nutzer beachtet werden müssen. Durch die

graphische Darstellung wird somit eine Übersichtlichkeit über die Parameter und den Typ der Parameter erreicht. Die Einhaltung der Abhängigkeiten, welche zwischen den Komponenten eines Betriebssystems bestehen, wird bisher leider nur zu einem kleinen Teil berücksichtigt, da es aus Zeitgründen nicht mehr zur Implementierung der *RulesEngine* kam. Diese sollte die Einhaltung aller Regeln zur Laufzeit des Programms überprüfen, und dem Nutzer damit ein sehr großes Problem der Konfigurierung abnehmen. Zur Zeit werden nur die Regeln beachtet, die durch die Gruppierung der Komponenten in der Konfigurierungsbeschreibung indirekt angegeben werden. Dies bedeutet, daß in einer *alternativen Gruppe* immer nur eine Komponente selektiert sein darf und muß. Damit impliziert die Auswahl einer solchen Komponenten das Verbot der Auswahl einer weiteren Komponente aus derselben Gruppe.

Die fertige Konfiguration kann dann in Form von Präprozessordirektiven in einer Datei abgespeichert werden, welche dann zur Übersetzung des Systems mit eingebunden wird. Dieser Übersetzungsvorgang kann auch direkt aus dem Werkzeug heraus gestartet werden, womit XConf in der Lage ist ein "fertiges", also ausführbares Betriebssystem zu erstellen. Sind dazu noch weitere Werkzeuge notwendig, können diese ebenfalls aus XConf heraus gestartet werden. Dadurch bildet XConf eine graphische Schnittstelle zwischen anderen zur Konfigurierung notwendigen Werkzeugen, und sichert sich damit eine (mögliche) zentrale Rolle in der Konfigurierung von Softwaresystemen.

XConf ist in der Programmiersprache Java implementiert. Dadurch ist die Ausführbarkeit des Werkzeuges plattformunabhängig. Somit muß sich der Nutzer nicht viel mit der Anpassung von XConf an sein Laufzeitsystem beschäftigen, sondern kann XConf schneller zur Anpassung anderer Software nutzen. Durch objektorientierte Implementation ist es für eventuelle Erweiterungen offen. Diese können notwendig werden, wenn zur Konfigurierung anderer Systeme weitere Informationen bzw. Typen von Informationen benötigt werden.

7.2 Aussichten

In diesem letzten Abschnitt sollen einige Gedanken zur möglichen Weiterentwicklung von XConf bzw. ähnlicher Werkzeuge angeführt werden.
Ein nächster Schritt in der Entwicklung von XConf ist eindeutig die Implementierung der *RulesEngine*, welche die Einhaltung aller Regeln überprüft. Damit wäre dem Nutzer eines der größten Probleme während der Konfigurierung abgenommen.

Weiterhin kann XCONF noch so erweitert werden, daß die Konfigurierungsbeschreibung nicht mehr per Hand geschrieben und verändert werden muß, sondern im Werkzeug selber deren graphische Manipulation möglich ist.

Da XCONF in JAVA implementiert ist, kann es relativ schnell in eine *Netzversion* konvertiert werden. Diese Version von XCONF wäre dann keine Applikation mehr, sondern ein sogenanntes JAVA-Applet[1]. Damit wäre es für jeden möglich das Betriebssystem über das World Wide Web zu konfigurieren. Das fertige System würde dann auf dem Server erstellt, und entweder dem Nutzer zugesendet oder in einem Pfad abgelegt werden, welcher dem Nutzer dann mitgeteilt wird, und aus welchem er sich das System herunterladen kann. Er benötigt also weder den Quellcode des Systems noch das Konfigurierungsprogramm, denn all dies ist auf dem Server.
Damit könnte sich der Nutzer *sein* Betriebssystem im Internet zusammensetzen und dann herunterladen.

[1] Applets sind Java-Programme, die über das World Wide Web zum jeweiligen Client-Rechner heruntergeladen werden und dort von einem Java-fähigen Web-Browser ausgeführt werden.

Literaturverzeichnis

[CE] Krzyszof Czarnecki and Ulrich W. Eisenecker. *Synthesizing Objects.*

[Con98] World Wide Web Consortium. *XML 1.0 Recommendation.* www.w3c.org, February 1998.

[Eis97] Ulrich W. Eisenecker. Generative Programmierung und Komponenten. *OBJEKTspektrum*, 1997.

[GJ90] Jack Goldberg and Robert J.Stroud. Adaptive fault-tolerant systems and reflective architctures. Technical report, Department of Computer Science, Universitiy of Newcastle upon Tyne, UK, 1990.

[GP98] Charles F. Goldfarb and Paul Prescod. *The XML Handbook.* Prentice Hall PTR, 1998.

[Mic99] Dr. Thomas Michel. *XML kompakt.* Carl Hanser Verlag, 1999.

[Par79] David L. Parnas. Designing software for ease of extension and contraction. *IEEE Transaction on Software Engineering*, SE-5(2):128–138, March 1979.

[SSPSS98] Friedrich Schön, Wolfgang Schröder-Preikschat, Olaf Spinczyk, and Ute Spinczyk. Design Rationale of the PURE Object-Oriented Embedded Operating System. In *Proceedings of the International IFIP WG 10.3/WG 10.5 Workshop on Distributed and Parallel Embedded Systems*, Paderborn, Germany, October 1998.

Anhang A

Die Konfigurierungsbeschreibung

A.1 Die DTD-Grammatik

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!ENTITY % maindata
        'name    ID       #REQUIRED
        text     CDATA    #REQUIRED
        help     CDATA    #IMPLIED'>

<!ENTITY % rules
        'pre     CDATA    #IMPLIED
         pos     CDATA    #IMPLIED'>

<!ENTITY % extdata
        '%maindata;
         %rules;'>

<!ELEMENT configuration     (domain)+>
<!ATTLIST configuration
          name     CDATA    #REQUIRED
          text     CDATA    #REQUIRED
          config_file CDATA  #IMPLIED
          keys_file   CDATA  #IMPLIED>

<!ELEMENT domain            (section)+>
<!ATTLIST domain
          %maindata;>

<!ELEMENT section           (interface|multiselect|input|(section)*)+>
<!ATTLIST section
          %maindata;
          pre      CDATA    #IMPLIED>
```

```
<!-- *************** REALISATION *************** -->
<!ELEMENT realisation       EMPTY>
<!ATTLIST realisation
          %extdata;
          class    CDATA    #IMPLIED
          header   CDATA    #IMPLIED>

<!-- *************** INTERFACE *************** -->
<!ELEMENT interface         (realisation+,notneeded?)>
<!ATTLIST interface
          %extdata;
          default  IDREF    #REQUIRED
          class    CDATA    #IMPLIED>

<!-- *************** MULTISELECT *************** -->
<!ELEMENT multiselect       (realisation)+>
<!ATTLIST multiselect
          %extdata;
          default  CDATA    #IMPLIED>


<!-- *************** INPUT *************** -->
<!ELEMENT input             EMPTY>
<!ATTLIST input
          %extdata;
          default  CDATA    #REQUIRED
          type     NMTOKEN  #REQUIRED
          range    NMTOKEN  #IMPLIED
          length   NMTOKEN  #IMPLIED>

<!-- ***** NOT-NEEDED ***** -->
<!ELEMENT notneeded       EMPTY>
<!ATTLIST notneeded
          %rules;
          name     CDATA    #FIXED 'NOT_NEEDED'
          text     CDATA    #FIXED 'Not needed'
          help     CDATA    #IMPLIED
          class    CDATA    #IMPLIED
          header   CDATA    #IMPLIED>
```

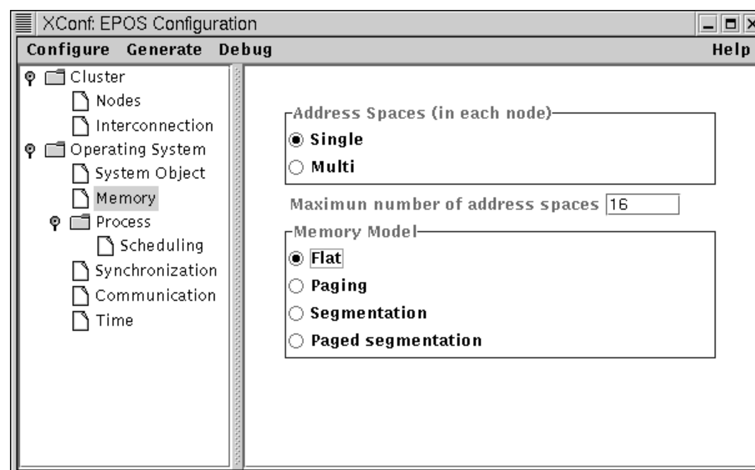
A.2 Ausschnitt der Epos-Konfigurierungsbeschreibung in Xml

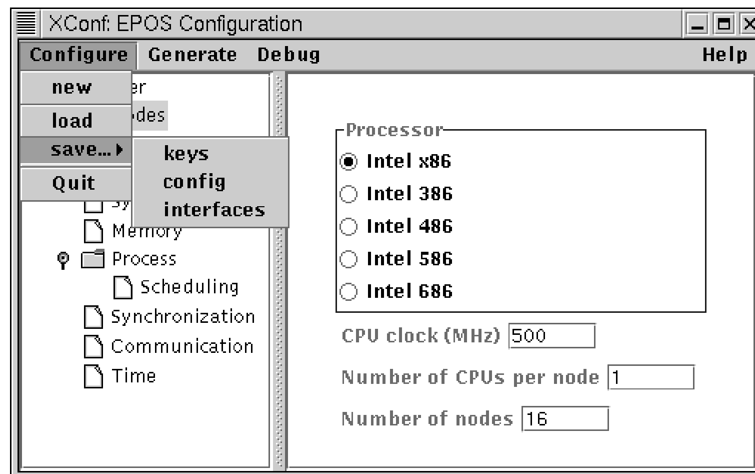
```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE configuration SYSTEM "config.dtd">

<configuration name="EPOS_CONFIGURATION"
  text="EPOS Configuration"
  config_file="system_config.h"
  keys_file="system_config_keys.h">
  ............
  <domain name="OS" text="Operating System">
    ............
    <section name="SECTION_MEMORY" text="Memory">
      <interface name="ADDRESS_SPACE" text="Address Spaces (in each node)"
        default="ADDRESS_SPACE_SINGLE">
        <realisation name="ADDRESS_SPACE_SINGLE" text="Single"
          pos="PROCESS_MODEL = PROCESS_SINGLE"/>
        <realisation name="ADDRESS_SPACE_MULTI" text="Multi"/>
      </interface>
      <input name="ADDRESS_SPACE_UNITS"
        text="Maximun number of address spaces"
        type="number" range="1.." default="16"
        pre="ADDRESS_SPACE == ADDRESS_SPACE_MULTI"/>
      <interface name="MEMORY_MODEL" text="Memory Model"
        default="MEMORY_FLAT">
        <realisation name="MEMORY_FLAT" text="Flat"
          pre="ADDRESS_SPACE == ADDRESS_SPACE_SINGLE"
          pos="PROCESS_MODEL = PROCESS_SINGLE"/>
        <realisation name="MEMORY_PAGING" text="Paging"/>
        <realisation name="MEMORY_SEGMENTATION" text="Segmentation"/>
        <realisation name="MEMORY_PAGED_SEGMENTATION"
          text="Paged segmentation"/>
      </interface>
    </section>
    ............
  </domain>
</configuration>
```

Der hier angegebene Ausschnitt der Epos-Konfigurierungsbeschreibung zeigt die Anwendung der Tags. Es beschreibt Einstellungsmöglichkeiten zum Arbeitsspeicher und Speichermodelle.
Durch die Anweisung `<!DOCTYPE configuration SYSTEM ''config.dtd''>` wird die zugehörige DTD eingebunden und das Root-Element des Dokuments bestimmt (`configuration`).
Ein komplette Epos-Konfigurierungsbeschreibung befindet sich auf der beiliegenden CD.

Anhang B

Screenshots





Anhang C

Die Ausgaben

C.1 Eine Konfigurationsbeschreibung für EPOS

```
#ifndef system_config_keys_h
#define system_config_keys_h 1

#define CONF_CPU                              CONF_CPU_IX86
#define CONF_CPU_CLOCK                        500
#define CONF_CPU_UNITS                        1
#define CONF_NODE_UNITS                       16
#define CONF_NETWORK                          CONF_NETWORK_FAST_ETHERNET
#define CONF_MYRINET_NETWORK_ADAPTER          CONF_NETWORK_ADAPTER_FAST_ETHERNET
#define CONF_NETWORK_ADAPTER                  CONF_NETWORK_ADAPTER_MYRINET
#define CONF_SYSTEM_OBJECT                    CONF_SYSTEM_OBJECT_BASIC
#define CONF_SYSTEM_OBJECT_ID                 CONF_LOCAL_SYSTEM_OBJECT_ID
#define CONF_ADDRESS_SPACE                    CONF_ADDRESS_SPACE_SINGLE
#define CONF_ADDRESS_SPACE_UNITS              16
#define CONF_MEMORY_MODEL                     CONF_MEMORY_FLAT
#define CONF_TASK                             CONF_TASK_SINGLE
#define CONF_TASK_UNITS                       16
#define CONF_THREAD                           CONF_THREAD_SINGLE
#define CONF_THREAD_UNITS                     16
#define CONF_SCHEDULER                        CONF_THREADSCHEDULER
#define CONF_TASK_SCHEDULER                   CONF_FCFS_TASK_SCHEDULER
#define CONF_RR_TASK_SCHEDULER_TIME_SLICE     100
#define CONF_PRIORITY_TASK_SCHEDULER_RANGE    16
#define CONF_THREAD_SCHEDULER                 CONF_FCFS_THREAD_SCHEDULER
#define CONF_RR_THREAD_SCHEDULER_TIME_SLICE   100
#define CONF_PRIORITY_THREAD_SCHEDULER_RANGE  16
#define CONF_SYNCHRONIZER                     CONF_SEMAPHORE_SYNCHRONIZER
#define CONF_COMMUNICATOR                     CONF_MAILBOX_COMMUNICATOR
#define CONF_TIMER                            CONF_SIMPLE_TIMER
#define CONF_CHRONOMETER                      CONF_NOT_NEEDED

#endif
```

C.2 Ausschnitt der EPOS-Konfigurationsinformationen

```
//=========================================================================
// SYSTEM_CONFIG.H
//
// created by XConf
//
// DATE: Mon Feb 21 19:02:12 CEST 2000
//=========================================================================
#ifndef __system_config_h
#define __system_config_h 1

//=========================================================================
// CONFIGURATION KEYS
//=========================================================================
#define CONF_NOT_NEEDED                      -1

.........
#define CONF_TASK_SINGLE                      1
#define CONF_TASK_MULTI                       2
........
#define CONF_FCFS_TASK_SCHEDULER              1
#define CONF_RR_TASK_SCHEDULER                2
#define CONF_PRIORITY_TASK_SCHEDULER          3

............

//=========================================================================
// APPLICATION SPECIFIC CONFIGURATION KEYS
//=========================================================================
#include "system_config_keys.h"
........

//=========================================================================
// TASK
//=========================================================================
#if      CONF_TASK              == CONF_TASK_SINGLE
#define  CURRENT_TASK_H         <system/task/single.h>
typedef  class Single_Task      Current_Task;

#elif    CONF_TASK              == CONF_TASK_MULTI
#define  CURRENT_TASK_H         <system/task/multi.h>
typedef  class Multi_Task       Current_Task;

#else
#error Configuration error: "TASK" not selected!
#endif

//=========================================================================
// TASK_SCHEDULER
//=========================================================================
#if      CONF_TASK_SCHEDULER            == CONF_FCFS_TASK_SCHEDULER
```

```
#define  CURRENT_TASK_SCHEDULER_H       <system/task/fcfs.h>
typedef  class FCFS_Task_Scheduler      Current_Task_Scheduler;

#elif    CONF_TASK_SCHEDULER            == CONF_RR_TASK_SCHEDULER
#define  CURRENT_TASK_SCHEDULER_H       <system/task/rr.h>
typedef  class RR_Task_Scheduler        Current_Task_Scheduler;

#elif    CONF_TASK_SCHEDULER            == CONF_PRIORITY_TASK_SCHEDULER
#define  CURRENT_TASK_SCHEDULER_H       <system/task/priority.h>
typedef  class Priority_Task_Scheduler  Current_Task_Scheduler;

#else
#error Configuration error: "TASK_SCHEDULER" not selected!
#endif

.........

#endif // __system_config_h
```

Die komplette Datei mit den Konfigurationsinformationen ist auf der beiliegenden CD zu finden.

C.3 Die alternativen Gruppen

```
interface NETWORK
        NOT_NEEDED:Dummy_Network:system/dummy.h
        NETWORK_ETHERNET:Ethernet:system/network/ethernet.h
        NETWORK_FAST_ETHERNET:Fast_Ethernet:system/network/fast_ethernet.h
        NETWORK_GIGA_ETHERNET:Giga_Ethernet:system/network/giga_ethernet.h
        NETWORK_MYRINET:Myrinet:system/network/myrinet.h
        NETWORK_SCI:SCI:system/network/sci.h
........

interface TASK
        TASK_SINGLE:Single_Task:system/task/single.h
        TASK_MULTI:Multi_Task:system/task/multi.h
........

interface COMMUNICATOR
        NOT_NEEDED:Dummy_Communicator:system/dummy.h
        LINK_COMMUNICATOR:Link:system/communicator/link.h
        PORT_COMMUNICATOR:Port:system/communicator/port.h
        MAILBOX_COMMUNICATOR:Mailbox:system/communicator/mailbox.h
........
```

Die komplette Datei ist auf der beiliegenden CD zu finden.