Einbindung von Geräten in Internet-basierte Informationssysteme mit Java und *XML*

Dissertation
der Fakultät für Informatik
der Eberhard-Karls-Universität Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften (Dr. rer. nat.)

vorgelegt von

Dipl.-Inform. Dieter Bühler
aus Tübingen

Tübingen
2002

Tag der mündlichen Qualifikation: 06.02.2002
Dekan: Prof. Dr. A. Zell
1. Berichterstatter: Prof. Dr. sc. techn. W. Küchlin
2. Berichterstatter: Prof. Dr. A. Zell

Für Andrea, Erika und Karl

Vorwort

Die vorliegende Arbeit entstand während meiner Tätigkeit als wissenschaftlicher Mitarbeiter am Wilhelm-Schickard-Institut für Informatik an der Universität Tübingen.

Meinen besonderen Dank möchte ich Herrn Prof. Wolfgang Küchlin für die Ermöglichung und Betreuung dieser Arbeit aussprechen. Ebenso bedanken möchte ich mich bei Herrn Prof. Gerhard Gruhler, der mir den Zugang zum Roboterlabor der Fachhochschule Reutlingen ermöglicht hat und mir ebenfalls wichtige Hinweise für die Erstellung dieser Arbeit geben konnte. Desweiteren danke ich Herrn Prof. Andreas Zell für die Erstellung des zweiten Gutachtens.

Für die ausgezeichnete Zusammenarbeit mit Siemens Medical Solutions in Erlangen möchte ich mich bei Don Medlar, Georg Görtler und Wolfgang Kalnischkies bedanken.

Ich bedanke mich auch bei den Kollegen am Wilhelm-Schickard-Institut, die mir in einer Vielzahl von Gesprächen und Diskussionen zur Seite standen. In diesem Kontext seien insbesondere Gerd Nusser und Ralf-Dieter Schimkat genannt. Michael Großmann, Elias Volanakis und Wolfgang Westje waren mir eine wichtige Hilfe bei der Realisierung der *Mathe-Tools* Umgebung.

Tübingen, im Februar 2002

Dieter Bühler

Inhaltsverzeichnis

Abbildungen

Tabellen

Listings

Kapitel 1

Einleitung

Thema der vorliegenden Arbeit ist die Entwicklung und Implementierung von offenen Konzepten zur durchgängigen Erfassung, Verwaltung und Visualisierung von gerätebezogenen Informationen. Durchgängig meint hier, dass auf die Gerätedaten über unterschiedliche Netzwerke und Protokolle für den Benutzer transparent zugegriffen wird und so eine Integration der Gerätedaten in moderne Informationsinfrastrukturen (z.B. Datenbanksysteme, Wartungs-, Service- und Evaluationswerkzeuge usw.) erreicht werden kann. Ermöglicht wird diese Integration durch die zunehmende Vernetzung von Geräten (z.B. industrielle Feldbusgeräte oder Büro-Peripherie) und die Verbreitung von offenen, plattformunabhängigen Kommunikationskonzepten.

Der *Mobile* bzw. *Pervasive Computing* Ansatz [FZ94, HLP99] befasst sich beispielsweise gezielt mit der Durchdringung des täglichen Lebens durch vernetzte Geräte und der Konsolidierung entsprechender Datenprotokolle. Geräte der unterschiedlichsten Art verfügen mittlerweile über Online-Zugang zu einer Vielfalt heterogener Informationen von unterschiedlichsten Datenquellen. Das Internet spielt hierbei eine zentrale Rolle nicht nur als Übertragungs- und Kommunikationsmedium, sondern auch als Integrationsmedium.

Für einen durchgängigen Datenaustausch muss die Beziehung zwischen den Informationen an sich und den rechnerbezogenen Repräsentationen der Informationen klar definiert sein. Um jedoch nicht für n Datenrepräsentationen von n Informationsquellen $\mathbb{O}(n^2)$ Transformationen zur Übersetzung der Datenformate angeben zu müssen, empfiehlt es sich, nach abstrakten Zwischenformaten zur systemunabhängigen Informationsrepräsentation zu suchen.

Vor diesem Hintergrund wurde Anfang 1998 die *eXtensible Markup Language* (*XML*) durch das *World Wide Web Consortium* (www.w3c.org) definiert. Sie ist eine Metasprache zur Spezifikation von spezialisierten Sprachen für bestimmte Anwendungsgebiete. *XML* erlaubt die Plattform und sprachunabhängige Repräsentation von strukturierter Information in textueller Form und die flexible Vernetzung von Information auf dem Internet. Der Einzug von *XML*-basierten Technologien in die unterschiedlichsten Bereiche der Informationstechnologie hat eine große Zahl von *XML*-basierten Sprachstandardisierungen z.B. in den Bereichen Mathematik (MathML [Wor01b]), Computergraphik (VML [MLDea98]), Robotik (RoboML [MT00]), Mobile Computing (SyncML [Syn00]) oder Feldbustechnologie [Büh00, OPC99] verursacht, die auch die Arbeiten, die im Rahmen dieser Dissertation entstanden sind, beeinflusst haben.

Durch diese standardisierten Austauschformate lassen sich zum Einen Informationsverarbeitungsprozesse automatisieren (z.B. *Business to Business Supply Chain Management*), zum Anderen neue Dienstleistungen, wie z.B. intelligente Fernwartung von industriellen Automatisierungsanlagen (vgl. Abschnitt 5.6), realisieren. Generell besteht der Hauptnutzen dieser zunehmenden Vernetzung und Konsolidierung von Datenformaten in einer erweiterten Anwendbarkeit von bestehenden Erkenntnissen und Konzepten der Informatik. So wird es beispielsweise möglich, dass innerhalb des bereits erwähnten Fernwartungssystems Gerätedaten der Feldebene über Feldbusse und Internet in zentralen Datenbanken verwaltet und über anspruchsvolle Recherchemöglichkeiten zur Diagnoseunterstüzung im Rahmen von Fernwartungsaufgaben eingesetzt werden können.

1.1 Problemstellung und Zielsetzung

Diese Dissertation befasst sich im weitesten Sinn mit der Integration von heterogenen Informationsquellen und -senken in Internet-basierte Informationssysteme. Als Informationsquellen werden hierbei neben Datenbanken und Dateisystemen insbesondere Geräte sowie Softwarepakete verstanden. Die entwickelten Prototypen entstammen den Gebieten Fernwartung von Automatisierungsanlagen, Integration von industriellen Geräten in Internet-basierte Lehr-/Lernumgebungen und Integration von Computeralgebrasystemen in einen Online-Mathematikkurs.

Das Problem der offenen Integration heterogener Informationsquellen in Internet-basierte Informationssysteme zerfällt im Wesentlichen in vier Teile:

1. Flexible Informationserfassung auf der Server-Seite
2. Abbilden der Information auf ein einheitliches Zwischenformat
3. Übertragung der Information über das Internet

4. Flexible Visualisierung der Information auf der Client-Seite

Je nach Anwendungsgebiet sind die einzelnen Teile jeweils aufwändiger oder einfacher zu lösen. Die Informationserfassung auf der Server-Seite kann z.B. einen komplexen Zugriff auf Geräteparameter über ein industrielles Netzwerk der Feldebene erfordern. Bei der Abbildung auf ein einheitliches Zwischenformat müssen die potentiellen Formate bewertet, ausgewählt und eventuell neu erstellt werden. Hierbei hat sich die Verwendung von *XML* als vorteilhaft erwiesen.

Zur Übertragung der Information über das Internet müssen Client/Server-Architekturen entworfen und realisiert werden. Die Visualisierung auf der Client-Seite soll innerhalb von Standard-Web-Browsern erfolgen und die Generierung von dedizierten Sichten auf die Information sowie die Möglichkeit zur Interaktion unterstützen.

Als nicht funktionale Zielsetzungen werden insbesondere Plattformunabhängigkeit, Offenheit und Skalierbarkeit der Lösungen festgelegt, wobei sich die Plattformunabhängigkeit hierbei auf Client-, Server- und die Geräteseite bezieht. Offenheit meint, dass die einzelnen Lösungen flexibel erweitert werden können, sei es durch einfache Erweiterung/Anpassung der Software oder durch Flexibilität der Konzepte an sich. Um einen hohen Grad an Anpassbarkeit und Skalierbarkeit der Software zu erreichen, werden nach Möglichkeit keine Speziallösungen, sondern erweiterbare, ressourcenschonende Softwarebaukästen (Frameworks) entwickelt.

Des Weiteren entstehen Prototypen, die eine Einsetzbarkeit der entwickelten Lösungen beweisen. Konkret werden drei Systeme für die folgenden Problemstellungen entworfen, implementiert und getestet:

1. Offene und intelligente Fernwartung von Geräten via Internet ($\mathbb{A}$)

2. Einbindung von industriellen Feldbusgeräten in eine Internet-basierte Lehr-/Lernumgebung ($\mathbb{B}$)

3. Einbindung von Computeralgebrasystemen in einen Online-Mathematikkurs ($\mathbb{C}$)

Das erste System stellt eine offene, plattformunabhängige Infrastruktur zur Verfügung, die es erlaubt, Geräte und Anlagen beliebiger Art auf einfache Weise in Internet-basierte Informationssysteme zu integrieren. Anlagen und Geräteschnittstellen werden durch ein gerätetypunabhängiges, universelles Datenformat beschrieben, das auch die Zugangswege zu der jeweiligen Hardware spezifiziert. Die Infrastruktur unterstützt sowohl den lesenden als auch den schreibenden Zugriff auf einzelne Geräteparameter. Erhobene Daten werden in entfernten Datenbanken in einem aggregierten, einheitlichen Format persistent abgelegt und können durch ein flexibles Recherchekonzept analysiert werden.

Die Aggregation derartiger Prozesszustände soll entweder zyklisch geschehen, um eine Anlage zu überwachen, oder zu bestimmten Zeitpunkten, z.B. aufgrund einer entfernten Anfrage eines Clients via Internet. Insbesondere das systematische Ablegen von Fehlerzuständen, in Verbindung mit einem flexiblen Ähnlichkeitsmaß, erlaubt eine automatisierte, intelligente Fehlerdiagnoseunterstützung für entfernte Anlagen.

Im Einzelnen werden folgende Zielsetzungen definiert:

- $\mathbb{A}1$ Entwicklung eines Konzepts zur gerätetypunabhängigen Beschreibung von Anlagen, Geräten und Geräteschnittstellen
- $\mathbb{A}2$ Entwicklung eines Konzepts zur gerätetypunabhängigen Repräsentation von Anlagenzuständen
- $\mathbb{A}3$ Entwicklung eines Konzepts zur flexiblen, dynamischen Gerätedatenerfassung, unabhängig von Gerätetypen und Zugriffsprotokollen
- $\mathbb{A}4$ Entwicklung eines Konzepts zur effizienten Ablage von Anlagenzuständen in entfernten Datenbanken
- $\mathbb{A}5$ Entwicklung eines Konzepts zur flexiblen, interaktiven Visualisierung von Anlagenzuständen in entfernten Web-Browsern
- $\mathbb{A}6$ Entwicklung eines Konzepts zum entfernten, interaktiven Zugriff auf Geräteparameter und Managementfunktionalität, unabhängig von Gerätetypen und Zugriffsprotokollen
- $\mathbb{A}7$ Entwicklung eines Konzepts zur flexiblen Ähnlichkeitsbewertung von Anlagenzuständen
- $\mathbb{A}8$ Implementierung entsprechender Prototypen zur Realisierung einer Gesamtlösung

Das zweite System behandelt die Einbindung von industriellen Feldbusgeräten in eine Internet-basierte Lehr-/Lernumgebung. Die Aufgabe besteht hier im Wesentlichen darin, Geräte über möglichst generische Client/Server-Systeme entfernten Benutzern (hier z.B. Studenten) zugänglich zu machen. Der Zugang kann auf unterschiedlichen Abstraktionsebenen unterstützt werden, z.B. über dedizierte GUI-Komponenten, die direkt die Funktionalität der Geräte zugänglich machen, oder über direkten Zugriff auf protokollspezifische Kommunikationsobjekte.

Im Einzelnen werden folgende Zielsetzungen definiert:

- $\mathbb{B}1$ Entwurf und Implementierung einer objektorientierten API für das CANopen-Protokoll in Java
- $\mathbb{B}2$ Entwicklung und Implementierung eines offenen Konzepts zur Integration von CANopen-Feldbusgeräten in das World Wide Web basierend auf $\mathbb{B}1$
- $\mathbb{B}3$ Entwurf und Implementierung eines objektorientierten Software-Frameworks zur einfachen Realisierung von Gerätesteuerungen für industrielle Feldbusanlagen in Java.
- $\mathbb{B}4$ Konzeption und Realisierung einer Internet-basierten Lehr-/Lernumgebung.
- $\mathbb{B}5$ Konzeption und Realisierung diverser Übungsaufgaben für die Lehr-/Lernumgebung basierend auf den Ergebnissen aus $\mathbb{B}1$-$\mathbb{B}3$

System Nummer 3 entspricht in weiten Teilen System Nummer 2, mit dem Unterschied, dass die Geräte als dynamische Informationsquellen durch Computeralgebrasysteme ersetzt sind. Konkret wird eine Tool-Box entwickelt, die unterschiedliche Konzepte (Funktionen, Folgen, Flächen, Differentiation, Integration usw.) der Mathematik interaktiv aufbereitet. An den Stellen, an denen eine numerische Behandlung der Konzepte nicht genügt, wird auf Server-seitige Computeralgebrasoftware zugegriffen. Wichtig ist hierbei für die Benutzer, dass dieses transparent geschieht und kein direkter Zugang zu diesen Systemen vorausgesetzt wird. Des Weiteren werden keine Kenntnisse bezüglich der speziellen Syntax der einzelnen Computeralgebrasysteme erwartet. Stattdessen werden spezielle Java-Applets mit dedizierten Benutzerschnittstellen entwickelt, die einen problemorientierten Zugang zu dem entfernten Computeralgebrasystem bilden.

Im Einzelnen werden folgende Zielsetzungen definiert:

- $\mathbb{C}1$ Entwicklung und Auswahl von Client/Server-Systemen zur Integration von Computeralgebrasystemen in eine Internet-basierte Lehr-/Lernumgebung.
- $\mathbb{C}2$ Entwicklung, Auswahl und Bewertung von Visualisierungskonzepten für mathematische Konzepte
- $\mathbb{C}3$ Implementierung einer Tool-Box für diverse mathematische Konzepte, basierend auf den Ergebnissen aus $\mathbb{C}1$ und $\mathbb{C}2$

1.2 Lösungen

Um die im vorigen Abschnitt definierten Zielsetzungen erreichen zu können, basieren die entwickelten Lösungen hauptsächlich auf den folgenden grundlegenden Konzepten:

1. Offene Standards:
   (a) X-Technologien (*XML*, *XSL*, SAX, DOM ...)
   (b) Netzwerkprotokolle (HTTP, TCP/IP, CANopen...)
2. Objektorientiertes Softwaredesign (Patterns, Frameworks ...)
3. Java (Applets, Servlets, Reflection, RMI, JDBC, JNI, JAXP ...)

Die Verwendung offener Standards ist besonders wichtig im Umfeld der Integration unterschiedlicher Geräteplattformen, da diese oft erst eine durchgängige Kommunikation unterschiedlicher Gerätetypen ermöglichen oder zumindest die Zahl der integrierbaren Plattformen erhöhen. Die *eXtensible Markup Language* (*XML*, siehe Abschnitt 2.2) hat sich beispielsweise in diesem Kontext als Standard zur textuellen Repräsentation von strukturierter Information stark durchgesetzt. Ein genereller Vorteil offener Standards ist die Verfügbarkeit generischer Werkzeuge von dritter Seite. Im *XML*-Umfeld ist z.B. in den letzten Jahren das Angebot an Editoren, Datenbanksystemen, Visualisierungskonzepten, Programmierumgebungen etc. stark gestiegen. Diese Werkzeuge können unabhängig von der jeweiligen konkreten Anwendung für alle *XML*-basierten Lösungen eingesetzt werden. Zusätzlich wurde darauf geachtet, dass die verwendeten Standards möglichst unabhängig von bestimmten Betriebssystemen sind.

Im Rahmen dieser Arbeit hat die Entscheidung, *XML* als universelles Datenformat zu verwenden, zur Entwicklung mehrerer Markup-Sprachen (u.a. der *Device Management Markup Language* (*DeMML*) und der *XML to Java Mapping Language* (*XJML*)) geführt, die jeweils durch entsprechende *XML*-Grammatiken definiert sind. Diese Sprachen können mit entsprechenden spezialisierten oder allgemeinen Werkzeugen plattformunabhängig bearbeitet und verwaltet werden.

Die entwickelte Software basiert auf dem Ansatz der Objektorientierung, was sich als dem Problemumfeld angemessen erwiesen hat. Es wurden diverse objektorientierte Software-Bibliotheken und -Frameworks geschaffen, die einen hohen Grad an Wiederverwendung der entwickelten Softwarelösungen ermöglichen.

Als Programmiersprachen zur Implementierung der Prototypen wurden Java und für den hardwarenahen Teil C++ eingesetzt. Java wurde insbesondere aufgrund der Plattformunabhängigkeit, der Vielfalt an frei verfügbaren Bibliotheken für die unterschiedlichsten Anwendungsgebiete und aufgrund des *Java Reflection*-Konzepts (vgl. Abschnitt 5.5.3) sowohl auf Client- als auch auf Server-Seite verwendet. C++ wurde als native Programmiersprache nur dann eingesetzt, wenn auf hardwarespezifische Ressourcen zugegriffen werden muss. Es zeichnet sich im Moment allgemein ein Trend zum verstärkten Einsatz von Java im Bereich eingebetteter Systeme und Realzeit Gerätesteuerung [BG00a, Har01, KK00] sowie zur Erstellung komplexer Internet-basierter Informationssysteme ab.

Die zentralen Ergebnisse dieser Dissertation sind die folgenden:

1. Offene und intelligente Fernwartung von Geräten via Internet
   (a) Definition der *Device Management Markup Language* (*DeMML*) Sprache als einheitliches, erweiterbares Datenformat zur Repräsentation von Anlagen, Geräten, Geräteschnittstellen und Anlagenzuständen ($\mathbb{A}$1, $\mathbb{A}$2)
   (b) Entwicklung des *XML to Java Mapping* (*XJM*) Konzepts und der *XML to Java Mapping Language* (*XJML*) Sprache zur Integration von Java-Logik in *XML*-Dokumente ($\mathbb{A}$3, $\mathbb{A}$7)
   (c) Implementierung des *XJM_Eval*-Systems zur Spezifikation und Auswertung von Ähnlichkeitsmaßen für *XML*-Dokumente durch die *XJM*-basierte Kombination von Java-Logik und *XML*-Querysprachen ($\mathbb{A}$7)
   (d) Entwurf und Implementierung von *DeviceInvestigator* als *XJM*-basiertes Tool und Software-Framework zum generischen Zugriff auf Geräteparameter über *Java Reflection* ($\mathbb{A}$3, $\mathbb{A}$8)
   (e) Entwurf und Implementierung des INSIGHT-Systems zur intelligenten Einbindung von Geräten und Anlagen in Internet-basierte Informationssysteme, basierend auf den Ergebnissen 1a - 1d ($\mathbb{A}$5, $\mathbb{A}$6, $\mathbb{A}$8)
   (f) Integration einer nativen *XML*-Datenbank in das INSIGHT-System ($\mathbb{A}$4)
2. Einbindung von industriellen Feldbusgeräten in eine Internet-basierte Lehr-/Lernumgebung
   (a) Entwurf und Implementierung der Java CAN API als objektorientierte CAN-Feldbus-Kommunikationsschnittstelle ($\mathbb{B}$1, $\mathbb{B}$2)
   (b) Entwurf und Implementierung des *Java Remote CAN Control* (*JRCC*) Systems zur Anbindung beliebiger CANopen-Geräte an das Internet ($\mathbb{B}$1, $\mathbb{B}$5)
   (c) Entwurf und Implementierung des *Java Fieldbus-based Control Framework*s (*JFCF*) als objektorientiertes Software-Framework zur Spezifikation von Gerätesteuerungen ($\mathbb{B}$3, $\mathbb{B}$5)
   (d) Mitgestaltung des *Virtual Automation Lab* als Internet-basierte Lehr-/Lernumgebung

(e) Realisierung von Internet-basierten Laborübungen aus den Bereichen Gerätesteuerung, Telematik und Feldbusprotokolle, basierend auf den Ergebnissen 2a - 2c ($\mathbb{B}4$, $\mathbb{B}5$)

3. Einbindung von Computeralgebrasystemen in einen Online-Mathematikkurs.

   (a) Realisierung eines Internet-basierten, interaktiven Mathematik-Kursbuches ($\mathbb{C}1$)

   (b) Implementierung und Bewertung unterschiedlicher Visualisierungskonzepte unter Einbeziehung von Computeralgebrasystemen ($\mathbb{C}2$)

   (c) Implementierung Browser-basierter Visualisierungen für diverse mathematische Konzepte ($\mathbb{C}3$)

Die einzelnen Konzepte und Systeme werden in den entsprechenden Abschnitten dieser Dissertation ausführlich eingeführt und erläutert.

1.3 Aufbau der Dissertation

Das Wesen der vorliegende Arbeit liegt hauptsächlich in der Erforschung und Entwicklung durchgängiger Konzepte zur flexiblen Integration von beliebigen Geräten in moderne, intelligente Informationssysteme. Sie tangiert daher eine Vielzahl von Disziplinen der Informatik, wie z.B. industrielle Datenerfassungssysteme, Feldbusprotokolle, Internet-Technologien, Datenbanksysteme sowie Information Retrieval- und Softwaretechnikkonzepte. Es wird deshalb darauf verzichtet, eine generelle Einführung in die einzelnen Problembereiche am Anfang dieser Dissertation zu geben. Diese erfolgt vielmehr innerhalb der entsprechenden Kapitel zusammen mit einem Überblick über die jeweils bereits vorhandenen Forschungsergebnisse und/oder alternativen Lösungsansätze.

Kapitel 2 gibt zunächst eine Einführung in die systemtechnischen Grundlagen und Basistechnologien, die für das Gesamtkonzept dieser Arbeit wichtig sind. Dies sind im Einzelnen Internet-basierte Informationssysteme, die *eXtensible Markup Language* (*XML*) als universelles Datenformat und das Konzept der objektorientierten Software-Frameworks, insbesondere von *XML*-basierten Software-Frameworks.

Die eigentliche Arbeit gliedert sich dann im Folgenden in zwei Teile. Der erste Teil stellt das INSIGHT-System vor, das die schnelle und flexible Erstellung Internet-basierter Fernwartungssysteme für industrielle Anlagen und Geräte unterstützt.

Kapitel 3 gibt zunächst einen kurzen Überblick über das INSIGHT-System. Es werden die wichtigsten Nutzungsarten definiert und ein Überblick über die entsprechende Systemfunktionalität und Systemarchitektur gegeben.

In **Kapitel 4** wird die *Device Management Markup Language* (*DeMML*) als universelles Datenformat des INSIGHT-Systems detailliert vorgestellt. In weiteren Abschnitten werden das Erweiterbarkeitskonzept von *DeMML* und das *DeMML*-Java-Paket erläutert sowie alternative Ansätze beschrieben und bewertet.

Kapitel 5 behandelt die *DeviceInvestigator*-Komponente als *XML*-basiertes Software-Framework zur flexiblen und gerätetypunabhängigen Datenaggregation. Das Alleinstellungsmerkmal von *DeviceInvestigator* gegenüber aktuellen Datenerfassungs- und Monitoring-Systemen beruht auf der konsequenten Verwendung von *XML* als Datenformat sowohl auf Datenerfassungs-, Kommunikations- als auch auf Datenbankebene in Verbindung mit dem *Java Reflection*-Konzept. Dieser Ansatz ermöglicht den Grad an Offenheit, Flexibilität, Plattformunabhängigkeit und Skalierbarkeit, der von der INSIGHT-Datenaggregationskomponente verlangt wurde.

Die Einbindung von *DeviceInvestigator* in ein Internet-basiertes Informationssystem wird in **Kapitel 6** beschrieben. Es werden die unterschiedlichen Kommunikationswege und Visualisierungskonzepte zur Generierung dedizierter Sichten auf *DeMML*-Zustandsdokumente vorgestellt. Weiterhin wird in diesem Kapitel die Einbindung eines nativen *XML*-Datenbanksystems in das INSIGHT-System erläutert und bewertet.

Kapitel 7 beschreibt mit *XJM_Eval* das Recherchekonzept des INSIGHT-Systems. *XJM_Eval* ist wieder als *XML*-basiertes Software-Framework organisiert und erlaubt eine flexible, inhalts- wie strukturbezogene Ähnlichkeitsbewertung für beliebige *XML*-Dokumente. Das Ähnlichkeitsmaß selbst wird als *XML*-Instanz der *XML to Java Mapping Language* formuliert und spezifiziert durch die Kombination von *XML*-Querysprachen mit der Programmiersprache Java die Evaluation des Ähnlichkeitsmaßes. Der *XJM_Eval*-Ansatz wird auch in Relation zu Forschungsergebnissen aus dem Bereich der Ähnlichkeitsbewertung semistrukturierter Daten gesetzt und entsprechend bewertet.

Kapitel 8 fasst die Konzepte und Ergebnisse des INSIGHT-Systems noch einmal zusammen.

Im zweiten Teil dieser Arbeit werden die entwickelten offenen Konzepte zur Integration externer Komponenten in Internet-basierte Lehr-/Lernumgebungen vorgestellt. Unter „externen Komponenten“ werden in diesem Kontext sowohl industrielle Anlagen als auch eigenständige Softwarepakete verstanden.

Kapitel 9 gibt eine Einführung in die Thematik der Integration realer Geräte in e-Learning Systeme zur Erstellung sog. *virtueller Labors*. Es werden die Zielsetzungen und Anforderungen an die entsprechenden Integrationskonzepte definiert, die Lösungsansätze skizziert und es wird ein Überblick über bestehende *virtuelle Labors* gegeben.

In **Kapitel 10 und 11** werden die grundlegen Software-Frameworks und APIs vorgestellt, die zur Realisierung des *virtullen Labors* entwickelt wurden. Im Einzelnen sind dies die Java CAN API (Kapitel 10), die eine objektorientierte Programmierschnittstelle für CAN-Feldbus-Kommunikation in Java realisiert, und das *Java Fieldbus-based Control Framework* (Kapitel 11), das objektorientierten Zugriff auf industrielle Geräte und die wiederverwendbare Erstellung nebenläufiger Gerätesteuerungsalgorithmen in Java unterstützt.

Kapitel 12 stellt dann die konkret realisierten Laborübungen vor, die auf Geräteseite eine Leuchtschrift, diverse Feldbusgeräte und eine Feldbusanlage zur Vereinzelung von Werkstücken umfassen. Die Laborübungen können mit Standard-Web-Browsern ohne zeitliche oder räumliche Einschränkungen durchgeführt werden.

Kapitel 13 behandelt die offene Integration von Computeralgebrasystemen als weiteren Vertreter „externer Komponenten“ in ein interaktives, Internet-basiertes Mathematik-Kursbuch und zeigt Parallelen zu den in Kapiteln 9-12 vorgestellten Konzepten auf. Es werden unterschiedliche Visualisierungskonzepte für mathematische Phänomene beschrieben und bewertet. Insbesondere wird das entwickelte Konzept der *hybriden Visualisierung* vorgestellt, das auf einer Java-basierten, interaktiven Darstellung mathematischer Phänomene unter Einbeziehung bestimmter Funktionalität entfernter Computeralgebrasysteme beruht.

Kapitel 14 fasst die Ergebnisse des zweiten Teils dieser Arbeit noch einmal zusammen.

In **Kapitel 15 und 16** werden die Ergebnisse dieser Dissertation zusammengefasst und Ansätze für weitere Forschungsarbeiten aufgezeigt.

Kapitel 2

Systemtechnische Grundlagen

2.1 Internet-basierte Informationssysteme

Das Internet durchdringt heute nahezu sämtliche Bereiche das alltäglichen Lebens. Insbesondere das *World Wide Web* (WWW) [BLCL$^+$94, BLCGP92] hat sich als Hypertext-basiertes *Internet Information System* (IIS) stark durchgesetzt. Die Omnipräsenz des WWW hat mittlerweile andere IIS-Technologien, wie z.B. WAIS oder Gopher in den Hintergrund treten lassen (Analysen und Vergleiche der entsprechenden Technologien finden sich in [APK96, SEKN92, ODL93]), so dass der IIS-Begriff heute oft synonym zu WWW-basierten Ansätzen verwendet wird. Auch im Rahmen dieser Arbeit bezeichnen Internet-basierte Informationssysteme i.d.R. WWW-basierte Systeme.

Das WWW hat sich in den letzten Jahren stark weiterentwickelt. Zum Einen wurde *HTML* [Wor99b] als Hypertext-Format des WWW um die erweiterbare Markup Sprache *XML* (siehe Abschnitt 2.2) ergänzt, zum Anderen hat neben ursprünglichem *Content Management* das Anbieten dedizierter Funktionalität (*Services*) an Bedeutung gewonnen. In diesem Kontext wird insbesondere zunehmend der Begriff *Internet-Portal* verwendet.

Unter einem Internet-Portal wird i.d.R. ein System verstanden, das Folgendes leistet:

1. Integration
2. Personalisierung
3. Sicherheit

Integration Unter Integration wird in diesem Zusammenhang die Schaffung einer zentralen Anlaufstelle zum einheitlichen und transparenten Zugriff auf heterogene Datenquellen verstanden. An diesem Integrationsprozess sind Komponenten zur Datenaggregation und Transformation beteiligt. Gelegentlich werden Konzepte der Künstlichen Intelligenz eingesetzt, um – für den Benutzer[1] unsichtbar – die nötigen Anfragen zur Konstruktion eines Ergebnisses aufgrund von z.B. Deduktionsregeln zu erschließen. Bekannte Systeme in diesem Umfeld sind beispielsweise das TSIMMIS Projekt [CGMH+94, HGMN+97] oder das SIMS-System [ACHK93].

Vor dem Hintergrund der zunehmenden Verbreitung von *HTML* als Kommunikationsformat wurden diverse sog. *Wrapper*-Systeme zur Integration heterogener semistrukturierter Daten (z.B. Web-Seiten) entworfen (z.B. [KWD97, MSS99, LHB+99, SA99]). Mit Hilfe dieser Systeme ist es möglich, Angebote unterschiedlicher Informationsanbieter auf dem WWW in ein Zwischenformat (z.B. *XML*) zu transformieren und zu neuen Angeboten zu konsolidieren (vgl. auch [Büh98]).

Personalisierung Unter Personalisierung von Internet-Portalen versteht man zum Einen, dass ein Benutzer aktiv das Erscheinungsbild des Portals anpassen kann. Hierzu kann der Benutzer i.d.R. auswählen, welche Informationen ihm auf welchem Weg zugänglich gemacht werden soll (z.B. Ein-, Ausblenden bestimmter Börsenkurse bei *my.yahoo.com*). Zum Anderen kann sich das Portal selbstständig an die Vorlieben des Benutzers anpassen. Das Portal *www.amazon.com* schlägt z.B. auf Wunsch selbstständig Neuerscheinungen von bereits gekauften Autoren vor.

Voraussetzung für eine derartige Personalisierung ist die Verwaltung von Benutzerprofilen und die Realisierung von Authentifizierungskonzepten.

Sicherheit Mit zunehmender Personalisierung von Portalen wachsen auch die Sicherheitsansprüche der Benutzer, da Personalisierung zwar eine Qualitätssteigerung eines Angebots ermöglicht, zugleich aber die Gefahr des Datenmissbrauchs steigt. Zusätzlich ist natürlich bei elektronischem Zahlungsverkehr die Implementierung von Sicherheitskonzepten unumgänglich.

Es werden ein ganze Reihe von Portaltypen, wie z.B. Unternehmens-Portale (für Intranet und Internet), Business-to-Business-Portale, e-Shops, Service-Portale usw. unterschieden. Eine Übersicht über unterschiedliche Typen von Portalen und einige Fallstudien finden sich beispielsweise in [Fau00].

[1] Ein Benutzer eines IIS kann sowohl ein menschlicher Benutzer oder ein Computerprogramm sein.

2.2 Die *eXtensible Markup Language* (XML)

In diesem Abschnitt werden die grundlegenden Konzepte der Sprache *XML* [Wor98b] und verwandter Technologien vorgestellt. Dabei wird besonderes Gewicht auf die im Rahmen dieser Arbeit relevanten Themenbereiche gelegt. Eine gute Übersicht über *XML*-Technologien findet sich beispielsweise in [BM00, LB99].

2.2.1 Markup-Sprachen

Ein Dokument einer Markup-Sprache ist zunächst eine einfache Zeichenkette. Diese Zeichenkette besteht auf logischer Ebene aus zwei unterschiedlichen Arten von Text, der entweder die eigentliche Information beinhaltet oder aber Information über die Information (Metainformation). Für jedes Zeichen ist die jeweilige Zugehörigkeit syntaktisch eindeutig festgelegt. Der Textanteil, der die Metainformation beinhaltet, wird dann als *Markup* bezeichnet.

Es werden im Wesentlichen zwei Arten von Markup unterschieden: Spezialisiertes Markup z.B. zur Formatierung von Text und allgemeines Markup (*Generalized Markup*) zur allgemeinen Induktion von expliziter Struktur in textuelle Daten.

Spezialisiertes Markup zur Formatierung von Text Dieses Markup enthält Informationen über das graphische Layout (z.B. Schriftgröße und Schriftart) der eigentlichen Information. Das vorliegende Dokument wurde beispielsweise mit dem Textsatzsystem LaTeX erstellt und enthält neben der dargestellten Textinformation eine beträchtliche Menge an LaTeX-Markup, das die graphische Formatierung der textuellen Information festlegt.

Markup in HTML Die *Hypertext Markup Language (HTML)* [Wor99b] stellt einen definierten Satz an sog. *Tags* zur Verfügung, die als Markup in Dokumente eingefügt werden können. Die einzelnen Tags werden im Wesentlichen zur Textformatierung und zur Referenzierung von externen Ressourcen auf dem Internet verwendet. Es besteht keine Möglichkeit für einen Autor, selbst neue *HTML*-Tags und damit Strukturierungsmöglichkeiten zu ergänzen.

Generalized Markup Generalized Markup zeigt allgemein eine beliebig geartete Signifikanz eines bestimmten Teiles eines Textes an und induziert so eine explizite logische Struktur in eine für den Rechner ansonsten flache Zeichenkette. Markup und Layout werden streng voneinander getrennt. Bestimmten Teilen des Markup kann über entsprechende separate sog. *Style-Sheets* eine visuelle Repräsentation nachträglich zugeordnet werden. Diese Trennung erlaubt die konsistente Erzeugung unterschiedlicher Sichten auf ein Dokument, da nur die jeweiligen Style-Sheets angepasst werden müssen und das eigentliche Dokument unverändert bleiben kann (*Single-Source*-Prinzip).

2.2.2 Die Entstehung von *XML*

Sowohl *HTML* als auch *XML* basieren auf der *Standard Generalized Markup Language* (*SGML*) [Int86]. Allerdings ist *XML* eine Teilmenge von *SGML*, während *HTML* lediglich eine sog. *SGML*-Applikation (s.u.) darstellt. Dies hat Auswirkungen auf Mächtigkeit, Flexibilität und Erweiterbarkeit der Sprachen.

SGML wurde ursprünglich entworfen, um große Datenmengen plattformunabhängig und über lange Zeiträume hinweg in textueller Form zu speichern. Insbesondere sollten die Daten über viele Rechnergenerationen hinweg verarbeitbar bleiben, auch wenn das ursprünglich verwendete Autorensystem mittlerweile auf keinem Rechner mehr verfügbar sein sollte. Seit 1986 ist *SGML* ein ISO-Standard [Int86], der vor allem in der Industrie zur strukturierten Speicherung von Informationen (z.B. Produktdokumentationen) eingesetzt wird.

Der entscheidende Nachteil von *SGML* ist, dass die Sprache zwar sehr mächtig, aber auch sehr komplex und damit schwierig zu bearbeiten ist. Dies hat zur Folge, dass Applikationen, die *SGML*-Dokumente bearbeiten können, oft ebenso komplex, groß und teuer sind.

Andererseits zeichnete es sich bereits früh ab, dass die weitverbreitete und leicht handhabbare *SGML*-Applikation *HTML* in ihrer starren Form nicht für alle Arten der Informationsrepräsentation geeignet sein würde. Insbesondere ist die Abbildung von hochstrukturierter Information, wie zum Beispiel Ergebnis-Tupel relationaler Datenbanken auf *HTML*-Dokumente nur unbefriedigend durchführbar, da sowohl Struktur- als auch Datentypinformationen verloren gehen.

Mitte 1996 fanden sich dann Mitglieder des *World Wide Web Consortium*[2] (W3C) und verschiedene *SGML*-Experten zu der sog. *XML Working Group* zusammen, um eine Markup-Sprache zu entwickeln, die die Mächtigkeit und Flexibilität von *SGML* und die Einfachheit von *HTML* in sich vereinen sollte. An diese Sprache wurden im Wesentlichen die folgenden Forderungen gestellt:

- Die Sprache sollte das Konzept des *Generalized Markup* unterstützen.
- Sie sollte aufwärtskompatibel zu *SGML* sein.

[2] http://w3c.org

- Sie sollte wie *SGML* auch eine Metasprache sein, die die Spezifikation von Dokumentklassen erlaubt.
- Sie sollte Referenzierungen von Ressourcen auf dem Internet durch das Konzept des *Uniform Resource Locator* (URL) [BLMM94] unterstützen.

Im November desselben Jahres wurde daraufhin der erste *XML Working Draft* zur Diskussion gestellt, und Anfang 1998 der erste *XML*-Standard [Wor98b] veröffentlicht. Allgemein wird der Trade-off als gut ausbalanciert (80% der Funktionalität bei 20% der Komplexität von *SGML*) beurteilt.

2.2.3 *XML*-Dokumente

XML-Dokumente bestehen aus der eigentlichen textuellen Information und zusätzlichem *Generalized Markup*. Informell besteht der Markup-Anteil eines *XML*-Dokuments aus den Teil-Zeichenketten, die jeweils durch spitze Klammern ($<$, $>$) eingeschlossen sind (vgl. Abbildung 2.1). Die verwendete Zeichenkodierung (z.B. Unicode UTF-8 [The96]) wird explizit in einem Dokumentprolog angegeben, so dass keine Beschränkungen bzgl. des verwendeten Alphabets bestehen. Dies ist z.B. besonders für den asiatischen Raum wichtig.

Durch *XML*-Markup wird sowohl die logische als auch die physische Struktur eines *XML*-Dokuments festgelegt.

Logische *XML*-Struktur Die logische Strukturierung von *XML* basiert auf der Schachtelung sog. *XML-Elemente*, die durch entsprechende *Tags* eingerahmt werden (vgl. Abbildung 2.1). Die einzelnen Tags sind jeweils durch einen Tag-Namen, der gleichzeitig den Namen des entsprechenden Elements definiert, und möglicherweise vorhandenen Attributen ausgezeichnet, wobei Attribute nur in den öffnenden Tags angegeben werden dürfen. Im Gegensatz zu den Elementen haben Attribute keine interne Struktur. Da die Elemente korrekt geschachtelt sein müssen und jedes *XML*-Dokument genau ein Wurzelelement besitzen muss, ist die logische Struktur eines jeden *XML*-Dokuments die eines geordneten Wurzelbaumes.

Die syntaktischen Strukturierungsmöglichkeiten sind in [Wor98b] formal als EBNF (*Extended Backus-Naur Form*) Grammatik formuliert. Eine Zeichenkette stellt genau dann ein wohlgeformtes *XML*-Dokument dar, wenn sie durch diese Grammatik erzeugt werden kann.

Physikalische *XML*-Struktur Die physikalische Strukturierung von *XML*-Dokumenten basiert auf der Vernetzung mehrerer Ressourcen durch entsprechende Referenzen, die zum größten Teil auf der URI [BLRIM98] Spezifikation beruhen. Ein *XML*-Dokument kann also dynamisch aus mehreren Teilen (*Entities*)

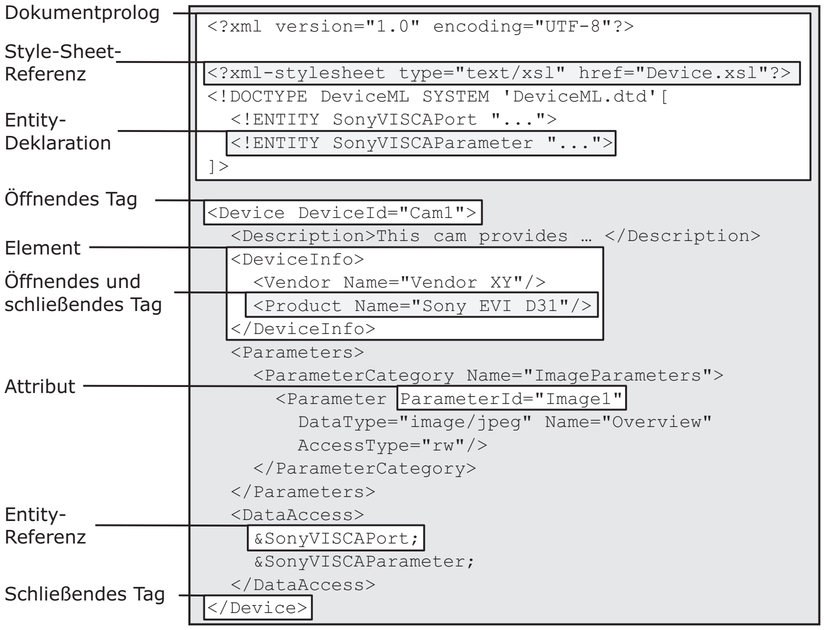

Abbildung 2.1: Ein *XML*-Dokument

zusammengesetzt werden, was wieder die Vermeidung von Redundanzen für oft verwendete Dokumentfragmente erleichtert.

2.2.3.1 *XML* als Metasprache

Die erlaubte logische Struktur von *XML*-Dokumenten kann durch eine sog. *Document Type Definition* (DTD) auf eine bestimmte Dokumentklasse eingeschränkt werden. Der *XML*-Standard definiert hierzu eine Reihe von Spezifikationsmöglichkeiten, die beispielsweise Informationen darüber enthalten, welche Tag-Namen erlaubt sind, welche Tags welche Attribute aufweisen und welche Elemente wie in welche anderen Elemente geschachtelt werden dürfen bzw. welche Elemente (zusätzlich) freien Text enthalten dürfen. Eine Dokumentklasse wird auch in Analogie zu *SGML* als *XML-Applikation* bezeichnet. Wie bereits erwähnt ist beispielsweise *HTML* eine *SGML*-Applikation.

Wenn ein wohlgeformtes *XML*-Dokument den Spezifikationen einer bestimmten DTD genügt, wird es als *gültiges* Dokument bezeichnet und ist eine Instanz der

Parser	Sprache	Urheber	URL
Xerces	Java, C++, Perl	Apache	xml.apache.org
expat	C	James Clark	www.jclark.com/xml/expat.html
xml4j, xjml4c	Java, C++	IBM	www.alphaworks.ibm.com/tech/xml4j www.alphaworks.ibm.com/tech/xml4c
PyXML	Python	python.org	pyxml.sourceforge.net/topics/
TclXML	Tcl	Steve Ball	www.zveno.com/zm.cgi/in-tclxml/

Tabelle 2.1: Einige *XML*-Parser für unterschiedliche Programmiersprachen

entsprechenden Dokumentklasse oder *XML*-Applikation. Das in Abbildung 2.1 dargestellte *XML*-Dokument ist beispielsweise eine Instanz der *Device Markup Language* (*DeviceML*) (vgl. Abschnitt 4.3), deren DTD im Anhang A.2 auf Seite 230 angegeben ist.

Zur Zeit existieren eine Vielzahl von Projekten zur Erstellung spezieller DTDs für die unterschiedlichsten Anwendungsgebiete, z.B. MathML [Wor01b] zur Repräsentation von mathematischen Formeln oder VML [MLDea98] zur Repräsentation von Vektorgraphik. Die Organisation *XML.org* (www.xml.org) verwaltet einen Online-Katalog über bestehende und entstehende *XML*-Applikationen. Der Vorteil dieser Standardisierungen ist, dass allgemein verständliche Formate und Vokabularien entwickelt werden, die einen Datenaustausch über Applikations- und Plattformgrenzen hinaus ermöglichen. Alle diese Formate können hierbei mit denselben allgemeinen *XML*-Werkzeugen (z.B. Editoren, Datenbanken, Viewer) und Programmierschnittstellen bearbeitet werden, da die einzelnen Dokumentinstanzen immer wohlgeformte *XML*-Dokumente darstellen.

2.2.3.2 Das *Document Object Model*

Zur Verarbeitung von *XML*-Daten wurden eine Vielzahl von frei verfügbaren Parsern und APIs für unterschiedliche Programmiersprachen entwickelt (siehe Tabelle 2.1).

Die meisten dieser Parser basieren auf der von David Megginson entwickelten *Simple API for XML* (SAX) Schnittstelle (http://www.megginson.com/SAX/). Diese Schnittstelle definiert einen Satz von Callback-Methoden (z.B. *startElement()*), die während der Analyse eines *XML*-Dokuments durch den SAX Parser aufgerufen werden. Eine Applikation kann diese Callback Methoden überschreiben, um die *XML*-Informationen zu verarbeiten oder eine interne Datenstruktur zur Repräsentation der *XML*-Information aufzubauen.

Das Standarddatenmodell zur Repräsentation von *XML*-Informationen ist das durch das W3C entwickelte *Document Object Model* (DOM) [Wor98a, Wor99a,

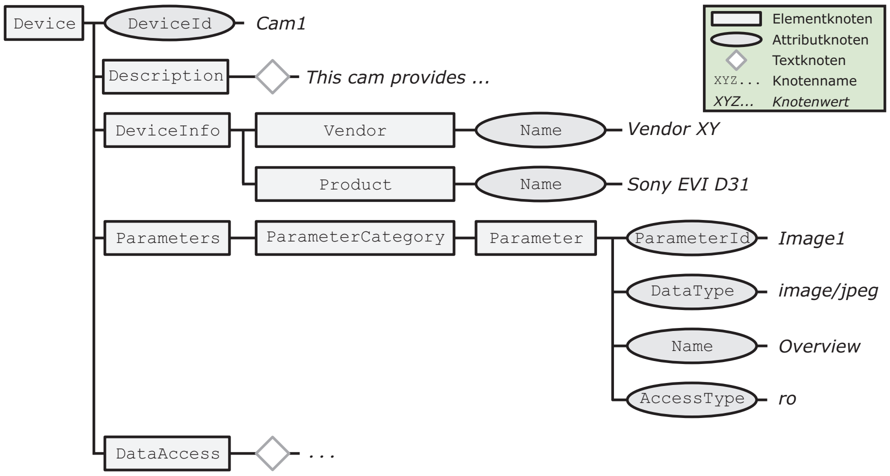

Abbildung 2.2: Das DOM-Datenmodell zu Abbildung 2.1

Wor01a]. Das DOM eines *XML*-Dokuments ist ein Baum mit unterschiedlichen Knotentypen für *XML*-Elemente, -Attribute, -Entities usw. (vgl. Abbildung 2.2). Je nach Knotentyp kann ein Knoten zusätzlich zu den Referenzen auf evtl. Kindknoten noch eigene Daten tragen (z.B. tragen Attributknoten jeweils den zugehörigen Attributwert). Die DOM-Spezifikation definiert zusätzlich die konkreten Programmierschnittstellen[3] zum Traversieren (z.B. *getFirstChild()*, *getNodeValue()*) und Modifizieren (z.B. *appendChild()*, *setNodeValue()*) dieser Baumstruktur.

2.2.4 Die *eXtensible Stylesheet Language* (*XSL*)

XSL ist eine *XML*-Applikation zur Formulierung von sog. *Style-Sheets* für *XML*-Dokumente. Ein *XSL*-Style-Sheet spezifiziert eine Transformation von *XML*-Dokumenten in andere Dokumentenformate, die beispielsweise eine graphische Präsentation der Dokumente erlauben.

XSL [Wor00a] besteht aus drei Teilen:

1. *XML Path Language* (*XPath*) [CD99]

2. *XSL Transformations* (*XSLT*) [Wor99c]

[3] Es werden wirklich nur die Schnittstellen definiert und keine Implementationen angegeben.

3. *XML Formatting Objects*

Die drei Teile können jeweils unabhängig voneinander verwendet werden, bilden zusammen aber eine Einheit zur Transformation von *XML*-Daten in graphische Information zur Darstellung auf unterschiedlichen Endgeräten oder Medien.

2.2.4.1 *XPath*

Der *XPath*-Standard wird sowohl im Kontext von *XSL* als auch von *XPointer* [Wor01c] zur Selektion bestimmter Teile eines *XML*-Dokuments verwendet. Die Selektion basiert auf der Angabe von Pfaden (ähnlich denen in URL Spezifikationen oder Dateisystemen), die eine Navigation in die Baumstruktur eines *XML*-Dokuments hinein zulassen. Zusätzlich wird ein Satz von Funktionen und Prädikaten definiert, der die Selektion bestimmter Pfade in Abhängigkeit von *XML*-Daten erlaubt (vgl. Abschnitt 7.3.3).

Die Syntax von *XPath* basiert nicht auf *XML*, sondern ist durch eine separate EBNF Grammatik definiert. Dies erlaubt insbesondere die einfache Verwendung von *XPath*-Ausdrücken als *XML*-Attributwerte (vgl. Abschnitt 7.7) und als Bestandteile von URL-Angaben.

2.2.4.2 *XSLT*

XSLT-Dokumente sind wohlgeformte *XML*-Dokumente zur Spezifikation der Transformation der Struktur eines *XML*-Dokuments. So können Transformationen definiert werden, die gültige Dokumente einer bestimmten DTD in gültige Dokumente einer anderen DTD verwandeln. Durch diesen Mechanismus ist es beispielsweise möglich, *XML*-Dokumente in *HTML*/*XHTML* [Wor00b] oder *WML*-Dokumente [Wir99] zu übersetzen, die direkt in den gängigen Web-Browsern bzw. auf WAP-fähigen Mobiltelefonen dargestellt werden können. In Anhang A.9 ist exemplarisch ein Ausschnitt aus einem *XSL*-Dokument zu sehen, das bestimmte *XML* -Dokumente des INSIGHT-Systems nach *HTML* übersetzt.

Jedes *XSLT*-Dokument besteht aus einem Satz *XPath*-basierter *Template Rules*, wobei jede *Template Rule* aus einem *Pattern* und einem *Template* besteht. Ein Pattern dient als Vorlage zur Selektion bestimmter Teile des Quelldokuments, das Template gibt die neue Struktur der selektierten Teile im Ergebnisdokument an.

2.2.4.3 *XSL* Formatting Objects

Dieser Teil der *XSL*-Spezifikation verwandelt *XML*-Dokumente (die evtl. durch eine vorherige *XSLT*-Transformation erzeugt wurden) in typographische Information für Textsatzformate wie beispielsweise PDF oder PostScript. Konkret wurde eine definierte Menge von sog. *Formatting Objects* festgelegt, die jeweils einen bestimmten typographischen Aspekt repräsentieren (z.B. Schriftgrößen, Zeilenumbruch oder Papiergröße). Diese *Formatting Objects* können einzelnen Teilbäumen des Quelldokuments zugeordnet werden und so die Formatierung dieser *XML*-Fragmente festlegen.

Die offiziellen Dokumente, die das W3C auf ihrer Homepage anbieten (z.B. der *XML*-Standard), sind i.d.R. in mehreren Formaten (*XML*, *HTML*, PDF ...) verfügbar, wobei die einzelnen Formate jeweils aus demselben *XML*-Quelldokument und entsprechenden *XSL*-Dokumenten generiert werden. Auch hier zeigt sich, dass sich durch Trennung von Inhalt und Layout leicht durchgängige Layout-Standards (z.B. zur Erzielung einer Corporate-Identity) durchsetzten lassen, da das Layout für jedes Format genau einmal definiert wird und dann auf mehrere Dokumente angewendet werden kann.

2.2.5 Eigenschaften von *XML* und *XSL*

Dieser Abschnitt fasst die wichtigsten Eigenschaften von *XML* und *XSL* kurz zusammen (vgl. auch [Bos97, CVM99]).

Plattformunabhängigkeit Da *XML* ein textbasiertes Format mit explizit ausgewiesener Zeichenkodierung ist, lassen sich *XML*-Dokumente i.d.R. problemlos auf unterschiedliche Rechnerplattformen übertragen. Zusätzlich ist *XML* sehr einfach zu parsen und daher auch unabhängig von spezifischer Software (z.B. von Autorensystemen). Mit der *Document Object Model* Spezifikation steht ein standardisiertes Datenmodell zur Verfügung, das eine plattformunabhängige rechnergestützte Bearbeitung von *XML*-Dokumenten ermöglicht.

Offenheit *XML* wurde als flexibles und offenes Format definiert, das prinzipiell beliebige Arten von Information in hierarchisch strukturierter Form repräsentieren kann[4]. Im Rahmen dieser Arbeit wird es beispielsweise u.a. dazu eingesetzt, industrielle Feldbusgeräte samt deren Schnittstellen zu beschreiben und die entsprechenden Prozessdaten aufzunehmen.

Verbreitung *XML* ist ein offener Standard, der sich in den letzten Jahren in den Bereichen Internet-basierte Informationssysteme und Portale stark durchgesetzt hat. Dies hat zum Einen zur Folge, dass durch *XML* kodierte Information an Wert gewinnt, da sie von vielen Seiten verarbeitet werden kann und zum Anderen, dass

[4]Eine hierarchische Strukturierung von Information erscheint allerdings nicht für alle Daten gleich natürlich.

allgemeine *XML*-Werkzeuge (Editoren, Editor-Generatoren, Datenbanken, Parser ...) und Programmierumgebungen in zunehmendem Maß verfügbar werden.

Trennung von Struktur und Darstellung *XML*-Dokumente enthalten selbst keine Informationen über die Darstellung der enthaltenen Information. Formatierungsinformationen lassen sich *XML*-Dokumenten über Referenzen auf entsprechende Style-Sheets (z.B. *XSL*-Dokumente) flexibel zuordnen. Je nach verwendetem Style-Sheet können unterschiedliche Sichten auf das Dokument generiert werden, die bestimmte Informationen aufbereiten, aggregieren oder verbergen. Zusätzlich lässt sich ein und dasselbe Dokument für unterschiedliche Ausgabemedien und Formate (z.B. Monitor, PDA, Handy, PDF, RTF ...) oder Benutzergruppen auf Anfrage dynamisch transformieren (*Single-Source*-Prinzip).

Bildung von Dokumentklassen Gültige *XML*-Dokumente enthalten oder referenzieren ihre eigene Strukturbeschreibung in Form von DTD-Spezifikationen und können daher als Instanzen der entsprechenden Dokumentklassen behandelt werden, was zum Einen den Zugriff auf Informationen beschleunigt, zum Anderen eine semantische Interpretation der Information erlaubt[5]. Insbesondere wenn es sich um standardisierte DTDs handelt, gewinnen die Dokumente dieser Dokumentklasse an Wert, da sie von unterschiedlichen Parteien auf unterschiedlichen Plattformen mit unterschiedlicher Software schnell bearbeitet und interpretiert werden können. Dieser Ansatz erleichtert insbesondere Business-to-Business-Kommunikation via Internet (vgl. z.B. [Ome01]).

Die Referenzierung einer DTD erlaubt zusätzlich eine automatische Validierung der Dokumente, die beispielsweise entscheiden kann, ob ein Dokument ein gültiges Datenbank-Update für ein bestimmtes Datenbankschema beschreibt.

Präziser und effizienter Datenzugriff Durch die explizite Strukturierung von *XML*-Dokumenten kann in Recherchesystemen die Präzision und Effizienz von textbasierten Anfragen gesteigert werden. Ein Benutzer kann in Anfragen Bezug auf die Dokumentstruktur nehmen und so den Kontext, in dem bestimmte Daten auftauchen, berücksichtigen (vgl. z.B. [NBY97]) und dadurch den Suchraum flexibel einschränken.

[5]Die semantische Information ist hierbei nicht Teil des *XML*-Dokuments, sondern wird erst über spezielle Kenntnis der entsprechenden DTD oder zusätzliche Informationen wie z.B. semantische RDF-Spezifikationen [BG00b, LS99, BLHL01] zugänglich.

2.3 Objektorientierte Software-Frameworks

Mehrere Konzepte dieser Arbeit, z.B. *DeviceInvestigator* (siehe Kapitel 5), *XJM_Eval* (siehe Kapitel 7) oder *JFCF* (siehe Kapitel 11), wurden als objektorientierte Software-Frameworks (im Folgenden abkürzend als *Frameworks* bezeichnet) implementiert.

> Object-oriented application frameworks are a promising technology for reifying proven software designs and implementations in order to reduce the cost and improve the quality of software. A framework is a reusable, "semi-complete" application that can be specialized to produce custom applications. [FS97]

Das zentrale Ziel von Frameworks ist somit das Konzept der Wiederverwendung von bewährten Softwarelösungen und zwar auf Design- und auf Implementierungsebene. Johnson definiert daher auch Frameworks als Synthese aus Softwarekomponenten (*Components*) und Software-Patterns [Joh97], wobei Komponenten die Wiederverwendung von Implementierung und Patterns die Wiederverwendung von Softwaredesign ermöglichen.

Ein wichtiges Paradigma von Frameworks ist das *Inversion of Control* [FS97] Konzept. Gemäß diesem Paradigma besteht die Spezialisierung eines Frameworks auf einen bestimmten Anwendungsfall im Wesentlichen aus der Implementierung spezieller Softwareschnittstellen, die jeweils von Framework-Seite aus als Reaktion auf bestimmte Ereignisse aufgerufen werden. Den aktiven Part übernimmt hierbei also das Framework, und der Programmierer spezifiziert primär das Verhalten des Systems in bestimmten Situationen. Die einzelnen Methoden dieser Schnittstellen werden auch als *Hook*- oder *Callback*-Methoden bezeichnet.

Die Schwierigkeit der Erstellung von Frameworks liegt vor allem in der Auswahl und Definition dieser Callback-Schnittstellen und der dadurch definierten Punkte hoher Flexibilität (Hot Spots), da i.d.R. ein Trade-off besteht zwischen Anpassbarkeit und Wiederverwendung (vgl. [DMNS97]). Je flexibler ein Framework angelegt ist, desto geringer ist der Grad der Wiederverwendung von Implementation. Zusätzlich steigt mit zunehmender Flexibilität eines Frameworks auch die Komplexität der Einarbeitung der Programmierer in das Framework.

Beispiele für weit verbreitete (und komplexe) Frameworks sind die *Microsoft Foundation Classes* (MFC) als der de facto Standard zur Entwicklung von nativen Windows Benutzerschnittstellen oder die *Java Swing* Bibliothek.

2.3.1 *XML*-basierte Software-Frameworks

I.d.R. werden mindestens zwei Arten von Frameworks unterschieden [FS97]:

1. White-Box-Frameworks
2. Black-Box-Frameworks

White-Box-Frameworks Dieser Ansatz beruht auf objektorientierten Konzepten wie Vererbung und dynamischer Polymorphie. Wiederverwendung wird hierbei durch Ableiten von Basisklassen und das Überschreiben von Callback-Methoden erzielt. Der Programmierer muss hierfür Einblick in große Teile der Klassenhierarchie haben.

Black-Box-Frameworks Black-Box-Frameworks basieren auf dem Ansatz der komponentenbasierten Softwareentwicklung. Die einzelnen Komponenten werden als Black-Boxes durch das Framework miteinander über abstrakte Softwareschnittstellen verbunden und verwaltet. Der Programmierer muss lediglich die Schnittstellen kennen, um das Framework einsetzen zu können.

Konkrete Frameworks sind normalerweise eine Mischung aus beiden Ansätzen, wobei mit zunehmender Reife eines Frameworks oft eine allgemeine Zunahme an Black-Box-Konzepten beobachtet werden kann [BMA97].

***XML*-basierte Frameworks** Die im Rahmen dieser Arbeit entstandenen *XML*-basierten Java-Frameworks *DeviceInvestigator* und *XJM_Eval* (siehe Kapitel 5 und 7) erweitern den beschriebenen Framework-Ansatz um das Konzept der externen Framework-Konfiguration. Hierbei wird die tatsächliche Konstellation eines Black-Box-Frameworks über ein externes *XML*-Dokument spezifiziert, das die beteiligten Softwarekomponenten jeweils über eine URL und den Klassennamen identifiziert und die einzelnen Callback-Methoden samt Übergabeparameter festlegt. Die konkrete Framework-Instanz entsteht hierbei erst während der Ausführung, zur Laufzeit.

Der Hauptvorteil dieses Ansatzes besteht darin, dass die Konfiguration, einschließlich der Komponentenauswahl und Definition der Hook-Methoden einer Framework-Anwendung, zur Laufzeit, d.h. ohne erneute Kompilierung des Systems, erfolgen kann.

Dieser Ansatz ist jedoch auf Programmiersprachen beschränkt, die eine Analyse der vorhandenen Methoden einer zur Compile-Zeit unbekannten Klasse erlauben, was beispielsweise auf C und C++ zunächst nicht zutrifft. Weiterhin geht Typsicherheit verloren, da die Verträglichkeit von Parametern und Methoden nicht mehr vom Compiler überprüft werden können (vgl. Abschnitt 7.9.1).

Teil I

INSIGHT

Kapitel 3

Einleitung

3.1 Überblick

INSIGHT ist eine offene, plattformunabhängige Infrastruktur zur Realisierung von Informationssystemen, die eine Einbindung von Geräten und Anlagen beliebiger Art in das World Wide Web realisieren. Die gerätetypspezifischen Software-Anteile können über eine generische Schnittstelle auf einfache Weise in die Infrastruktur eingeklinkt und somit in entsprechende Informationssysteme integriert werden. Die logische Einbindung der Geräte geschieht hierbei über den Zugriff auf einzelne oder mehrere Geräteparameter. Unter einem Parameter kann hierbei eine beliebige Eigenschaft eines Gerätes verstanden werden, die über eine digitale Erfassung zugänglich ist. Ein Geräteparameter kann beispielsweise die momentane Drehgeschwindigkeit eines Motors widerspiegeln oder das momentane Bild einer Digitalkamera. Ein Parameter liefert damit einen Teil des Gerätezustands zu einem bestimmten Zeitpunkt, wobei die Veränderung/Ableitung eines Parameters über der Zeit ebenfalls durch einen entsprechenden Parameter explizit zur Verfügung gestellt werden kann, also z.B. die Beschleunigung eines Motors.

Parameter haben hierbei bestimmte Eigenschaften wie z.B. Zugriffsrechte (lesbar, veränderbar, konstant ...) und Maßeinheit. Der schreibende Zugriff auf einen Parameter bewirkt eine Zustandsänderung des entsprechenden Gerätes und ruft i.d.R. ein verändertes Verhalten hervor, z.B. das Stoppen eines Motors.

Entscheidend ist hierbei, dass INSIGHT sämtliche Parameterwerte als serialisierte Zeichenketten in Form von *XML*-Fragmenten verwaltet. Jedem Parameterwert ist ein Parameterdatentyp in Form eines weiteren *XML*-Fragments, das die Bezeichnung des Datentyps enthält, zugeordnet. Die Erfassung der Gerätedaten sowie deren Transformation in eine textuelle Repräsentation wird an externe, gerätespezifische Software-Anteile delegiert und bleibt für INSIGHT transparent.

Dieser Ansatz bewirkt, dass die INSIGHT-Infrastruktur völlig unabhängig von gerätespezifischen Binärkodierungen der Datenformate (z.B. *little-endian* vs. *big-endian*) realisiert werden kann. Wie in dem obigen Beispiel bereits angedeutet, gibt es daher auch hinsichtlich der Datentypen keine grundsätzlichen Einschränkungen, so dass integrale Datentypen (ganze Zahlen, Fließkommazahlen usw.) ebenso erfasst werden können wie Video- oder Audiodaten. Die Verfügbarkeit einzelner Parameter kann hierbei naturgemäß jedoch sehr unterschiedlich sein. Manche Parameter können zu beliebigen Zeitpunkten gelesen/geschrieben werden, andere werden beispielsweise nur bei einer Zustandsänderung unter potentiellen Interessenten publiziert.

3.2 Funktionalität

INSIGHT ermöglicht die voll konfigurierbare Erfassung von Parametersätzen von Geräten und Anlagen sowie deren Abbildung in ein einheitliches *XML*-Format (sog. *Snapshot*-Dokumente). Die Konfiguration der Parametererhebung wird ebenfalls durch ein *XML*-Dokument spezifiziert. Dieses Dokument enthält sämtliche Informationen über die Anlage in strukturierter Form, d.h. Informationen über die Anlage an sich, über die darin enthaltenen Geräte, deren Parameter und deren Zuordnung zu bestimmten Hardwarezugriffsklassen, die den Gerätezugriff realisieren. Dieses Dokument kann mit allgemeinen oder speziellen *XML*-Werkzeugen bequem editiert und angepasst sowie zum Teil auch automatisch generiert werden. Es kann weiterhin komplett oder in Teilen (z.B. einzelne Gerätebeschreibungen) über die integrierte Datenbankanbindung persistent gespeichert oder für entfernte Benutzer in unterschiedlichen Sichten visualisiert und interaktiv zugänglich gemacht werden.

Die *Snapshot*-Dokumente können sowohl zyklisch als auch auf spezielle Anfragen hin generiert werden. Bei einer zyklischen Generierung werden die einzelnen Dokumente i.d.R. in einer Datenbank abgelegt, um das Verhalten der Anlage über der Zeit überwachen und analysieren zu können. Explizit angefragte Dokumente können beispielsweise zu entfernten Benutzern übertragen und dort visualisiert werden. In diesem Fall liefert die Benutzerschnittstelle eine aus dem *Snapshot*-Dokument und dem entsprechenden Konfigurationsdokument gewonnene, aussagekräftige Einsicht in den momentanen Prozesszustand der Anlage.

Aufgrund der für *XML* typischen Trennung von Information und Layout kann die Art der Informationspräsentation flexibel gehandhabt werden. Die INSIGHT-Benutzerschnittstellen stellen demnach auch mehrere dedizierte Sichten auf die Daten zur Verfügung. Diese Sichten betonen und/oder aggregieren bestimmte

Aspekte des momentanen Zustands, während sie andere Aspekte verbergen. Diese Funktionalität ist wichtig, um die relevanten Informationsanteile aus der eventuellen Fülle an momentan unwichtiger Information herauszufiltern.

Mit der beschriebenen Funktionalität unterstützt die INSIGHT-Infrastruktur bereits eine geräteunabhängige, Internet-basierte Fernwartung von Geräten und Anlagen. Von beliebigen Orten innerhalb des Internet kann ein konfigurierbarer Teil des Anlagenzustands mit einem herkömmlichen Web-Browser flexibel visualisiert werden. Zusätzlich kann auf weitere Online-Ressourcen wie z.B. Handbücher und Produktkataloge zugegriffen werden. Ein Benutzer kann die aktuellen Parameterwerte, insbesondere auch Fehlerregister, die ebenfalls als Parameter abgebildet werden können, analysieren und aufgrund seines Expertenwissens u.U. eine Diagnose stellen. Weiterhin ist es möglich, durch schreibenden Zugriff auf einzelne Parameter und erneute Abfrage des Anlagenzustands das reaktive Verhalten der Anlage zu testen, um potentielle Fehlerdiagnosen zu untermauern oder zu verwerfen.

Um das für eine Fehlerdiagnose notwendige Expertenwissen zu reduzieren, wurde mit *XJM_Eval* eine auf Ähnlichkeitsmessung beruhende Möglichkeit zur automatisierten Diagnoseunterstützung geschaffen. *XJM_Eval* ist ein *XML*-basiertes Software-Framework zur Integration von Java-Logik in *XML*-Querysprachen. Es ermöglicht die symmetrische Selektion bestimmter *XML*-Fragmente aus zwei zu vergleichenden *XML*-Dokumenten und deren flexible Abbildung auf frei wählbare Java-Klassen, die eine Abstandsfunktion für diese speziellen *XML*-Fragmente implementieren können. Die Definition der *XML*-Selektionen und die Abbildung auf Java-Klassen kann durch die *XML*-Applikation der *XML to Java Mapping Language* (*XJML*) in Form von *XML*-Dokumenten spezifiziert und verwaltet werden. *XJM_Eval* ist dann dazu in der Lage, basierend auf einem *XJML*-Dokument eine Suche nach dem nächsten Nachbarn (*Nearest Neighbor Search*) für ein *XML*-Dokument in Bezug auf eine Menge von *XML*-Dokumenten durchzuführen. Diese Menge von *XML*-Dokumenten kann hierbei ein gefiltertes Verzeichnis des Dateisystems oder die Ergebnismenge einer Anfrage an eine *XML*-Datenbank sein.

Eine automatische Diagnoseunterstützung ist mit *XJM_Eval* dann möglich, wenn für einen unbekannten Fehlerzustand, kodiert durch ein *XML*-Dokument, ein bezüglich einer zu wählenden *XJML*-Abstandsfunktion ähnlicher oder sogar der „gleiche“ Fehlerzustand von INSIGHT erfasst und in der Datenbasis abgelegt wurde. Da Abstandsfunktionen durch *XJML*-Dokumente spezifiziert sind, können diese zum Einen flexibel angepasst, zum Anderen ebenfalls in *XML*-Datenbanken gehalten und zur Verfügung gestellt werden. Es können schnell spezialisierte Abstandsfunktionen entwickelt und verwaltet werden, die für bestimmte Arten von Fehlern, z.B. Fehler, die auf Materialverschleiß beruhen, optimiert sind. Wenn die

Abstandsfunktionen in einer Datenbank gehalten werden, kann des Weiteren auch über die Abstandsfunktionen recherchiert werden[1].

XJM_Eval erlaubt grundsätzlich die flexible Bestimmung der Ähnlichkeit zweier beliebiger *XML*-Dokumente und ist damit nicht auf die Verwendung innerhalb von INSIGHT beschränkt. Das Verpacken von selektierten *XML*-Fragmenten in frei wählbare Java-Klassen ist allerdings besonders für durch INSIGHT erfasste Dokumente vorteilhaft, da diese Klassen eine entsprechende Rückführung der für INSIGHT typischen textuellen Repräsentation von Parameterwerten in gerätespezifische Datenformate vornehmen können, bevor Parameterwerte auf datentypspezifische Weise miteinander verglichen werden.

3.3 Architektur

INSIGHT ist eine offene, plattformunabhängige Infrastruktur zur Realisierung von Informationssystemen, die eine Einbindung von Geräten und Anlagen in das World Wide Web realisieren. INSIGHT wurde als *XML*-basiertes Java-Software-Framework realisiert (vgl. Abschnitt 2.3.1) und erlaubt die *XML*-basierte Integration von gerätetypspezifischen Hardwarezugriffsklassen zur Laufzeit.

Abbildung 3.1 gibt einen Überblick über die einzelnen Komponenten von INSIGHT: Die *Device Management Markup Language* (*DeMML*), die *DeviceInvestigator*-Komponente zur generischen Gerätedatenaggregation, den INSIGHT-Server, den Java-Applet-Client, die HTTP-Client-Schnittstelle, das Datenbank-Backend und das *XML*-Recherchewerkzeug *XJM_Eval*.

Im Folgenden werden die einzelnen Komponenten kurz vorgestellt und eine kurze Beschreibung der Arbeitsweise des INSIGHT-Systems gegeben, bevor die individuellen Komponenten in entsprechenden Kapiteln ausführlich erläutert werden.

INSIGHT benutzt die *XML*-Applikation *Device Management Markup Language* (*DeMML*) als universelles Datenformat zur Systemkonfiguration und als Datenprotokoll für die Client/Server-Kommunikation. Sämtliche innerhalb INSIGHT relevanten Daten werden als *DeMML*-Dokumente erfasst, verschickt bzw. abgespeichert. Im Einzelnen werden in einem *DeMML*-Dokument folgende Informationen verwaltet:

1. Anlagenbezogene Informationen
2. Gerätebezogene Informationen
3. Parameterbezogene Informationen

[1]Es ist es auch möglich, *XJM_Eval* selbst zum Auffinden geeigneter Abstandsfunktionen zu verwenden.

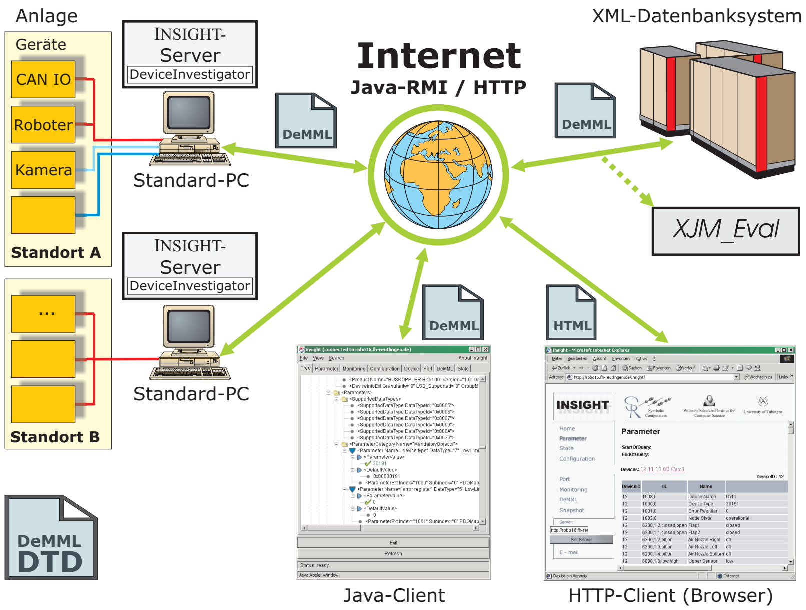

Abbildung 3.1: Das INSIGHT-System

4. Parameterwerte und Zeitstempel

Unter einer Anlage wird in diesem Kontext die Verbindung mehrerer Geräte zu einem Gerätekomplex zur Erfüllung einer oder mehrerer Aufgaben verstanden. Eine Anlage kann beispielsweise eine industrielle Produktionsanlage, ein medizinisches Gerät oder auch ein Fahrzeug sein. Die anlagenbezogenen Informationen umfassen Informationen, die eine Anlage als Ganzes betreffen und identifizieren (z.B. Daten über den Standort der Anlage und Kontaktmöglichkeiten zum zuständigen Personal).

Die einzelnen Geräte einer Anlage werden durch die gerätebezogenen Informationen beschrieben. Es werden z.B. Versionsnummern der Hard- und Software der einzelnen Geräte und Gerätebeschreibungen abgelegt. Insbesondere werden für jedes Gerät die zugänglichen Parameter beschrieben. Zum Einen werden die einzelnen Parameterattribute (Name, ID, Datentyp usw.) angegeben, um die Parameter an sich zu beschreiben. Zum Anderen wird aber auch für jeden Parameter spezifiziert, welche Java-Klasse für den Zugriff auf diesen Parameter zuständig sein soll. Diese Delegation von Gerätezugriffszuständigkeit an dynamisch erzeugte Java-Klassen ist ein zentrales Konzept innerhalb von INSIGHT und die Basis für die Offenheit des Systems. Auf *XML*-Seite bedient sich *DeMML* für die Abbildung von *XML*-Fragmenten auf Java-Klassen der *XML to Java Mapping Language* (*XJML*), die im Rahmen von *XJM_Eval* ausführlich erläutert wird (siehe Abschnitt 7.7).

Da *DeMML*-Dokumente auch Prozesszustände repräsentieren können, können neben den Parameterbeschreibungen auch die zu einem bestimmten Zeitpunkt gültigen Parameterwerte gespeichert werden. Die Parameterwerte können sowohl direkt innerhalb der zugehörigen Parameterbeschreibungen abgelegt werden als auch in einem separaten *DeMML*-Dokument, das nur den dynamischen Datenanteil enthält und die entsprechende Metainformation über Attributreferenzen zuordnet. Dieses schlankere Format erlaubt das effiziente Verwalten von dynamischen Prozesszuständen, die sich auf dieselbe Anlagenbeschreibung beziehen.

Die *DeviceInvestigator*-Komponente ist für die Analyse und die dynamische Generierung von *DeMML*-Dokumenten verantwortlich. In einer Initialisierungsphase analysiert *DeviceInvestigator* die *DeMML*-Konfiguration einer Anlage und erstellt entsprechende Datenstrukturen zum effizienten Zugriff auf die spezifizierten Geräteparameter der einzelnen Geräte. Hierzu werden insbesondere die erforderlichen Hardwarezugriffsklassen dynamisch geladen und instanziiert. *DeviceInvestigator* unterstützt zum Einen den schreibenden Zugriff auf einzelne Parameter, zum Anderen die Generierung von *Snapshot*-*DeMML*-Dokumenten, die jeweils die aktuellen Werte sämtlicher entsprechend spezifizierter Parameter enthalten. Die Generierung von *Snapshot*-Dokumenten kann entweder zyklisch erfolgen, um eine Anlage zu überwachen, oder sie kann auf eine externe Anfrage hin geschehen. Die generierten Dokumente werden von *DeviceInvestigator* im Dateisystem abgespeichert, per HTTP in eine entfernte *XML*-Datenbank geschrieben oder über entsprechende Schnittstellen anderen Java-Objekten zur Verfügung gestellt.

DeviceInvestigator kann durch den INSIGHT-Server in ein Web-basiertes Informationssystem eingebunden werden. Der INSIGHT-Server erweitert die Funktionalität von *DeviceInvestigator* zu einem Information-Server, der von entfernten Clients über Java *Remote Method Invocation* (RMI) [Sun98, Dow98] angesprochen werden kann. Er unterstützt im Wesentlichen Schnittstellen zum Abfragen des momentanen Systemzustandes, zum schreibenden Zugriff auf einzelne Parameter und zum Zugriff auf gerätespezifische Managementfunktionalität. Systemzustände und Konfigurationsdaten werden hierbei grundsätzlich als serialisierte *DeMML*-Dokumente verschickt. Zusätzlich synchronisiert der INSIGHT-Server konkurrierende Client-Anfragen über ein Zeitscheibenverfahren.

Das INSIGHT-System sieht zwei unterschiedliche Benutzerschnittstellen zum entfernten Zugriff auf Gerätedaten vor: Eine Java-Applet-Schnittstelle und eine

HTTP-Schnittstelle. Der INSIGHT-*Applet-Client* kommuniziert direkt mit dem INSIGHT-Server über Java-RMI und transformiert die vom INSIGHT-Server gelieferten *DeMML*-Dokumente in unterschiedliche interaktive GUI-Komponenten. Die HTTP-Schnittstelle kann direkt von beliebigen Web-Browsern angesprochen werden. Bei diesem Verfahren werden *DeMML*-Dokumente bereits auf der Server-Seite über die Referenzierung von entsprechenden Style-Sheets in *HTML*-Dokumente transformiert, die als unterschiedliche Sichten auf die in *DeMML* kodierte Information direkt in der *HTML*-Engine eines Browser visualisiert werden können.

Zur persistenten Speicherung von *DeMML*-Dokumenten benutzt INSIGHT primär das native *XML*-Datenbanksystem *Tamino* der Software AG. Dieses System erhält bei der Abbildung auf das datenbankinterne Datenmodell die hierarchische Struktur der *XML*-Dokumente. Es wird insbesondere keine Abbildung auf ein relationales Datenmodell vorgenommen, was eine verbesserte Performance erhoffen lässt. Als Grundlage für das datenbankinterne Datenmodell kann hierbei eine *XML*-DTD verwendet werden, die durch zusätzliche Informationen über Datentypen und Datenbank-Indexstrukturen zu einem sogenannten Datenbankschema erweitert werden kann. Dieses Datenbankschema basiert auf dem *XML Schema* Standard [Fal01, TBMM01, Wor01d] des W3C. Das entworfene *DeMML*-Schema konfiguriert das einheitliche Datenbanklayout für die INSIGHT-Infrastruktur unabhängig von der Systemkonfiguration oder den beteiligten Gerätetypen.

Als Rechercheschnittstelle bietet *Tamino* einen *XPath*-basierten Ansatz [CD99], der den direkten Zugriff auf einzelne *XML*-Fragmente zulässt. Als Client-Schnittstelle unterstützt *Tamino* zusätzlich zu ODBC [Mic97] primär einen HTTP-basierten Zugang. Dieser Ansatz hat den Vorteil, dass jeder gewöhnliche Web-Browser automatisch als voll funktionsfähiger Datenbank-Client verwendet werden kann, was in einem Internet-basierten Informationssystem besonders vorteilhaft ist.

Als Recherche-Komponente innerhalb von INSIGHT wird das *XJM_Eval*-System verwendet. Dieses erlaubt die flexible auf *XML* und Java basierende Definition und Auswertung von Ähnlichkeitsmaßen auf beliebigen *XML*-Dokumenten. *XJM_Eval* selbst ist nicht an eine Verwendung innerhalb von INSIGHT gebunden und wurde daher auch nicht speziell in das System integriert, sondern als externe Komponente entwickelt und nur lose integriert (siehe Abschnitt 7.10).

3.4 Nutzungsartenanalyse

Nutzungsarten (*Use Cases*) beschreiben die Abfolge von Ereignissen, die ein externer Benutzer zum Erreichen eines bestimmten Zieles in einem System hervor-

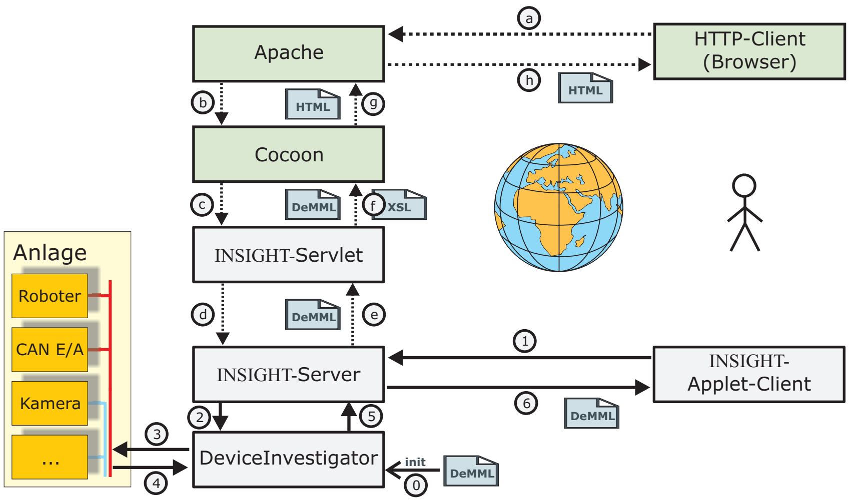

Abbildung 3.2: Use-Case 1: Entfernte Zustandsabfrage

ruft [JC92]. Sie geben damit Aufschluss über die grundlegende Arbeitsweise des Systems und das Zusammenspiel der einzelnen Systemkomponenten. In diesem Abschnitt werden die zentralen high-level Use Cases von INSIGHT vorgestellt, um einen detaillierteren Einblick in die Funktionalität des Gesamtsystems zu geben. Die einzelnen Vorgänge sind teilweise vereinfacht dargestellt und werden dann in den entsprechenden Abschnitten ausführlich behandelt.

3.4.1 Use-Case 1: Entfernte Zustandsabfrage

Abbildung 3.2 zeigt das Ablaufdiagramm einer entfernten Zustandsabfrage an ein INSIGHT-System. Da zwei unterschiedliche Zugriffskonzepte (über den Applet-Client und über die HTTP-Schnittstelle) unterstützt werden, gibt es auch zwei unterschiedliche, sich zum Teil überlappende Kommunikationswege. Der Weg über den Applet-Client ist mit den Ziffern 1-6 bezeichnet, der Weg über die HTTP-Schnittstelle mit den Buchstaben a-h. Für den externen Benutzer besteht dieser Use-Case nur aus der entfernten Abfrage des momentanen Systemzustandes und der Visualisierung des Resultats, wobei die Visualisierung an separater Stelle erläutert wird.

Schritt 0: Dieser Schritt beinhaltet die Konfiguration des INSIGHT-Servers bzw. der *DeviceInvestigator*-Komponente und wird nur einmalig beim Start des Systems durchgeführt. *DeviceInvestigator* liest hierbei ein *DeMML*- Konfigurationsdokument, das sämtliche Informationen über die Anlagenkonfiguration und deren Zugangswege enthält. Wann immer sich ein Client beim INSIGHT-Server anmeldet, wird neben einem Authentifizierungsschlüssel als erstes dieses Konfigurationsdokument an den Client übertragen, um auch den Client über die aktuelle Systemkonfiguration zu informieren. Um zukünftige Client-Anfragen effizient behandeln zu können, legt *DeviceInvestigator* in der Initialisierungsphase zusätzlich diverse Indexstrukturen zum schnellen Zugriff auf die spezifizierten Geräteparameter an.

Schritt 1: Die Zustandsanfrage des Applet-Clients besteht aus einem entfernten Java-RMI-Aufruf an den INSIGHT-Server. Als Parameter wird nur der Authentifizierungsschlüssel (siehe Abschnitt 6.1.1) des Clients übertragen.

Schritt 2: Der INSIGHT-Server prüft die Gültigkeit des Identifikationsschlüssels und reicht die Anfrage an *DeviceInvestigator* weiter.

Schritt 3: *DeviceInvestigator* benutzt seine Indexstrukturen, um gemäß seiner Konfiguration die spezifizierten Parameterwerte aus den Geräten zu lesen. Dies geschieht durch Aufruf entsprechender Methoden auf generischen Schnittstellen von spezialisierten Hardwarezugriffsklassen, die von *DeviceInvestigator* in der Initialisierungsphase dynamisch instanziiert wurden. Der Hardwarezugriff auf die einzelnen Parameter kann über unterschiedliche Kommunikationsmedien (z.B. Feldbusse, Ethernet usw.) und Protokolle stattfinden.

Schritt 4: Die Ergebnisse der einzelnen Parameteranfragen werden von *DeviceInvestigator* eingesammelt und in eine neue *Snapshot*-DOM-Struktur verpackt. Diese Datenstruktur wird in ein *DeMML*-Dokument, kodiert als Java-Unicode-String [The96], serialisiert.

Schritt 5: Der *DeMML*-String wird an den INSIGHT-Server als Ergebnis des Aufrufs aus Schritt 2 zurückgegeben.

Schritt 6: Der INSIGHT-Server gibt den *DeMML*-String an den Applet-Client zurück. Der Client rekonstruiert mit Hilfe eines *XML*-Parsers die entsprechende DOM-Datenstruktur aus dem *DeMML*-String und visualisiert das Ergebnis für den Benutzer durch unterschiedliche GUI-Komponenten.

Schritt a: Der Browser schickt einen durch einen *HTML*-Link spezifizierten HTTP-*Get-Request* an den Web-Server (Apache). Dieser Request enthält unter anderem als URL-Parameter kodierte Informationen über die Quelle (*Producer*) des angefragten Dokuments (siehe Schritt c) und das zu verwendende Style-Sheet (*Style*).

Schritt b: Der Web-Server untersucht die angefragte URL und reicht aufgrund seiner Konfiguration den Request weiter an das Cocoon-Servlet [Apab], das *XSLT*-Transformationen durchführen kann.

Schritt c: Cocoon analysiert den *Producer* Parameter und reicht daraufhin aufgrund seiner Konfiguration die Anfrage an das INSIGHT-Servlet weiter.

Schritt d: Dieser Schritt entspricht im Wesentlichen Schritt 1, abgesehen davon, dass die Anfrage lokal vom INSIGHT-Servlet aus stattfindet und nicht von einem entfernten Client aus. Anschließend werden die Schritte 2-5 in der oben beschriebenen Art und Weise abgearbeitet.

Schritt e: Der INSIGHT-Server gibt den erhaltenen *Snapshot-DeMML*-String an das INSIGHT-Servlet zurück.

Schritt f: Das INSIGHT-Servlet analysiert den *Style*-Parameter der HTTP-Anfrage, fügt eine dementsprechende Referenz auf ein *XSL*-Dokument in den *DeMML*-String ein und gibt das Ergebnis an Cocoon zurück.

Schritt g: Cocoon rekonstruiert den DOM-Baum des *DeMML*-Strings mit Hilfe eines *XML*-Parsers, lädt das referenzierte *XSL*-Dokument und führt die darin enthaltenen Transformationen auf dem DOM-Baum aus. Das Resultat dieser Transformationen ist ein *HTML*-Dokument, das je nach referenziertem *XSL*-Dokument eine spezialisierte Sicht auf den Systemzustand darstellt. Dieses *HTML*-Dokument wird an den Web-Server zurückgegeben.

Schritt h: Der Web-Server liefert das *HTML*-Dokument als Antwort auf die *Get*-Anfrage an den Browser.

3.4.2 Use-Case 2: Anlagen-Monitoring

Der Begriff *Monitoring* meint hier das zyklische Erfassen und Abspeichern von Prozesszuständen einer laufenden Anlage. Diese Daten können für unterschiedliche Zwecke verwendet werden, so z.B. im Fehlerfall zur Analyse der Vorgeschichte eines Fehlerzustandes oder zur Dokumentation der Produktionsvorgänge und Qualitätssicherung (z.B. im *Health Care* Bereich).

DeviceInvestigator kann unabhängig vom INSIGHT-Server als selbstständige Monitoring-Komponente betrieben werden, um *DeMML*-kodierte Prozesszustände im Dateisystem oder in einer entfernten *XML*-Datenbank abzuspeichern. Abbildung 3.3 zeigt den Ablaufplan dieses Use-Cases. Für den externen Benutzer besteht dieser Use-Case aus der Anpassung der Monitoring-Konfiguration und der Initiierung des Monitoring-Vorgangs.

Schritt 1: Falls gewünscht, kann der Benutzer das *DeMML*- Konfigurationsdokument, das mit dem Konfigurationsdokument aus Use-Case 1 identisch sein kann, anpassen. Es können z.B. bestimmte Parameter stummgeschaltet werden, um das Datenaufkommen zu reduzieren. Die Bearbeitung der Konfiguration kann entweder mit generischen *XML*-Editoren oder dem INSIGHT-*Applet-Client* bzw. dem *ConfigurationWizard* (siehe Abschnitt 4.8) erfolgen.

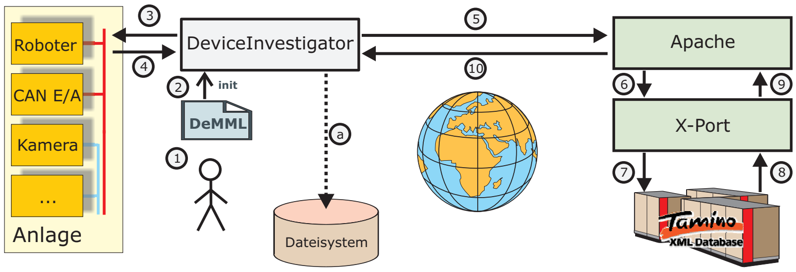

Abbildung 3.3: Use-Case 2: Anlagen-Monitoring

Schritt 2: Der Benutzer startet den Monitoring-Vorgang, woraufhin sich die *DeviceInvestigator*-Komponente entsprechend Schritt 0 von Use-Case 1 initialisiert. Zusätzlich kann eine Datenbank-Basis-URL angegeben werden, die eine *Tamino*-Datenbank identifiziert, in die die generierten Dokumente abgelegt werden sollen.

Schritt 3 und 4: *DeviceInvestigator* triggert selbstständig den Methodenaufruf zum Erfassen des Systemzustandes mit einer einstellbaren Frequenz. Die Hardwarezugriffe laufen analog zu Schritt 3 und 4 von Use-Case 1.

Schritt 5: Jedes Mal, wenn ein neuer Prozesszustand erhoben wurde, schreibt *DeviceInvestigator* mittels eines HTTP-*Post-Requests* den neuen *DeMML*-String auf eine von der angegebenen Basis-URL abgeleitete URL. Die Ableitung beruht auf *Tamino*-spezifischen Schlüsselwörtern, wie z.B. *define*, um ein Datenbank-*Insert* durchzuführen.

Schritt 6: Der entfernte Web-Server untersucht die Anfrage und reicht sie aufgrund seiner Konfiguration an die *X-Port*-Komponente der *Tamino*-Datenbank weiter.

Schritt 7: Die *X-Port*-Komponente leitet die Anfrage an den zuständigen *Tamino*-Server weiter, der letztendlich das *DeMML*-Dokument gemäß den entsprechenden URL-Parametern in die Datenbank einfügt.

Schritt 8-10: Die Erfolgs- oder Fehlermeldung des *Tamino*-Servers wird in Form eines *XML*-Dokuments an *DeviceInvestigator* zurückgeleitet.

Schritt a**:** Falls kein Zugriff auf eine entfernte Datenbank erwünscht oder möglich ist, kann *DeviceInvestigator* die erhobenen *DeMML*-*Snapshot*-Dokumente auch lokal in einem Verzeichnis des Dateisystems ablegen.

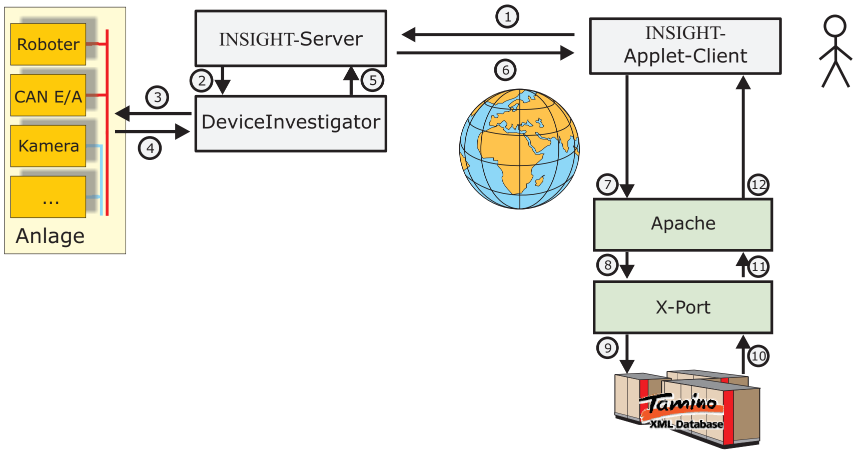

Abbildung 3.4: Use-Case 3: Interaktiver Datenbankzugriff

3.4.3 Use-Case 3: Interaktiver Datenbankzugriff

Dieser Use-Case beschreibt zunächst den lesenden und anschließend den schreibenden Zugriff auf eine *DeMML*-Datenbank mittels des INSIGHT-*Applet-Client*s. Abbildung 3.4 zeigt den Ablaufplan dieses Use-Cases.

Schritt 1-6: Diese Schritte sind analog zu den Schritten 1-6 aus Use-Case 1. Als Ergebnis ist der momentane Prozesszustand als *DeMML*-Dokument im Client verfügbar.

Schritt 7: Zunächst kann der Benutzer über die Angabe einer Basis-URL eine entfernte *Tamino*-Datenbank auswählen, in die er den Prozesszustand ablegen möchte. Er hat dann die Möglichkeit, Informationen über den Prozesszustand direkt in das *DeMML*-Dokument einzufügen, um beispielsweise auf externe, weiterführende Zustandsbeschreibungen (z.B. Fehlerdokumentationen) zu verweisen. Das auf diese Weise erweiterte Dokument wird daraufhin mittels eines HTTP-*Post-Requests* an den Web-Server übergeben.

Schritt 8-12: Diese Schritte sind analog zu den Schritten 6-9 aus Use-Case 2.

Der INSIGHT-Client unterstützt auch den Import von Prozesszuständen aus Datenbanken. Der Benutzer wählt hierzu eine entfernte *Tamino*-Datenbank und eine Anlagen-Identifikation. Die Anlagen-Identifikation ist ein Attribut, das in jedem *DeMML*-Dokument vorhanden ist. Der Benutzer bekommt daraufhin eine Liste aller verfügbarer Prozesszustände dieser Anlage in dieser Datenbank. Aufgelistet

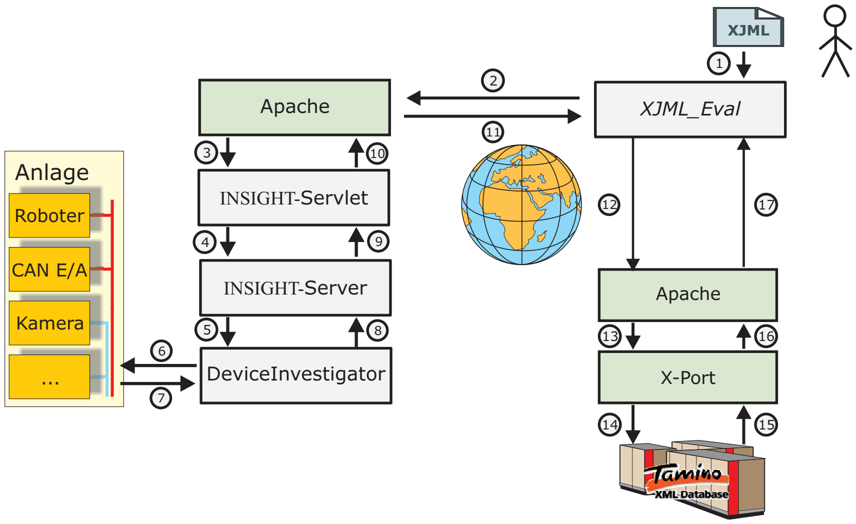

Abbildung 3.5: Use-Case 4: Fehlersuche

werden der Zeitstempel der Datenerhebung und die Zustandsbeschreibung. Der Benutzer kann einen Zustand aus der Liste auswählen, der dann über einen HTTP-*Get-Request* entsprechend Schritt 6-9 aus Use-Case 2 aus der Datenbank importiert und visualisiert wird.

3.4.4 Use-Case 4: Fehlersuche

Dieser Use-Case beschreibt den Einsatz von *XJM_Eval* zum Auffinden von ähnlichen Zuständen innerhalb von INSIGHT als Mittel zur entfernten Fehlerdiagnose. Abbildung 3.5 zeigt den Ablaufplan dieses Use-Cases.

XJM_Eval verwendet *XJML*-Dokumente zur Definition von Ähnlichkeitsmaßen. Innerhalb eines *XJML*-Dokuments wird das *XML*-Ähnlichkeitsmaß und der Suchraum als Dokumentenmenge spezifiziert. In diesem Use-Case wird das Anfragedokument über die Schritte 2-11 von einem entfernten INSIGHT-System geliefert und der Suchraum als Ergebnismenge einer Query an eine entfernte *Tamino*-Datenbank (Schritte 12-17) definiert. Als Ergebnis wird eine Referenz auf das Element des Suchraums gegeben, das dem Anfragedokument am „ähnlichsten“ ist.

Die Situation sei die folgende: Die entfernte Anlage arbeitet nicht korrekt. Für diese und ähnliche Anlagen existiert eine *DeMML*-Datenbasis von bekannten Fehlerzuständen, die über einen längeren Zeitraum gewachsen ist. Ein Ingenieur vor Ort teilt einem entfernten Fachmann, z.B. über Telefon oder E-Mail mit, dass die Anlage nicht korrekt arbeitet.

Schritt 1: Der externe Fachmann kann ein *XJML*-Dokument neu erstellen, auswählen oder anpassen, um ein für diese Art von Anlagen angemessenes Ähnlichkeitsmaß zu definieren. Da *XJML*-Dokumente *XML*-Dokumente sind, kann das *XJML*-Dokument auch aus einer zentralen *XML*-Datenbank, die *XJML*-Dokumente für unterschiedliche Anlagenarten und Anwendungsgebiete bereitstellt, importiert werden. Wie bereits erwähnt, werden der entfernte Systemzustand als Anfragedokument und die Datenbasis der Fehlerzustände als Suchraum definiert.

Schritt 2: Der Benutzer schickt mittels *XJM_Eval* einen HTTP-*Get-Request* an den entfernten Web-Server.

Schritt 3: Der Web-Server untersucht die angefragte URL und reicht aufgrund seiner Konfiguration den Request weiter an das INSIGHT-Servlet.

Schritt 4-9: Diese Schritte sind analog zu den Schritten d-e aus Use-Case 1.

Schritt 10: Das INSIGHT-Servlet reicht das erhaltene *DeMML*-Dokument an den Web-Server weiter.

Schritt 11: Der Web-Server beantwortet den HTTP-*Get-Request* mit dem erhaltenen *DeMML*-Dokument. *XJM_Eval* hat damit den aktuellen Fehlerzustand der entfernten Anlage als *XML*-Dokument vorliegen.

Schritt 12-17: Diese Schritte laufen ähnlich ab, wie die Schritte 7-12 beim lesenden Datenbank-Zugriff aus Use-Case 3. Konkret wird eine im *XJML*-Dokument enthaltene Datenbankanfrage per HTTP-*Get* auf der entfernten Datenbank ausgewertet. Das Resultat ist logisch gesehen ein *XML*-Resultatdokument[2], das die einzelnen Ergebnisdokumente als Teilbäume enthält. *XJM_Eval* hat damit auch den Suchraum für die Ähnlichkeitssuche vorliegen.

XJM_Eval sucht gemäß den Angaben des *XJML*-Dokuments (siehe Schritt 1) aus dem nun vorliegenden Suchraum das Dokument mit dem geringsten Abstand zum Anfragedokument heraus. Dieses Dokument und insbesondere die Zustandsbeschreibung (vgl. Schritt 7 aus Use-Case 3) kann im besten Fall bereits die Fehlerdiagnose inklusive Angaben über entsprechende Testmethoden enthalten oder andernfalls Hinweise für mögliche Fehlerursachen geben. Insbesondere wenn exakt der gleiche Fehlerzustand bereits in der Datenbank vorhanden ist und das verwendete *XJML*-Ähnlichkeitsmaß diesen identifiziert, kann die Fehlerdiagnose u.U. ohne zusätzliches Fachwissen gestellt werden.

[2]Tatsächlich wird das Anfrageresultat je nach Größe auf mehrere Dokumente verteilt, die durch einen Cursor-Mechanismus traversiert werden.

Kapitel 4

Device Management Markup Language (*DeMML*)

4.1 Überblick und Motivation

Die *Device Management Markup Language* (*DeMML*) ist ein universelles Datenformat zur plattformunabhängigen Repräsentation von Anlagenkonfigurationen und -zuständen sowie zur Benutzung als Datenprotokoll innerhalb einer verteilten Client/Server-Architektur. Im Einzelnen werden in *DeMML*-Dokumenten folgende Informationen verwaltet:

1. Anlagenbezogene Informationen
2. Gerätebezogene Informationen
3. Parameterbezogene Informationen
4. Parameterwerte und Zeitstempel (Prozesszustände)

Parameterwerte unterschiedlicher Geräte unterscheiden sich i.d.R. beträchtlich in Bezug auf ihre Zugangswege und Hardware-interne Darstellung (z.B. *little endian* vs. *big endian*). Diese Informationen werden den einzelnen Parameterbeschreibungen in Form von Referenzen auf Java-Klassen und -Methoden zugeordnet, die zum Einen den protokollspezifischen Zugriff auf die Hardware, zum Anderen die Serialisierung der Parameterwerte in flache Zeichenketten realisieren. Innerhalb von *DeMML*-Dokumenten können daher sämtliche Parameterwerte unabhängig von ihrer jeweiligen Hardware-internen Darstellung als serialisierte Zeichenketten in Form von *XML*-Fragmenten verwaltet werden. Dieser Ansatz erlaubt die

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE DeMML SYSTEM
  'http://www-sr.informatik.uni-tuebingen.de/Insight/DeMML.dtd'>
<DeMML>
  <SystemConfiguration>
    <FileInfo CreationTime="2001-03-19␣03:52:38.253"/>
    <SystemInfo>
      <Description>Configuration file for ...</Description>
      <Location Id="13" Country="Germany" Address="..." ServerIP="..."/>
      <Contact Id="0001" FirstName="Dieter" Surname="Buehler" Email="..."/>
    </SystemInfo>
    <Device DeviceId="0F">
      <Description>Lift IO (sensors + flaps)</Description>
      <DeviceInfo>
        <Vendor Name="Selectron" VendorId="0"/>
        <Product Name="DIOC711" Revision="1" ProductId="0" Version="1"/>
      </DeviceInfo>
      <Parameters>
        <ParameterCategory Name="MandatoryObjects">
          <Parameter Name="Error␣Register" DataType="5" AccessType="ro"
            ParameterId="1001,0" Monitoring="1" Mute="0">
            <ParameterValue>0</ParameterValue>
          </Parameter>
          <!-- ... -->
```

Listing 4.1: Ausschnitt aus einem einfachen *DeMML*-Konfigurationsdokument

Erstellung datentypunabhängiger Infrastrukturen (wie z.B. INSIGHT) zur homogenen Verwaltung ehemals heterogener Gerätedaten.

In einem Recherche-Kontext kann die Serialisierung der Parameterwerte über analoge Referenzen auf Java-Logik wieder umgekehrt und auf die tatsächlichen Parameterwerte zugegriffen werden. Realisiert werden diese Referenzen jeweils über das *XJM*-Konzept (siehe Kapitel 7), das eine dynamische Ausführung von weltweit eindeutig identifizierter Java-Logik erlaubt.

Listing 4.1 und 4.2 zeigen einen Ausschnitt eines sehr einfachen *DeMML*-Konfigurations- bzw. *Snapshot*-Dokuments. Die einzelnen Elemente der Beispieldokumente werden in entsprechenden Abschnitten ausführlich erläutert.

Entstanden ist *DeMML* ursprünglich aus der *CANopen Markup Language* (*CoML*) [Büh00, BG00c], die auf CANopen-basierte Feldbusanlagen beschränkt war. *CoML* entstand im Rahmen der Entwicklung des CANINSIGHT Gerätemanagementsystems [BK00], das ebenfalls auf die Verwendung von CANopen-Feldbusgeräten ausgelegt war. Zusammen mit der Abstraktion von *CoML* zu *DeMML* wurde CANINSIGHT zu INSIGHT generalisiert.

Für das Design von *DeMML* bedeutete dies zum Einen, die allen Gerätetypen gemeinsame Teilmenge an Gerätespezifikationsmöglichkeiten zu identifizieren und direkt in *DeMML* zu verankern, zum Anderen eine Möglichkeit zu schaffen, die

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE DeMML SYSTEM
  'http://www-sr.informatik.uni-tuebingen.de/Insight/DeMML.dtd'>
<DeMML>
  <Snapshot LocationId="13" StartOfQuery="2001-03-19␣10:52:53.321"
    EndOfQuery="2001-03-19␣10:52:54.703"
    SystemConfigurationCreationTime="2001-03-19␣03:52:38.253">
    <Description>Error State: sensor x offline ...</Description>
    <ScannedDevice DeviceId="0F">
      <LogEntry ParameterId="1001,0" Valid="1" Value="0"/>
      <LogEntry ParameterId="1002,0" Valid="1" Value="50000"/>
      <!-- ... -->
```

Listing 4.2: Ausschnitt aus einem *DeMML-Snapshot*-Dokument

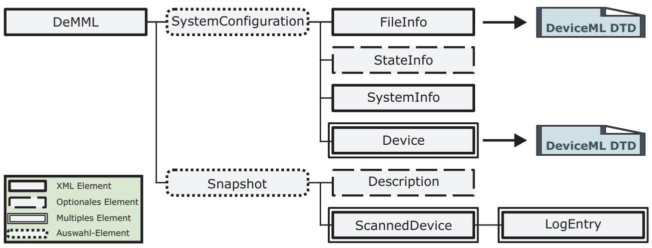

Abbildung 4.1: Ausschnitt aus der logischen Struktur eines *DeMML*-Dokuments

jeweiligen gerätetypbezogene Daten flexibel in *DeMML*-Dokumente integrieren, bzw. *DeMML* selbst erweitern zu können. Diese Flexibilität wird durch die Repräsentation von Parameterwerten als serialisierte Zeichenketten, durch die Flexibilität des DTD-Konzepts und durch eine flexible Modularisierung der *DeMML*-DTD in mehrere Teil-DTDs erreicht. Eine grundlegende Anforderung war hierbei jeweils, dass die erweiterten DTDs ohne Anpassung des INSIGHT-Systems verwendbar sein sollten.

4.2 Die *DeMML* Document Type Definition

In diesem Abschnitt wird die *DeMML*-DTD (siehe Anhang A.1) und damit die logische Struktur von *DeMML* beschrieben. Wie bereits erwähnt ist die *DeMML*-DTD modular organisiert und baut auf weiteren DTDs auf, die über entsprechende Entity-Deklarationen eingebunden werden (vgl. Zeile 8-18 von Listing A.1 auf Seite 229). Abbildung 4.1 zeigt einen Ausschnitt aus der logischen Struktur von *DeMML*.

Das Wurzelelement eines jeden *DeMML*-Dokuments ist das *DeMML*-Element. Dieses kann entweder ein *SystemConfiguration*- (vgl. Listing 4.1) oder ein *Snapshot*-Element (vgl. Listing 4.2) enthalten. Während das *SystemConfiguration*-Element für die Repräsentation von statischer Konfigurationsinformation ausgelegt ist, dient das *Snapshot*-Element zur Repräsentation von dynamischen Prozesszuständen. Im Folgenden werden *DeMML*-Dokumente, die ein *SystemConfiguration*-Element enthalten, als Konfigurationsdokumente und Dokumente, die ein *Snapshot*-Element enthalten, als *Snapshot*-Dokumente bezeichnet.

4.2.1 Repräsentation von Systemkonfigurationen

Das *SystemConfiguration*-Element

```
<!ELEMENT SystemConfiguration
   (FileInfo, StateInfo?, SystemInfo, (Device)*)>
```

Listing 4.3: Das *SystemConfiguration*-Element

Ein *SystemConfiguration*-Element enthält sämtliche Informationen, die zum Einbinden einer Geräteanlage in das INSIGHT-System notwendig sind. Im Einzelnen sind dies:

1. Anlagenbezogene Informationen
2. Gerätebezogene Informationen
3. Parameterbezogene Informationen

Die anlagenbezogenen Informationen werden durch ein entsprechendes *SystemInfo*-Element definiert. Informationen, die die einzelnen involvierten Geräte betreffen, werden für jedes Gerät in einem entsprechenden *Device*-Element zusammengefasst. Zusätzlich können über das *FileInfo*-Element und das optionale *StateInfo*-Element weitere Informationen über das *DeMML*-Dokument an sich bzw. die Default-Zustandsbeschreibung für spätere Zustandserhebungen angegeben werden. Insbesondere ist das *CreationTime*-Attribut des *FileInfo*-Elements (vgl. Lisiting 4.1, Zeile 6) dafür verantwortlich, unterschiedliche Konfigurationen einer Anlage voneinander zu unterscheiden.

Das *SystemInfo*-Element

```
<!ELEMENT SystemInfo (Description?, Location, (Contact)+)>
<!ELEMENT Location EMPTY>
<!ATTLIST Location Id CDATA #REQUIRED
   Country CDATA #REQUIRED
   Address CDATA #REQUIRED
```

```
    ServerIP CDATA #REQUIRED
    ServerName CDATA #IMPLIED>
```

Listing 4.4: Das *SystemInfo*-Element

Dieses Element enthält die folgenden Informationen:

1. *Description*: Anlagenbeschreibung in Klartext
2. *Location*: Identifikation der Anlage
3. *Contact*: Kontakt zum lokalen Personal (E-Mail, Telefon etc.)

Zur eindeutigen Identifikation einer zu einem bestimmten Zeitpunkt gültig gewesenen Konfiguration wird das *Id*-Attribut des *Location*-Elements und das bereits erwähnte *CreationTime*-Attribut des *FileInfo*-Elements verwendet.

Das *Device*-Element
Die Struktur dieses Elements ist durch die DTD der *Device Markup Language* (*DeviceML*) definiert. Sie dient zur typunabhängigen Repräsentation von Geräten und Geräteschnittstellen. Insbesondere werden durch dieses Element die unterstützten Geräteparameter und die Zugangswege zu diesen Parametern spezifiziert. Die vollständige DTD ist in Anhang A.2 angegeben und wird in Abschnitt 4.3 ausführlich vorgestellt.

4.2.2 Repräsentation von Prozessdaten

```
<!-- StateInformation -->
<!ELEMENT Snapshot (Description?, (ScannedDevice)*)>
<!ATTLIST Snapshot StartOfQuery CDATA #REQUIRED
    EndOfQuery CDATA #REQUIRED
    LocationId CDATA #REQUIRED
    SystemConfigurationCreationTime CDATA #REQUIRED>
<!ELEMENT ScannedDevice (LogEntry)*>
<!ATTLIST ScannedDevice DeviceId CDATA #REQUIRED>
<!ELEMENT LogEntry EMPTY>
<!ATTLIST LogEntry ParameterId CDATA #REQUIRED
    Value CDATA #IMPLIED
    Valid (0|1) "1">
```

Listing 4.5: Das *Snapshot*-Element

Die durch *Snapshot*-Elemente definierten *Snapshot*-Dokumente enthalten folgende Informationen (vgl. Listing 4.2):

1. Parameterwerte
2. Referenzen auf Konfigurationsdokumente

3. Zeitstempel
4. Zustandsbeschreibung

Die einzelnen Parameterwerte können über die Attribute *LocationId, SystemConfigurationCreationTime, DeviceId* und *ParameterId* den entsprechenden Informationen eines Konfigurationsdokumentes zugeordnet werden (vgl. Listings 4.1 und 4.2). Die Attribute *LocationId* und *SystemConfigurationCreationTime* identifizieren eine zu einem bestimmten Zeitpunkt gültige Konfiguration für eine bestimmte Anlage (vgl. Abschnitt 4.2.1). *DeviceId* referenziert eine *DeviceML*-Gerätebeschreibung für ein bestimmtes Gerät und *ParameterId* eine entsprechende Beschreibung für einen bestimmten Parameter dieses Gerätes.

Die Zeitstempel des Beginns und des Abschlusses der Datenerhebung werden in den Attributen *StartOfQuery* und *EndOfQuery* abgelegt. Das *StartOfQuery*-Attribut wird auch zur eindeutigen Identifikation von einzelnen Systemzuständen einer bestimmten Anlage verwendet. Ein als *DeMML*-Dokument kodierter Systemzustand ist durch Angabe der Attribute *LocationId* und *StartOfQuery* eindeutig identifiziert.

In jedes *Snapshot*-Dokument kann ein *Description*-Element eingehängt werden, das zur Beschreibung des Zustands in Klartext verwendet werden kann. Falls es sich bei einem *Snapshot*-Dokument um die Zustandsdaten eines Fehlerzustandes handelt, kann an dieser Stelle beispielsweise auf weiterführende Information bzgl. dieses Fehlerzustandes verwiesen werden.

4.3 Die *DeviceML* Document Type Definition

Wie bereits im vorigen Abschnitt erwähnt, wird die Struktur eines *DeMML-Device*-Elements durch die *Device Markup Language* (*DeviceML*) definiert. Die *DeviceML*-DTD wird als externe Entity in die *DeMML*-DTD eingebunden (vgl. Zeile 8-10 in Listing A.2 auf Seite 230).

4.3.1 Repräsentation von Geräteprofilen

```
<!ELEMENT Device (Description?, FileInfo?, DeviceInfo?,
    Parameters, DataAccess?, DeviceExt?)>
<!ATTLIST Device DeviceId CDATA #REQUIRED>
```

Listing 4.6: Das *Device*-Element

Das *Device*-Element enthält sämtliche Informationen, die zur Einbindung eines Gerätes in ein INSIGHT-System notwendig sind, wie z.B. Geräteidentifikation, Hard- und Software-Versionsnummern, Beschreibungen der Geräteparameter

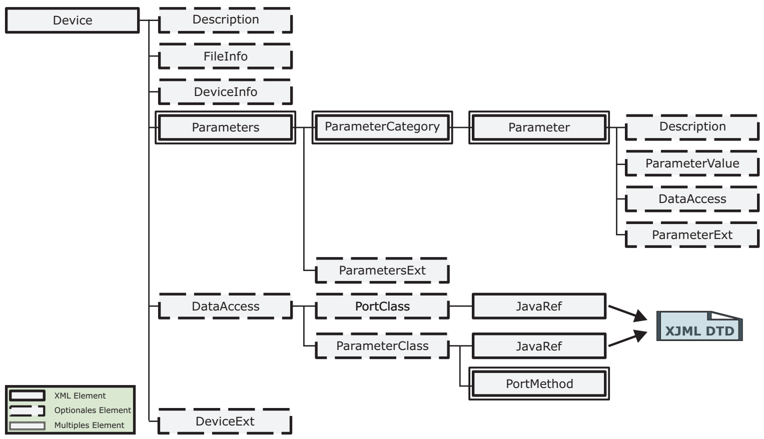

Abbildung 4.2: Ausschnitt aus der logischen Struktur eines *DeviceML*-Dokuments

und Referenzen auf entsprechende externe Hardwarezugriffsklassen. Diese Information ist zunächst gerätetypunabhängig und wird erst durch die Implementierung der referenzierten Hardwarezugriffsklassen typspezifisch spezialisiert. Gerätetypspezifische Erweiterungen der *XML*-Beschreibung können durch entsprechende DTDs um das *Device*-Element herumgebaut und in die *DeMML*-DTD eingebunden oder über sog. Extensionselemente (z.B. *DeviceExt*) ergänzt werden. Abschnitt 4.4 wird näher auf diese Thematik der *DeMML*-Erweiterbarkeit eingehen.

Abbildung 4.2 zeigt einen Ausschnitt aus der logischen Struktur von *DeviceML* bzw. des *Device*-Elements. Die vollständige *DeviceML*-DTD ist in Anhang A.2 angegeben. Die einzelnen Subelemente des *Device*-Elements werden in den folgenden Abschnitten vorgestellt.

Das *Description*-Element

```
<!ELEMENT Description ANY>
```

Listing 4.7: Das *Description*-Element

Dieses optionale Element kann dazu verwendet werden, eine Beschreibung des Gerätes an sich bzw. seiner Funktion innerhalb der Anlage zu geben (vgl. Listing 4.1, Zeile 13). Die interne Strukturierung dieses Elements unterliegt keinen weiteren Restriktionen. Es können auch Verweise auf zusätzliche externe Informationen (z.B. Online-Handbücher) abgelegt werden. Eine eventuelle interne Struktur des *Description*-Elements wird jedoch momentan vom INSIGHT-System ignoriert.

Das *FileInfo*-Element

```
<!ELEMENT FileInfo (FileInfoExt?)>
<!ATTLIST FileInfo FileName CDATA #IMPLIED
   FileVersion CDATA #IMPLIED
   FileRevision CDATA #IMPLIED
   CreationTime CDATA #IMPLIED
   CreationDate CDATA #IMPLIED
   CreatedBy CDATA #IMPLIED
   Description CDATA #IMPLIED>
```

Listing 4.8: Das *FileInfo*-Element

Dieses optionale Element enthält Informationen, die ein *DeviceML*-Dokument als solches betreffen, wie z.B. Autor oder Datum der letzten Änderung.

Das *DeviceInfo*-Element

```
<!ELEMENT DeviceInfo (Vendor?, Product?, DeviceInfoExt?)>
```

Listing 4.9: Das *DeviceInfo*-Element

Dieses optionale Element enthält Informationen über das Gerät an sich. Wichtige Informationen sind hier die Bezugsquelle und die exakte Typbezeichnung inklusive Versionsnummer (vgl. Listing 4.1, Zeilen 14-17).

Das *Parameters*-Element

```
<!ELEMENT Parameters (SupportedDataTypes?, ParameterCategory+,
   ParametersExt?)>
```

Listing 4.10: Das *Parameters*-Element

```
<!ELEMENT ParameterCategory (Description?, (Parameter | ParameterGroup)*)>
<!ATTLIST ParameterCategory Name CDATA #IMPLIED>
```

Listing 4.11: Das *ParameterCategory*-Element

Dieses Element enthält sämtliche Parameter-Deklarationen für ein konkretes Gerät. Die einzelnen Parameter werden hierbei zu Parameterkategorien zusammengefasst, die jeweils durch ein *ParameterCategory*-Element repräsentiert werden. Diese Elemente enthalten dann die einzelnen *Parameter*-Elemente, die jeweils einen Geräteparameter spezifizieren.

Das *Parameter*-Element

```
<!ENTITY % parameterAtts "Name␣CDATA␣#IMPLIED
   DataType CDATA #IMPLIED
   LowLimit CDATA #IMPLIED
   HighLimit CDATA #IMPLIED
   Unit CDATA #IMPLIED
   AccessType (ro|wo|rw|rwr|rww|const) 'rw'
   Mute (0|1) '0'">
```

Listing 4.12: Die *ParameterAtts*-Entity

```
<!ELEMENT Parameter (Description?, DefaultValue?, ParameterValue?,
   DataAccess?, ParameterExt?)>
<!ATTLIST Parameter %parameterAtts;
   ParameterId CDATA #REQUIRED
   Monitoring CDATA #IMPLIED>
```

Listing 4.13: Das *Parameter*-Element

Ein *Parameter*-Element spezifiziert einen Geräteparameter. Jeder Parameter eines Gerätes wird durch das Attribut *ParameterId* innerhalb einer Gerätebeschreibung eindeutig identifiziert. Parameter unterschiedlicher Geräte können dieselbe *ParameterId* besitzen. Wie bereits erwähnt, wird ein Parameter global über die Attribute *LocationId, SystemConfigurationCreationTime, DeviceId* und *ParameterId* identifiziert.

Die Parameter an sich werden mit Eigenschaften wie *Name*, *DataType*, *AccessType*, *DefaultValue* usw. beschrieben (vgl. Listing 4.1, Zeilen 20-23). Das Subelement *ParameterValue* kann dazu verwendet werden, aktuelle Parameterwerte direkt innerhalb einer Parameterbeschreibung abzulegen. So können beispielsweise die Parameterwerte eines *Snapshot*-Dokuments in ein Konfigurationsdokument gemischt werden, um eine reiche Darstellung eines Prozesszustandes zu ermöglichen.

Das *DataAccess*-Element

```
<!ELEMENT DataAccess (PortClass?, ParameterClass?)>
<!ELEMENT PortClass (JavaRef, (PortMethod)*)>
<!ATTLIST PortClass InitArgs CDATA #IMPLIED>
<!ELEMENT PortMethod (#PCDATA)>
<!ELEMENT ParameterClass (JavaRef)>
<!ATTLIST ParameterClass InitArgs CDATA #IMPLIED>
```

Listing 4.14: Das *DataAccess*-Element

Das *DataAccess*-Element, das bereits in der Deklaration des *Device*-Elements aufgetaucht ist (vgl. Listing 4.6), spezifiziert den Zugangsweg zu einem bestimmten Geräteparameter durch Referenzierung entsprechender gerätetypspezifischer

Hardwarezugriffsklassen. Das *DataAccess*-Element eines *Device*-Elements legt hierbei den Default-Zugangsweg zu den Parametern eines Gerätes fest, der dann durch optionale *DeviceAccess*-Elemente einzelner *Parameter*-Elemente ganz oder teilweise überschrieben werden kann.

Ein *DataAccess*-Element enthält mit den Subelementen *PortClass* und *ParameterClass* bis zu zwei Referenzen auf Java-Klassen. Die Referenzierung an sich ist jeweils durch ein *JavaRef*-Element definiert. Das *JavaRef*-Element ist Teil der *XML to Java Mapping Language* (*XJML*), die in Kapitel 7 ausführlich behandelt wird. Ein *JavaRef*-Element enthält i.d.R. eine Basis-URL und einen vollständig qualifizierten Java-Klassennamen. Mit diesen Informationen kann ein entsprechender Java-Klassenlader den Byte-Code via Internet laden und die entsprechende Klasse instanziieren.

Die durch das *PortClass*-Element referenzierte Hardwarezugriffsklasse ist hierbei zuständig für den protokollspezifischen Zugriff auf den gewünschten Hardwaretreiber (z.B. auf den CAN-Feldbus für einen CANopen-Parameter). Die *ParameterClass*-Klasse übernimmt die Repräsentation des Parameterwertes als Java-Objekt innerhalb der *DeviceInvestigator*-Komponente, die protokollspezifische Identifizierung eines bestimmten Parameters auf dem Gerät und die Serialisierung des Wertes in eine textbasierte Darstellung. Parametriert werden die einzelnen Java-Objekte durch die jeweils zugehörigen *DeMML*-Knoten.

Beide referenzierten Java-Objekte können bei Bedarf zusätzlich über das *InitArgs*-Attribut des entsprechenden *ParameterClass*- oder *PortClass*-Elements parametriert werden. Weiterhin können dem System über entsprechende *PortMethod*-Elemente protokollspezifische Managementfunktionen der Port-Objekte bekannt gemacht werden. Diese Methoden können dann über *Java Reflection* durch die *DeviceInvestigator*-Komponente aufgerufen werden. Das Zusammenspiel von *DeviceML*-Parameterbeschreibungen und Hardwarezugriffsklassen wird in Abschnitt 5.5 ausführlich beschrieben.

4.4 Erweiterbarkeit

Die Erweiterbarkeit von *DeMML* ist im Wesentlichen eine Erweiterbarkeit der Darstellung des *Device*-Elements, welches ein generisches Gerät repräsentiert. Ein *Device* enthält sämtliche Informationen, die erforderlich sind, um ein entsprechendes Gerät in ein INSIGHT-System zu integrieren. Bei der Festlegung dieser Elementdeklaration wurde darauf geachtet, möglichst viele der generischen Aspekte eines Geräts mit parameterbasierter Schnittstelle zu erfassen. Hierbei flossen insbesondere auch Erfahrungen mit ein, die im Rahmen der Spezifikation der *CANopen Markup Language* gesammelt wurden.

```
<Parameter Name="Device␣Type" DataType="7" AccessType="ro"
    ParameterId="1000,0" Monitoring="1" Mute="0">
  <DefaultValue>0x30191</DefaultValue>
  <ParameterExt Index="1000" Subindex="0" PDOMapping="0" ObjectType="7"/>
</Parameter>
```

Listing 4.15: Ein erweiterter Geräteparameter

Im Kontext der durchgeführten Fallstudien und implementierten Prototypen konnte das generische *Device*-Element zur Repräsentation von CAN/CANopen-Feldbusgeräten, einer TCP/IP-basierten Web-Kamera und weiteren Geräten, die über eine RS232-Schnittstelle ansprechbar sind, ohne Erweiterungen eingesetzt werden.

Es kann dennoch wünschenswert sein, gerätespezifische Aspekte, die innerhalb von INSIGHT u.U. keine Rolle spielen, in die *XML*-Repräsentation eines Geräts mit aufzunehmen. Ein Beispiel hierfür ist das *PDOMapping*-Attribut aus Listing 4.15, das den Zugriff auf einen CANopen-Parameter mittels des CANopen *Process Data Object* Protokolls zulässt oder verbietet (vgl. Abschnitt 10.1.2).

Diese zusätzlichen Informationen können zwar von den gerätetypspezifischen *PortClass*- und *ParameterClass*-Java-Objekten verwendet werden, sie können aber auch nur in einem völlig unabhängigen Anwendungskontext, außerhalb der INSIGHT-Infrastruktur (z.B. zur Produktdokumentation), relevant sein. So ist beispielsweise aus der Parameterbeschreibung aus Listing 4.15 mittels des *EDS2XML*-Übersetzers (siehe Abschnitt 4.7) eine Parameterbeschreibung im CANopen-spezifischen *Electronic Data Sheet* (EDS) Format [Can99] komplett rekonstruierbar (vgl. Abschnitt 4.4.3).

4.4.1 Extensionselemente

Die bereits erwähnten Elemente *DeviceExt, FileInfoExt, DeviceInfoExt* usw. (siehe Anhang A.2) wurden definiert, um Erweiterungen der entsprechenden Elemente *Device, FileInfo, DeviceInfo* usw. ohne Modifikation der *DeviceML*-DTD zuzulassen. Diese Extensionselemente haben alle das Inhaltsmodell (*Content Type*) *EMPTY* und keine deklarierten Attribute. Diese Attribute können später in einer weiteren typspezifischen DTD oder in einem lokalen Deklarationsteil im Prolog eines *XML*-Dokumentes hinzugefügt werden. Dieses Verfahren ist aufgrund der Spezifikation des *XML*-Standards nur auf Attributebene anwendbar, da Attributdeklarationen im Gegensatz zu Elementdeklarationen mehrmals überschrieben bzw. erweitert werden dürfen.

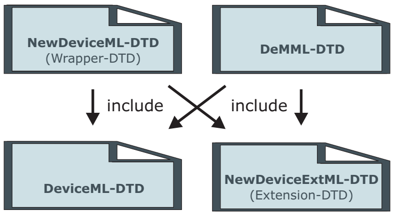

Abbildung 4.3: Erweiterung der *DeMML*-DTD für ein *NewDevice*

4.4.2 DTD-Einbindungen

Der *XML*-Standard verbietet es in der aktuellen Version 1.0, das Inhaltsmodell von bereits deklarierten Elementen später zu erweitern bzw. zu überschreiben. Daher können keine hierarchischen, polymorphen Erweiterungen der logischen Struktur einer DTD durch direktes Einbinden von weiteren DTDs erreicht werden. Ist jedoch eine strukturelle Erweiterung der *DeMML*-DTD erwünscht, weil die Erweiterbarkeit der DTD über Extensionselemente nicht ausreicht, so muss dies über die Erstellung entsprechender *DeMML*-Extensions- bzw. Wrapper-DTDs erfolgen. Eine Wrapper-DTD bindet das generische *Device*-Element durch entsprechende Referenzen ein und legt die zusätzlichen Informationen parallel zu diesem Element ab. Die Erweiterung der *DeMML*-DTD für eine neue Geräteart erfolgt in drei Schritten:

1. Erstellung der Extensions-DTD
2. Einbindung der Extensions-DTD in die *DeMML*-DTD
3. Erstellung der Wrapper-DTD (optional)

Abbildung 4.3 gibt einen Überblick über die beteiligten DTDs für ein exemplarisches *NewDevice*. Die Notwendigkeit der Unterscheidung der Extensions-DTD von der Wrapper-DTD liegt in dem bereits erwähnten Verbot der mehrfachen Deklaration von *XML*-Elementen begründet.

Zunächst werden die zusätzlichen typspezifischen Element- und Attributdeklarationen in einer neuen Extensions-DTD festgehalten. Diese wird anschließend über eine entsprechende Entity-Deklaration in die *DeMML*-DTD eingebunden. Zusätzlich kann eine entsprechende typspezifische Wrapper-DTD angelegt werden, die lediglich die *DeviceML*-DTD sowie die Extensions-DTD einbindet und keine weiteren DTD-Deklarationen enthält. Diese DTD kann zur *DeMML*-unabhängigen

```
<!ENTITY % DeviceML.dtd SYSTEM
  "http://www-sr.informatik.uni-tuebingen.de/Insight/DeviceML.dtd">
%DeviceML.dtd;

<!ENTITY % CANopenDeviceExtML.dtd SYSTEM
 "http://www-sr.informatik.uni-tuebingen.de/Insight/CANopenDeviceExtML.dtd">
%CANopenDeviceExtML.dtd;
```

Listing 4.16: Die *CANopenDeviceML*-Wrapper-DTD

Beschreibung von Geräten und deren Schnittstellen, z.B. innerhalb von *XML*-basierten Geräte-Datenbanken und Produktkatalogen, verwendet werden.

4.4.3 Fallstudie: Die *CANopen Device Markup Language*

Als Beispiel für eine derartige DTD-Erweiterung wird im Folgenden die *CANopen Device Markup Language* (*CANopenDeviceML*) kurz vorgestellt. CANopen [Can96b] ist ein offenes Schicht-7-Protokoll für den Feldbus *Controller Area Network* (CAN) (vgl. Abschnitt 10.1.2). Die Wrapper-DTD ist in Listing 4.16 abgebildet. Sie wird zur *DeMML*-unabhängigen, *XML*-basierten Repräsentation von CANopen-Geräten und -Geräteprofilen verwendet. *CANopenDeviceML*-Dokumente können CANopen-Geräteprofile vollständig beschreiben, da sie sämtliche Informationen aufnehmen können, die normalerweise im CANopen-spezifischen *Electronic Data Sheet* (EDS) Format [Can99] abgelegt werden. Die Übersetzung von EDS nach *CANopenDeviceML* kann automatisch erfolgen, z.B. mit Hilfe des *EDS2XML*-Übersetzers (siehe Abschnitt 4.7).

Die entsprechende Extensions-DTD ist im Anhang A.4 vollständig angegeben. Der in Listing 4.17 abgebildete Ausschnitt aus dieser DTD zeigt das Einschließen des generischen *Device*-Elements der *DeviceML*-DTD in das neue *CANopenDevice*-Element (Zeile 1) sowie exemplarische Attributdeklarationen eines Extensionselements der *DeviceML*-DTD (Zeile 6-11). Das neue Wurzelelement enthält nun parallel zu einem *Device*-Element das typspezifische *CANopenDeviceExt*-Element, dessen Struktur frei definiert werden kann (Zeile 3-4).

Wenn die neu geschaffene typspezifische DTD innerhalb von INSIGHT verwendet werden soll, muss das Einschließen des generischen *Device*-Elements in der dargestellten Art und Weise erfolgen, d.h. das neue Wurzelelement muss das *Device*-Element als direktes Subelement deklarieren. Die Einbindung der Extensions-DTD erfolgt über die übliche Entity-Deklaration. Zusätzlich muss das *DeMML-SystemConfiguration*-Element angepasst werden, da es nun auch Repräsentationen des spezialisierten Gerätetyps aufnehmen können soll (vgl. Listing 4.18). Dies ist

```
<!ELEMENT CANopenDevice (Device, CANopenDeviceExt)>

<!ELEMENT CANopenDeviceExt (LMT?, BaudRate?, BootMode?,
   DummyUsage? <!-- ... -->)>

<!ATTLIST DeviceInfoExt Granularity CDATA #IMPLIED
  DynamicChannelsSupported CDATA #IMPLIED
  GroupMessaging (0|1) "0"
  NrOfRXPDO CDATA #IMPLIED
  NrOfTXPDO CDATA #IMPLIED
  LSS_Supported (0|1) "0">
```

Listing 4.17: Ein Ausschnitt aus der *CANopenDeviceExtML*-Extensions-DTD

```
<!ENTITY % CANopenDeviceExtML.dtd SYSTEM
 "http://www-sr.informatik.uni-tuebingen.de/Insight/CANopenDeviceExtML.dtd">
%CANopenDeviceExtML.dtd;

<!ELEMENT SystemConfiguration (FileInfo, StateInfo?, SystemInfo,
 (Device | CANopenDevice)*)>
```

Listing 4.18: Anpassung des *SystemConfiguration*-Elements der *DeMML*-DTD

die einzige Veränderung, die in der *DeMML*-DTD vorgenommen werden muss.

4.4.4 Zusammenfassung

Die *DeviceML* als Teilbestandteil von *DeMML* wurde für die Repräsentation von Geräten beliebiger Art ausgelegt. Es können unabhängig vom Gerätetyp sämtliche Informationen abgelegt werden, die für die Einbindung des Gerätes in ein INSIGHT-System notwendig sind.

Falls zusätzliche Eigenschaften in dieses generische *XML*-Gerüst mit aufgenommen werden sollen, kann *DeMML* über Extensionselemente und/oder DTD-Einbindungen erweitert werden. Die Extensionselemente sind variabel gehaltene, optionale Elemente der *DeviceML*-DTD, deren Attributdeklarationen erweiterbar sind, ohne dass die *DeMML*-DTD selbst geändert werden muss. Falls eine Strukturerweiterung der *DeviceML* erfolgen soll, kann dies über die Erstellung entsprechender Extensions-DTDs bzw. Wrapper-DTDs, die das generische *Device*-Element enthalten, erreicht werden. Hierzu muss die Deklaration des *SystemConfiguration*-Elements der *DeMML*-DTD geändert werden. Wenn die DTD-Einbindung in der im vorigen Abschnitt beschriebenen Art und Weise erfolgt, kann das INSIGHT-System die erweiterten Formate ohne Anpassung verar-

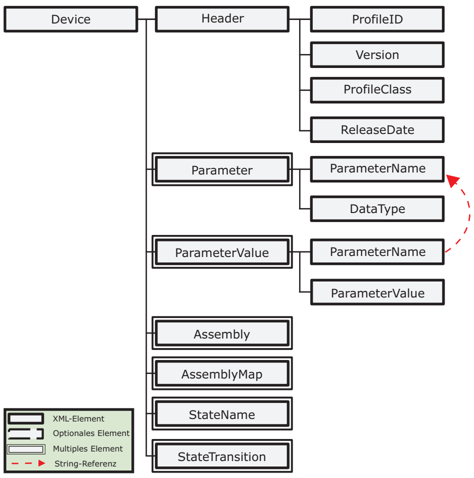

Abbildung 4.4: Ausschnitt aus der logischen Struktur eines PEL-Dokuments

beiten.

4.5 Alternative Ansätze

4.5.1 IEC61915: *Profile Exchange Language*

Die von der IEC vorgeschlagene Profile Exchange Language (PEL) [Int00] ist eine *XML*-Applikation zur Repräsentation von industriellen, vernetzen Geräten und deren Schnittstellen unabhängig vom jeweiligen Gerätetyp. Abbildung 4.4 zeigt einen Ausschnitt aus der logischen Struktur der PEL-DTD. Neben Informationen über das Gerät an sich und die verfügbaren Parameter der Geräteschnittstelle wird auch ein Zustandsdiagramm inklusive Zustandsübergangstabelle verwaltet.

Der Hauptvorteil von PEL ist, dass sich dieses Format evtl. als Industriestandard durchsetzen wird. Die Standardisierung ist noch nicht abgeschlossen und ist für Januar 2002 geplant.

PEL hat jedoch sowohl konzeptionelle wie auch strukturelle Eigenschaften, die dazu geführt haben, dass die *DeviceML* nicht auf PEL aufbaut.

Die Wertebereiche mancher Attribute, z.B. für die Geräteprofil-Klassifizierung und Datentypen, müssen einer statischen Tabelle entnommen werden, die auf Geräte aus dem industriellen Umfeld ausgelegt ist. Dies ist eine unnötige Einschränkung für eine offene Geräterepräsentation außerhalb industrieller Anwendungen.

Die Abbildung der für PEL definierten Eigenschaften auf eine entsprechende DTD nützt die hierarchischen Strukturierungsmöglichkeiten von *XML* nur oberflächlich. Zum Beispiel werden, wie in Abbildung 4.4 angedeutet, Parameterwerte im selben Dokument wie die Parameterbeschreibungen abgelegt, aber nicht als Subelemente der Beschreibung wie in *DeMML*, sondern auf derselben Hierarchieebene parallel zu der Parameterbeschreibung. Die Zuordnung von Parameterwerten zu Parameterbeschreibungen wird über String-Referenzen (hier: *ParameterName*) hergestellt[1]. Dieser Ansatz produziert ein erhebliches Maß an Redundanzen innerhalb eines *XML*-Dokuments mit allen damit verbundenen Nachteilen, wie z.B. der Gefahr von *Dangling References*. Diese Art der String-Referenzierung tritt innerhalb von PEL an mehreren Stellen auf, z.B. auch bei der Definition des Zustandsdiagramms.

Im Gegensatz zu *DeMML* bietet PEL kein Konzept zur Integration zusätzlicher typspezifischer Information, die in einem nicht-generischen Anwendungskontext von Bedeutung sein kann. Die parallele Verwaltung von mehreren *XML*-basierten Gerätebeschreibungen wäre aus Redundanzgründen wieder sehr unvorteilhaft. Die Offenheit von *DeMML* erlaubt zusätzlich prinzipiell die Übersetzung von *DeMML*-Dokumenten in PEL-Dokumente durch ein entsprechendes Tool oder ein entsprechendes *XSLT*-Dokument.

Das wesentliche Konzept der flexiblen Spezifikation der Zugangswege zu einzelnen Geräteparameter über *DeMML-DataAccess*-Elemente ist in dieser Form nicht in PEL integrierbar.

4.5.2 *XML*-basierte e-Business Gerätebeschreibungen

Neben den allgemeinen *XML*-Applikationen aus dem Umfeld der Business-to-Business-Kommunikation (z.B. *XML Common Business Library* (xCBL)[2] oder *Commerce XML* (cXML)[3]) werden zur Zeit auch spezielle Standards zur eindeutigen Identifizierung und Kategorisierung von Geräten entwickelt [Ome01] (z.B: XML-EDIFACT[4] oder RosettaNet[5]). Diese *XML*-Applikationen definieren ein einheitliches Vokabular, das einen automatischen Datenaustausch über Unternehmensgrenzen hinweg ermöglicht, z.B. zur dynamischen Kostenabschätzung unter Einbeziehung mehrerer Zulieferer.

[1] *DeMML* verwendet ebenfalls String-Referenzen zur Zuordnung von Parameterwerten, aber nur, wenn diese aus Effizienzgründen in einem separaten *Snapshot*-Dokument verwaltet werden.

[2] http://www.xcbl.org

[3] http://www.cxml.org

[4] http://www.xml-edifact.org

[5] http://www.rosettanet.org

Die Abstraktionsschicht dieser *XML*-Applikationen liegt höher als bei *DeMML*. Da die Geräte an sich zwar samt ihrer Features beschrieben werden, aber die Parameter-, bzw. Hardwarezugriffsebene nicht spezifiziert wird, haben diese Entwicklungen den Entwurf von *DeMML* kaum beeinflusst. Konzeptionell sind diese *XML*-Applikationen Sammlungen von Spezialfällen, die jeweils eine Zuordnung von realen Objekten zu den *XML*-Beschreibungen erlauben. *DeMML* verfolgt in dieser Hinsicht einen offeneren Ansatz, da hardwarespezifische Informationen und Funktionalitäten über Referenzen auf Java-Code realisiert werden und nicht an Spezifikationskataloge gebunden sind. Denkbar wäre allerdings eine Einbindung von *DeMML* in diese *XML*-Applikationen, um eine vollständige Beschreibung und Identifikation der Geräte zu erreichen.

4.6 Das *DeMML* Java-Paket

Das Java-Paket *de.wsi.demml* unterstützt den abstrakten Zugriff auf *DeMML*-Informationen und deren Visualisierung durch *Swing*-GUI-Komponenten [RV99]. Es baut hierbei auf der *Java API for XML Processing* (JAXP) [Suna] von Sun microsystems und der *Crimson* [Apac] Bibliothek von Apache auf.

Die *JAXP* Bibliothek enthält den *ProjectX*-*XML*-Parser und ist kostenfrei verfügbar. Über eine *ParserFactory*-Klasse können unterschiedliche *XML*-Parser (z.B. effizienz- oder speicherplatzoptimierte Parser) transparent in die API integriert werden. Die ebenfalls kostenfreie *Crimson* Bibliothek ist eine Implementierung und Erweiterung der SAX- und DOM-Schnittstellen und wird innerhalb des *Apache XML Project* [Apaa] entwickelt und gepflegt. Beide Bibliotheken zusammen unterstützen den bequemen Zugriff auf *XML*-kodierte Information sowohl über die SAX-Schnittstelle als auch über die DOM-Baumstruktur (vgl. Abschnitt 2.2.3.2).

Auf diesen Bibliotheken aufbauend stellt das *de.wsi.xml*-Paket zunächst grundlegende Funktionalitäten zur Erzeugung und Verwaltung von allgemeinen *XML*-Dokumenten zur Verfügung, die dann durch das *de.wsi.demml*-Paket auf *DeMML*-Dokumente spezialisiert werden.

4.6.1 Zugriff auf *DeMML*-Dokumente

Die Klasse *de.wsi.xml.Traversal* in Verbindung mit dem *de.wsi.xml.Traversal-Callback*-Interface erlaubt die Traversierung von DOM-Bäumen gemäß dem *Visitor*-Pattern [GHJV95] (vgl. Abbildung 4.5). Die Traversierung wird mit einer Klasse parametriert, die die abstrakte Methode *visit()* des *Traversal-Callback*-Interface implementiert. Diese Methode wird dann während einer

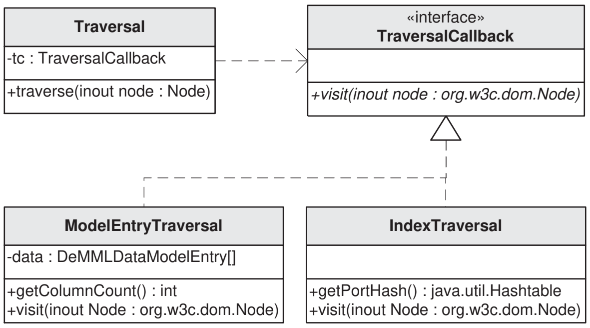

Abbildung 4.5: Die *Traversal*-Schnittstelle

```java
ModelEntryTraversal met = new ModelEntryTraversal(dataModelIndex.data);
Traversal trav = new Traversal(met);
trav.traverse(doc.getDocumentElement());
```

Listing 4.19: Benutzung der *Traversal*-Schnittstelle

Baumtraversierung rekursiv auf allen besuchten Knoten aufgerufen. Sie kann die besuchten Knoten selbst verändern oder Traversierungsinformationen aggregieren. Die Traversierungslogik ist fest in der *Traversal*-Klasse implementiert und unterstützt sowohl *Preorder-, Postorder-* als auch *Level-Order*-Traversierungen [Sed92].

In Abbildung 4.5 sind exemplarisch zwei Implementierungen der *TraversalCallback*-Schnittstelle angegeben. Die *de.wsi.demml.ModelEntryTraversal*-Implementierung erstellt ein Array von *DeMMLDataModelEntry*-Objekten für ein *XML*-Element zur Darstellung in *Swing*-Tabellen. Listing 4.19 zeigt den Ausschnitt aus der *DeMMLDataModel*-Implementierung (siehe Abschnitt 4.6.2), der das *Traversal*-Objekt erzeugt und die Traversierung durch Aufruf der *traverse()*-Methode startet.

Die *de.wsi.devin.IndexTraversal*-Klasse wird von *DeviceInvestigator* zum Erstellen der Indexstrukturen zum schnellen Zugriff auf alle spezifizierten *DeMML*-Parameter verwendet.

Die Klassen *XmlHelper* und *DeMMLHelper* bieten diverse statische Hilfsmethoden zum Zugriff auf allgemeine *XML* bzw. *DeMML*-Dokumente.

4.6.2 Visualisierung von *DeMML*-Dokumenten durch *Swing*-Komponenten

Die in diesem Abschnitt vorgestellten Visualisierungskonzepte und Klassen des *DeMML*-Pakets werden von sämtlichen GUI-basierten *DeMML*-Tools (z.B. *EDS2XML*, *ConfigurationWizard*, INSIGHT-*Applet-Client*) in der gleichen Art und Weise verwendet. Sie bilden ein Software-Framework zum schnellen Erstellen von spezialisierten Werkzeugen für *XML*- bzw. *DeMML*-basierte Applikationen.

Die zentrale Klasse der Visualisierung von *DeMML*-Dokumenten, die nach dem *Model View Controller* Konzept entwickelt wurde [KS88], ist die *DeMMLDataModel*-Klasse. Sie verwaltet die einzelnen Sichten (*Views*) und das gemeinsame Datenmodell (*Model*), das letztlich durch eine entsprechende *DeMML*-DOM-Struktur gegeben ist. Insbesondere werden alle Sichten von eventuellen Veränderungen innerhalb des Datenmodells gemäß dem *Observer*-Pattern [GHJV95] benachrichtigt und so konsistent gehalten. Im Einzelnen werden eine Baum-Sicht, diverse Tabellen-Sichten, eine Text-Sicht und eine generische Dialog-Sicht unterstützt.

Die Baum-Sicht Abbildung 4.6 zeigt eine *DeMML*-Baum-Sicht als Java-*JTree*-Komponente innerhalb von *ConfigurationWizard* (vgl. Abschnitt 4.8). Die Schnittstelle eines DOM-Knotens (*org.w3c.dom.Node*) unterscheidet sich von der Schnittstelle eines *JTree*-Knotens (*javax.swing.tree.TreeNode*) im Wesentlichen nur in der Bezeichnung der einzelnen Zugriffsmethoden. Damit sich *Node*-Knoten wie *TreeNode*-Knoten verhalten und in einer *JTree*-Komponente visualisiert werden können, wurde die Klasse *de.wsi.xml.TreeElement* implementiert. Diese Klasse ist von *ElementNode* abgeleitet und implementiert zusätzlich das *TreeNode*-Interface, indem die einzelnen *JTree*-Navigationsmethoden auf ihre DOM-Entsprechung abgebildet werden. Die Methode *getChildAt()* der *TreeNode*-Schnittstelle wird beispielsweise durch Zurückgreifen auf die *item()*-Methode der *Node*-Schnittstelle implementiert.

Da die Generierung von DOM-Knoten während einer *XML*-Dokumentanalyse gemäß dem *Factory*-Pattern [GHJV95] implementiert ist, kann die zu instanziierende Knoten-Klasse über eine externe Properties-Datei (s.u.) spezifiziert werden. In Listing 4.20 ist ein Code-Fragment abgebildet, das zunächst einen validierenden SAX-Parser (*parser*) mit entsprechendem Callback-Handler zur Erstellung von DOM-Knoten (*builder*) instanziiert (Zeile 1-3). In Zeile 4-6 wird eine neue Element-Factory (*factory*) erstellt und über ein *Properties*-Objekt (*properties*) konfiguriert. Eine entsprechende Property-Datei ist in Listing 4.21 abgebildet. Der Code in den Zeilen 9-11 startet dann den eigentlichen Parse-Vorgang und resultiert mit *doc* in einer DOM-Struktur, bestehend aus *TreeElement*-Knoten.

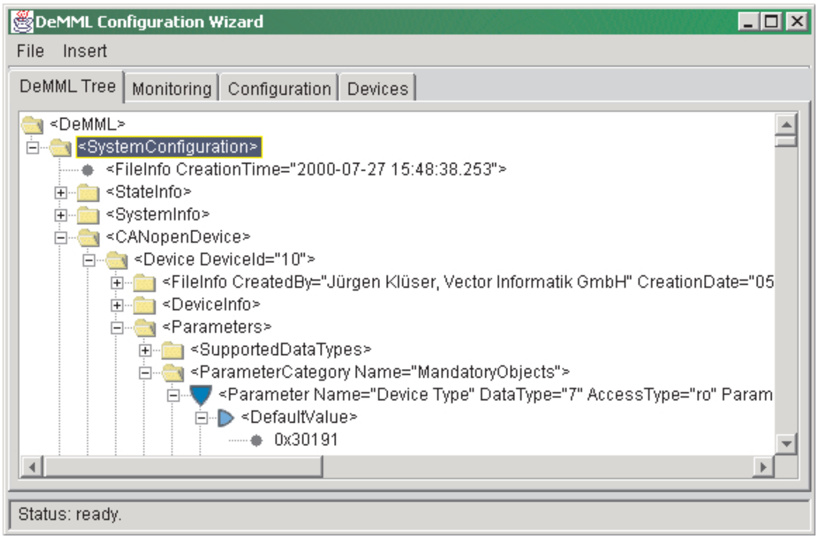

Abbildung 4.6: *DeMML*-Baum-Sicht im *ConfigurationWizard*

```
parser = XmlHelper.createValidatingParser();
builder = new DeMMLDocumentBuilder();
parser.setContentHandler(builder);

factory = new SimpleElementFactory();
factory.addMapping(properties, DeMMLDataModel.class.getClassLoader());
builder.setElementFactory(factory);

input = Resolver.createInputSource(new java.io.File(fileName));
parser.parse(input);
doc = builder.getDocument();
```

Listing 4.20: Verwendung einer Element-Factory zur konfigurierbaren DOM-Erstellung

Die in Listing 4.21 abgebildete Java-Properties-Datei enthält zwei sog. *Mappings*. Das Default-Mapping (Zeile 2) gibt an, welche Knoten-Klasse standardmäßig für alle *XML*-Elementtypen verwendet werden soll. In diesem Fall ist dies die bereits vorgestellte *TreeElement*-Klasse. Da sich *ParameterValue*-Elemente innerhalb des GUI-Frameworks teilweise speziell verhalten sollen, wird diesem Elementtyp eine spezialisiertere Klasse (*ParameterValueElement*) zugeordnet (Zeile 4), die von *TreeElement* abgeleitet ist.

```
# default mapping:
*Element = de.wsi.xml.TreeElement
# special mapping for ParameterValue elements:
ParameterValue = de.wsi.demml.ParameterValueElement
```

Listing 4.21: Eine Properties-Datei für eine Element-Factory

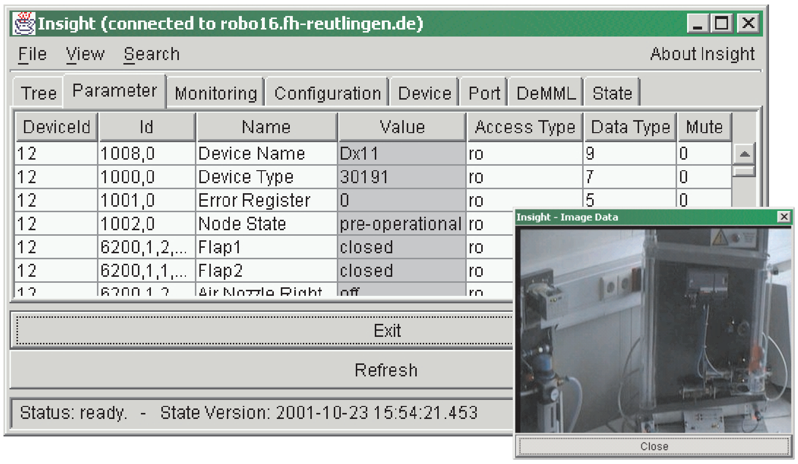

Abbildung 4.7: Eine *DeMML*-Tabellen-Sicht im INSIGHT-*Applet-Client* incl. Darstellung multimedialer Daten

Die Tabellen-Sichten Zur Erstellung unterschiedlicher tabellarischer Sichten auf *DeMML*-Dokumente wurden diverse Klassen implementiert, die alle von *javax.swing.table.AbstractTableModel* abgeleitet sind. Diese Klassen hüllen die hierarchische DOM-Datenstruktur in das logische Datenmodell der *Swing*-Tabellen. Hierzu werden je nach gewünschter Sicht bestimmte $1 : n$ Beziehungen der hierarchischen DOM-Struktur dynamisch in entsprechende zweidimensionale Relationen umgewandelt und visualisiert. Abbildung 4.7 zeigt eine tabellarische Sicht auf *DeMML*-Parameterbeschreibungen im INSIGHT-*Applet-Client*.

Die generische Dialog-Sicht Zur dynamischen Erzeugung von Dialogen, die eine Modifizierung beliebiger *XML*-Elemente erlauben, wurde die Klasse *de.wsi.xml.ElementNodePanel* implementiert. Dieses Panel traversiert rekursiv ein *XML*-Element und legt für alle besuchten Elemente neue *ElementNodePanel*-Objekte an, die entsprechend der Rekursionstiefe geschachtelt angeordnet wer-

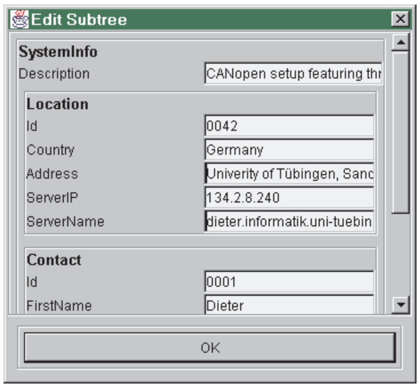

Abbildung 4.8: Ein dynamisch generierter Dialog für ein *SystemInfo*-Element

den. Ein *ElementNodePanel* für ein bestimmtes *XML*-Element E bietet GUI-Komponenten zum direkten Editieren aller vorhandener Attribute und angehängter terminaler Textknoten von E sowie weitere *ElementNodePanel*-Objekte für alle Subelemente von E. Abbildung 4.8 zeigt einen Dialog für ein *SystemInfo*-Element eines *DeMML*-Dokuments. Der Dialog wurde nur durch die Übergabe eines DOM-Knotens, in diesem Fall des *SystemInfo*-Elements, parametriert.

4.7 Der *EDS2XML*-Übersetzer

EDS2XML [BG00c] ist eine Beispiel-Applikation zur Konvertierung von gerätetypspezifischen Schnittstellenbeschreibungen in *DeMML*-Dokumente. *EDS2XML* ist speziell ausgelegt für CANopen-Feldbusgeräte, deren Schnittstellen (das sog. Geräteprofil) durch *Electronic Data Sheet* (EDS) [Can99] Dateien spezifiziert werden. Sie kann sowohl EDS-Dateien in *DeMML*-Dokumente konvertieren als auch *DeMML*-Dokumente in das EDS-Format übersetzen. Die linke Hälfte von Abbildung 4.9 zeigt einen Ausschnitt einer EDS-Datei für ein E/A-Modul, der weitgehend der *DeMML*-Information aus Listing 4.15 (S. 51) entspricht. Das EDS-Format basiert zum großen Teil auf keiner formalen Grammatik, sondern auf informellen Beschreibungen und Beispielen. EDS-Dateien werden i.d.R. zusammen mit den jeweiligen Geräten ausgeliefert, wobei sowohl standardisierte EDS-Dateien (z.B. [Can96a]) als auch vom Hersteller spezialisierte EDS-Dateien eingesetzt werden.

Konzeptionell beschreiben EDS-Dateien CANopen-Geräteprofile in textueller und maschinenlesbarer Form. Viele CANopen-Konfigurations- und Analysewerkzeuge verwenden direkt die EDS-kodierten Informationen, um z.B. ein neues Gerät in eine bestehende CANopen-Konfiguration einzubinden. Diese textuelle Beschreibung von Feldbusgeräteprofilen ist weit verbreitet, allerdings hat i.d.R. jeder Feldbus sein eigenes Beschreibungsformat. Es existieren stellenweise Tools, um Geräteprofilbeschreibungen von einem Format in ein anderes zu übersetzen.

Wie *XML*-Dokumente sind diese Geräteprofilbeschreibungen eine Form der textbasierten Repräsentation von strukturierter Information. Im Gegensatz zu *XML* ist aber die Strukturierung dieser Beschreibungen nicht an einem offenen, außerhalb der jeweiligen Feldbuswelt verbreiteten Standard orientiert. Es müssen daher jeweils spezialisierte Tools verwendet werden, die auf ein bestimmtes Format beschränkt sind.

DeMML in Verbindung mit *DeviceML* bietet hier die Möglichkeit, ein einheitliches Zwischenformat zu verwenden, das auf alle Gerätetypen anwendbar ist und mit einer Vielzahl von *XML*-Werkzeugen ohne weitere Medienbrüche in einer heterogenen Umgebung verwaltet werden kann. Wie bereits in Abschnitt 4.4 ausgeführt, sieht *DeMML* transparente, typspezifische Erweiterungsmöglichkeiten vor, die eine Anpassung an bestimmte Gerätetypen zulässt. Dieser Ansatz verhindert, dass Informationen über ein Gerät, die nicht in den generischen Kern von *DeviceML* passen, nicht in das *XML*-Format aufgenommen werden können.

Die ebenfalls bereits in Abschnitt 4.4 vorgestellte *CANopenDeviceML* ist eine solche Erweiterung speziell für CANopen-Geräte. Das *EDS2XML*-Werkzeug übersetzt EDS-Dateien in *CANopenDeviceML*-Dokumente und *CANopenDeviceML*-Dokumente in EDS-Dateien, ohne dass Informationen verloren gehen. Die durch *CANopenDeviceML* beschriebenen Geräte können dann direkt in die INSIGHT-Infrastruktur aufgenommen werden, indem das entsprechende *CANopenDevice*-Element in das *DeMML*-Konfigurationsdokument eingefügt wird.

EDS2XML ist als *Java Bean*-Komponente implementiert und kann entweder von der Kommandozeile oder über eine graphische Benutzerschnittstelle bedient werden. Das GUI des Tools ist aus Komponenten des im vorigen Abschnitt beschriebenen *DeMML*-GUI-Frameworks aufgebaut.

Im Einzelnen werden vier Sichten unterstützt:

1. EDS (textuelle Darstellung im EDS-Format, vgl. Abbildung 4.9)
2. *DeMML* Tree (*DeMML*-Baum-Ansicht, vgl. Abbildung 4.9 oder 4.6)
3. *DeMML* File (textuelle Darstellung als *DeMML*-Dokument)
4. Translation Log (SAX-Warnungen)

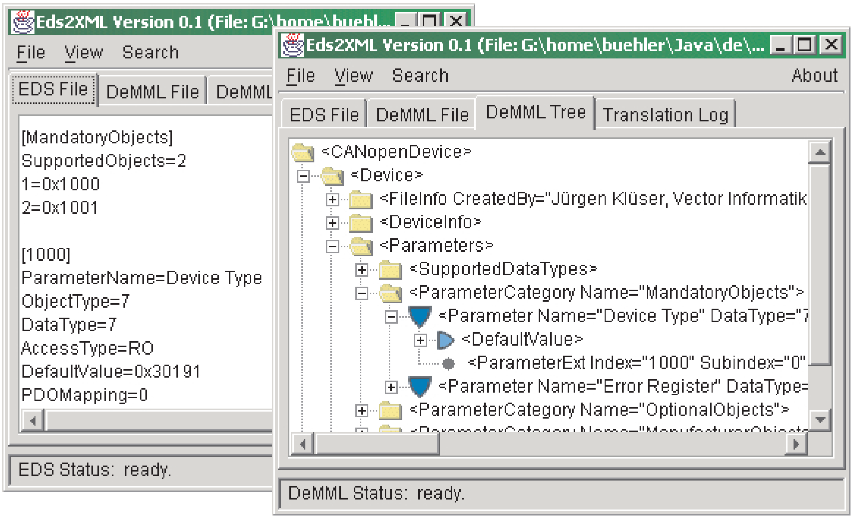

Abbildung 4.9: Eine *EDS-Datei* und die entsprechende *DeMML*-Baum-Sicht des *EDS2XML*-Übersetzers

Es können EDS-Dateien und *DeMML*-Dokumente geladen, gespeichert, visualisiert, editiert und konvertiert werden. Die *Translation Log*-Ansicht zeigt hierbei eventuelle Auffälligkeiten einer Übersetzung von EDS nach *DeMML* und weist dadurch i.d.R. auf Fehler in der EDS-Datei hin. Diese Auffälligkeiten werden durch Warnungen und Fehlermeldungen des verwendeten, validierenden *XML*-Parsers automatisch generiert, von einem entsprechenden *ErrorHandler*-Objekt abgefangen und in die Log-Datei eingetragen. In diesem Sinn kann *EDS2XML* als Vorstufe eines EDS-Conformance-Tests betrachtet werden.

Das *EDS2XML*-Werkzeug ist im Rahmen der Abstraktion von *CoML* zu *DeMML* aus dem älteren *Eds2CoML*-Übersetzer [BG00c] entstanden.

4.8 Der *ConfigurationWizard*

Der *ConfigurationWizard* ist ein weiteres Tool (vgl. Abbildung 4.6 auf Seite 60), das die GUI Komponenten des *DeMML*-Java-Pakets nutzt. Es unterstützt die komfortable Erstellung und Anpassung von *DeMML*-Konfigurationsdokumenten. Es können neue Konfigurationsdokumente erstellt und einzelne Gerätebeschreibungen per Menü in ein *DeMML*-Konfigurationsdokument eingefügt und anschließend editiert werden. Die einzelnen Gerätebeschreibungen können als Datei aus dem Dateisystem geladen oder über Angabe einer URL referenziert werden. Bei der Angabe einer URL wird das entsprechende *XML*-Dokument mit Hilfe des HTTP-Protokolls über das Internet geladen. Da, wie bereits im Abschnitt 3.3 erwähnt, auf eine *Tamino-XML*-Datenbank ebenfalls über das HTTP-Protokoll zugegriffen werden kann, kann die URL auch eine Query an eine Datenbank sein, um eine bestimmte Gerätebeschreibung aus einer entfernten Datenbank zu selektieren.

Der *ConfigurationWizard* wurde ursprünglich bereits für *CoML* entwickelt und später an *DeMML* angepasst. Das Tool integriert deshalb auch standardmäßig den *EDS2XML*-Übersetzer, der sich aus dem *Eds2CoML*-Übersetzer entwickelt hat. Somit können EDS-Dateien, die automatisch nach *DeMML* übersetzt werden, direkt in ein *DeMML*-Konfigurationsdokument als gültige Gerätebeschreibung eingefügt werden.

Kapitel 5

Generische Datenaggregation mit *DeviceInvestigator*

5.1 Überblick

DeviceInvestigator ist ein *XML*-basiertes Software-Framework zur flexiblen Erstellung von generischen, *XML*-basierten Datenaggregationskomponenten. Die jeweiligen Datenquellen werden hierbei durch *DeMML*-Dokumente beschrieben. In diesen Dokumenten werden sowohl die einzelnen Geräte und ihre Parameter als Datenbestandteile als auch die Zugangswege zu den Geräten bzw. Parametern beschrieben. Die Beschreibung der Zugangswege geschieht hierbei durch die Referenzierung von Java-Klassen, deren Instanzen den Datenzugriff und die Serialisierung der Parameterwerte übernehmen. Die Aufgabe von *DeviceInvestigator* ist es, die spezifizierten Hardwarezugriffsklassen dynamisch zu instanziieren, mit ihnen die entsprechenden Daten zu erheben und zu einem neuen *DeMML-Snapshot*-Dokument zu aggregieren. *Snapshot*-Dokumente können von *DeviceInvestigator* im Dateisystem abgelegt oder direkt in eine lokale oder entfernte Datenbank exportiert werden. Weiterhin kann *DeviceInvestigator* schreibend auf einzelne Parameter zugreifen und gerätetypspezifische Managementfunktionalität zugänglich machen.

DeMML ist zwar speziell angepasst auf die Verwendung von Geräten als Datenquellen, aber *DeviceInvestigator* integriert prinzipiell beliebige Arten von Datenquellen. Es wäre beispielsweise auch möglich, laufende Programme (z.B. Server-Software) mit *DeviceInvestigator* zu überwachen.

5.2 Stand der Forschung

5.2.1 Industrielle Datenerfassungssysteme

Im Umfeld der PC-basierten industriellen Datenerfassungssysteme (*Data Acquisition Systems*) hat sich insbesondere die *OLE for Process Control* (OPC) [OPC98b, OPC98a] Technologie durchgesetzt. OPC wurde ursprünglich von Microsoft entwickelt und wird seit 1996 von einer unabhängigen Organisation[1] gepflegt.

Der OPC-Ansatz beruht auf der Definition von Datentypen und OLE (*Object Linking and Embedding*) bzw. COM/DCOM-Schnittstellen [Box98, EE98], die einen einheitlichen und transparenten Zugriff auf einzelne und gruppierte Geräteparameter ermöglichen. Eine Implementierung dieser Schnittstellen wird OPC-Server genannt. OPC-Server sind i.d.R. mächtige und teure Softwaresysteme und werden von vielen Herstellern für die unterschiedlichsten Feldbusse angeboten. Konzeptionell kann ein beliebiger OPC-Client mit beliebigen OPC-Servern kommunizieren. Da die OPC-Schnittstellen auch DCOM-konform angelegt wurden, kann diese Kommunikation auch über Netzwerke erfolgen. OLE, COM und DCOM sind als Microsoft-Technologien auf das Windows-Betriebssystem beschränkt[2] und nicht ohne weiteres auf andere Plattformen portierbar.

Ein weiterer Microsoft-COM-basierter Ansatz ist der *Open Data Acquisition Standard* (ODAS), der durch die *Open Data Acquisition Association*[3] gepflegt wird. Dieser Ansatz definiert standardisierte Schnittstellen zu PC-Datenerfassungshardware, auf die über generische ODAS-Client-Software zugegriffen werden kann.

Die *Object Management Group* (OMG) hat 1999 einen *Request for Proposal*[4] mit dem Titel *Data Acquisition for Industrial Systems* [Obj99] veröffentlicht, um eine plattformunabhängige Standardisierung von CORBA bzw. IDL-Schnittstellen in diesem Bereich voranzutreiben. Es werden jeweils spezialisierte Vorschläge für bestimmte Anwendungsgebiete und Gerätetypen motiviert.

Ein detaillierter Vergleich dieser und weiterer Datenerfassungssysteme findet sich beispielsweise in [Nie01].

[1] OPC Foundation: http://www.opcfoundation.org

[2] Es gibt allerdings Bestrebungen, zumindest COM und DCOM als Bibliotheken auch auf Unix/Linux-Systemen zu etablieren.

[3] http://www.opendaq.org

[4] Eine Übersicht über die mittlerweile eingegangenen Vorschläge findet sich unter http://www.omg.-org/techprocess/meetings/schedule/Data_Acquisition_RFP.html

5.2.2 Industrielle Monitoring-Systeme

Das Internet-basierte Überwachen von Anlagen ist von industriellem Interesse, da in diesem Bereich ein großes Einsparungspotential bzgl. Fernwartung und Qualitätssicherung vorhanden ist. Die ABB-Gruppe (www.abb.com) hat beispielsweise mit ihrem GLASS-System (*Global Access for Service and Support*) [IPR98] ein System entwickelt, das die Einbindung industrieller ABB-Komponenten über eine Web-Server- und eine CGI-Schnittstelle in ein Internet-basiertes Monitoring-System erlaubt. Auf Server-Seite werden die einzelnen Geräte jeweils über gerätetypspezifische Proxy-Objekte gekapselt und auf Client-Seite durch spezielle Grafiken innerhalb von Java-Applets visualisiert. Der entfernte Zugriff auf die Gerätedaten läuft über eine Datenbankschicht, die den direkten Zugriff auf die Geräte von unvorhersehbaren Client-Anfragen entkoppelt.

Das auf GLASS basierende *RoMain*-System [Nie99, Nie01] realisiert beispielsweise so die Fernüberwachung von Zügen. In [Nie01] werden insbesondere auch unterschiedliche *Data Acquisition* und Monitoring-Konzepte auf Design-Ebene verglichen und eine generische Softwarearchitektur für derartige Systeme vorgeschlagen.

5.3 Das *DeviceInvestigator*-Konzept

DeviceInvestigator ist als *XML*-basiertes Software-Framework unter den folgenden Anforderungen realisiert worden:

- Offenheit
- Plattformunabhängigkeit
- Skalierbarkeit
- Flexibilität
- Datenbankanbindung

Offenheit Offenheit meint hier, dass das *DeviceInvestigator*-Framework die Integration beliebiger Datenquellen unterstützen muss. Hierfür ist die saubere Trennung zwischen Datenaggregationsinfrastruktur und Hardwarezugriffsklassen sowie ein einheitliches und dennoch flexibles Datenformat erforderlich. Die Trennung wird durch Reduzierung der Schnittstelle auf zwei generische Java-Interfaces in Verbindung mit dem Konzept des *XML*-basierten Software-Frameworks realisiert (vgl. Abschnitt 2.3.1). Als einheitliches Datenformat wird sowohl auf Datenerfassungsebene als auch auf Datenbankebene die *Device Management Markup*

Language (*DeMML*) verwendet. Dies impliziert, dass sämtliche erfassten Gerätedaten als serialisierte Zeichenketten in Form von *XML*-Fragmenten verwaltet werden. Die entsprechenden Datentypen werden jeweils über *DeMML*-Attribute zugeordnet, wobei die Interpretation der Datentypen den Hardwarezugriffsklassen überlassen wird.

Plattformunabhängigkeit *DeviceInvestigator* ist eine reine Java 1.1 Lösung und somit auf allen Plattformen lauffähig, auf denen eine entsprechende virtuelle Maschine verfügbar ist. Dies können insbesondere auch kleine, kostengünstige eingebettete Systeme sein, die als Datenerfassungseinheit an laufende Systeme angesetzt werden können. Eine Verwendung derartiger Hardware-Plattformen ist insbesondere auch deshalb möglich, weil *DeviceInvestigator* die erfassten Daten direkt in entfernte Datenbanken ablegen kann und diese daher nicht auf dem Erfassungssystem selbst, das in diesem Fall i.d.R. über sehr begrenzte Ressourcen verfügt, gehalten werden müssen.

Skalierbarkeit *DeviceInvestigator* sollte sich an die Komplexität der zu überwachenden/integrierenden Anlage anpassen. Bei kleinen, einfachen Anlagen sollten nur wenige Ressourcen (Hauptspeicher, CPU, Betriebssystemdienste) beansprucht werden. Insbesondere sollte in diesen Fällen kein CORBA ORB, OPC-Server, Windows Registry o.ä. vorausgesetzt werden. Auf der anderen Seite sollten diese mächtigen Dienste auf einfache Weise in das System integriert werden können, falls dies erforderlich ist. Die Skalierbarkeit resultiert hier aus der Offenheit (s.o.).

Flexibilität Neben der bereits erwähnten Offenheit und Skalierbarkeit sollte das System flexibel und aus der Ferne rekonfiguriert werden können, um beispielsweise unterschiedliche Gerätedaten ein- bzw. auszublenden. *DeviceInvestigator* erreicht diese Flexibilität durch seine Realisierung als *XML*-basiertes Software-Framework.

Datenbankanbindung Das *DeviceInvestigator*-Framework soll über eine effiziente, transparente Anbindung an ein entferntes Datenbanksystem basierend auf einem einheitlichen Datenformat verfügen. Wie bereits erwähnt verwendet *DeviceInvestigator* *DeMML* als universelles Datenformat. Die Datenbankanbindung erfolgt über HTTP-Aufrufe, wie sie direkt von den Java-Core-Klassen unterstützt werden. Als Datenbank kommt das native *XML*-Datenbanksystem *Tamino* zum Einsatz, für das ein entsprechendes universelles *DeMML*-Datenbankschema zum effizienten Zugriff auf *DeMML*-Daten erstellt wurde.

5.4 Arbeitsweise

Wie in Use-Case 2 bereits angedeutet, zerfällt die Ausführung von *Device-Investigator* in unterschiedliche Phasen:

1. Initialisierung
   (a) Konfigurationsanalyse
   (b) Erstellung der Zugriffsstrukturen inkl. dynamischer Instanziierung der Hardwarezugriffsklassen
2. Betrieb (zyklisch oder auf Anfrage)
   (a) Datenaggregation (Scan)
   (b) Speicherung

5.4.1 Initialisierung

Die Initialisierung von *DeviceInvestigator* wird durch ein *DeMML*-Konfigurationsdokument spezifiziert. Dieses *XML*-Dokument wird in der Analysephase zunächst in eine entsprechende DOM-Struktur übersetzt. Der *SystemConfiguration*-Elementknoten (vgl. Abschnitt 4.2) dieser Struktur enthält danach sämtliche Informationen, die für die Initialisierung und Ausführung von *DeviceInvestigator* nötig sind:

1. Den Zeitstempel des Konfigurationsdokumentes
2. Die Identifikation der Anlage
3. Die Default-Zustandsbeschreibung des *StateInfo*-Elements
4. Die einzelnen Geräte- und Parameterbeschreibungen, insbesondere mit
   (a) *DeviceId*
   (b) *ParameterId*
   (c) *DataAccess* Konfiguration

Wie in Abschnitt 4.2.1 bereits ausgeführt, wird eine bestimmte Konfiguration für eine bestimmte Anlage durch den Zeitstempel des *DeMML*-Konfigurationsdokumentes und der Anlagen ID eindeutig identifiziert. Damit von *DeviceInvestigator* erhobene *DeMML*-Zustandsdokumente für diese Anlage später wieder dem entsprechenden Konfigurationsdokument zugeordnet werden können, werden diese Daten in jede neue Datenaggregation (*Snapshot*-Dokument) übernommen.

Die Zustandsbeschreibung des *StateInfo*-Elements wird als Default-Zustandsbeschreibung verwendet, die über entsprechende Java-Methoden angepasst werden kann. Sie wird als *Description*-Element in jedes neu erstellte *Snapshot*-Dokument eingefügt.

Als letztes benötigt *DeviceInvestigator* noch die Informationen über die einzelnen Parameter. Für einen bestimmten Geräteparameter auf einem bestimmten Gerät sind diese Informationen zum größten Teil in dem entsprechenden *Parameter*-Element enthalten (vgl. Abschnitt 4.3). Zusätzlich werden noch die Geräte-ID (*DeviceId*) und die *DataAccess*-Konfiguration für diesen Parameter benötigt. Die *DataAccess*-Konfiguration beschreibt die Zugangswege zu diesem Parameter, indem sie Java-Klassen referenziert, die den Hardware-Zugriff durchführen. Sie wird für ein ganzes Gerät, wie in Abschnitt 4.3.1 bereits erläutert, durch ein entsprechendes Subelement des *Device*-Elements definiert. Diese Konfiguration kann für jeden Parameter individuell durch Einhängen eines entsprechenden *DataAccess*-Elements in das *Parameter*-Element ganz oder teilweise überschrieben werden.

Mit diesen Informationen erstellt *DeviceInvestigator* zunächst ein *Snapshot*-DOM, in dem das *LocationId*- und *SystemConfigurationCreationTime*-Attribut eingetragen, die *Value*-Attribute der *LogEntry*-Elemente aber noch nicht gesetzt sind (vgl. z.B. Listing 4.2 auf Seite 43). Diese Datenstruktur bleibt für die komplette Laufzeit von *DeviceInvestigator* im Hauptspeicher vorhanden und wird bei jeder Datenaggregation mit den erhoben Parameterwerten aktualisiert. Als letzter Schritt der Initialisierungsphase werden, basierend auf den jeweiligen *DataAccess*-Konfigurationen, Indexstrukturen für die Hardwarezugriffsklassen angelegt. Diese Indexstrukturen enthalten dynamisch instanziierte Java-Objekte, die für jeden Parameter den entsprechende Hardwarezugriff und die Verbindung zu den entsprechenden *LogEntry*-Elementen des *Snapshot*-DOM realisieren. Weitere Details zu diesen Hardwarezugriffsklassen finden sich in Abschnitt 5.5.1.

5.4.2 Betrieb

Nach erfolgter Initialisierung ist *DeviceInvestigator* bereit, die spezifizierten Parameterwerte mit Hilfe der Hardwarezugriffsklassen der Indexstrukturen zu lesen und in die *Value*-Attribute der jeweils assoziierten *LogEntry*-Elemente zu schreiben. Dieser Vorgang wird als *Scan* bezeichnet. Ferner werden durch einen Scan die Zeitstempel der Attribute *StartOfQuery* und *EndOfQuery* des *Snapshot*-Elements entsprechend dem Beginn und dem Ende der Datenaggregation aktualisiert. Nach erfolgtem Scan wird das *Snapshot*-DOM in ein entsprechendes *DeMML*-Dokument serialisiert, das entweder im Dateisystem oder in einer evtl.

entfernten Datenbank abgelegt wird (siehe Abschnitt 6.2.3.2).

DeviceInvestigator kann entweder als eigenständiges Werkzeug zum zyklischen Erfassen des Zustandes einer Anlage (*Monitoring*) verwendet werden (vgl. Use-Case 2 in Abschnitt 3.4.2) oder als Softwarekomponente zur Integration in weitere Applikationen, wie z.B. in den INSIGHT-Server (vgl. Use-Case 1 in Abschnitt 3.4.1). Im ersteren Fall werden die einzelnen Scans von *DeviceInvestigator* selbst durch einen entsprechenden Java-Thread gestartet.

Im zweiten Fall werden die einzelnen Scans extern initiiert. Zusätzlich erlaubt *DeviceInvestigator* über eine Einbettung in den INSIGHT-Server den schreibenden Zugriff auf einzelne Geräteparameter, die über *DeviceId* und *ParameterId* identifiziert werden sowie den Zugriff auf gerätespezifische Managementfunktionen, die über *PortMethod*-Elemente des *DataAccess*-Elements angemeldet werden können (vgl. Listing 5.1, Zeile 7).

5.5 Implementierung

In diesem Abschnitt wird die Implementierung der im vorigen Abschnitt beschriebenen Arbeitsweise von *DeviceInvestigator* vorgestellt. *DeviceInvestigator* ist ein *XML*-basiertes Software-Framework zur generischen Gerätedatenaggregation, das durch Referenzierung gerätetypspezifischer Hardwarezugriffsklassen auf die jeweiligen Geräte einer Anlage maßgeschneidert wird. In der Initialisierungsphase werden die referenzierten Hardwarezugriffsklassen dynamisch geladen, instanziiert und in einer Index-Datenstruktur zum effizienten Zugriff auf die spezifizierten Parameter verwaltet. Die eigentliche Datenaggregation (Scan) besteht dann aus definierten Methodenaufrufen auf diesen zur Compile-Zeit unbekannten Hardwarezugriffsklassen.

5.5.1 Allokation der Hardwarezugriffsklassen

Ein zentrales Konzept innerhalb von *DeviceInvestigator* ist die Abbildung von *XML*- Fragmenten des *DeMML*-Konfigurationsdokumentes auf Java-Klassen, die diese Informationen analysieren und verarbeiten[5]. Die jeweiligen Java-Klassen werden durch entsprechende *JavaRef*-Elemente der *XML to Java Mapping Language* (*XJML*) (vgl. Abschnitt 7.8.2) referenziert. Im Rahmen von *DeviceInvestigator* wird jedem definierten *DeMML*-Parameter durch die zwei *JavaRef*-Elemente eines *DataAccess*-Elements eine *Port*- und eine *Parameter*-Java-Klasse zugeordnet. Listing 5.1 zeigt die exemplarische *DataAccess*-Konfiguration für

[5]Eine Erweiterung dieses Konzepts wird ebenfalls von *XJM_Eval* zur flexiblen Ähnlichkeitsbewertung von *XML*-Dokumenten verwendet (vgl. Abschnitt 7.8.2).

```
   <Parameter ParameterId="Image1" DataType="image/jpeg"
2   Name="Overview" AccessType="ro" Mute="0">
    <DataAccess>
4     <PortClass InitArgs="http://134.2.8.240:8888/video/push">
       <JavaRef Codebase="http://suvretta.informatik.uni-tuebingen.de/classes"
6        Class="de.wsi.devin.HttpImagePort"/>
       <PortMethod>getStatistics</PortMethod>
8     </PortClass>
     <ParameterClass>
10     <JavaRef Codebase="http://suvretta.informatik.uni-tuebingen.de/classes"
        Class="de.wsi.devin.HttpImageParameter"/>
12   </ParameterClass>
    </DataAccess>
14 </Parameter>
```

Listing 5.1: Ein *DataAccess*-Element für einen Bild-Parameter

einen Parameter, der das aktuelle Bild einer Web-Kamera über das HTTP-Protokoll anfordert.

Diesem speziellen Parameter werden damit die Klassen *de.wsi.devin.HttpImagePort* und *de.wsi.devin.HttpImageParameter* zugeordnet. Weiterhin wird angegeben, dass der entsprechende Java-Byte-Code jeweils unter der URL *http://suvretta.informatik.uni-tuebingen.de/classes/* zu finden ist. Zusätzlich werden Informationen über Initialisierungsargumente (*InitArgs*) und Port-spezifische Managementfunktionalität (*PortMethod*) gegeben.

Für jeden in einem *DeMML*-Dokument spezifizierten Parameter existiert eine entsprechende *DataAccess* Konfiguration entweder durch die Default-Konfiguration des jeweiligen *Device*-Elements oder durch zusätzliche *DataAccess*-Elemente für einzelne Parameter. In der Initialisierungsphase werden zwei entsprechende Indexstrukturen angelegt, eine für die *Port*- und eine für die *Parameter*-Objekte. Diese Indexstrukturen bilden die Grundlage für spätere Datenaggregationen.

Der Port-Index und der Parameter-Index werden durch die bereits in Abschnitt 4.6.1 kurz vorgestellte Klasse *IndexTraversal* erstellt. Die *visit()*-Methode dieser *TraversalCallback* Implementierung, die auf allen Elementknoten des Konfigurations-DOMs aufgerufen wird, hat für einen Knoten $node$ die folgende Aufgaben:

1. Falls $node$ kein *DeMML-Parameter*-Element ist, wird $node$ ignoriert

2. Erzeugung eines *Parameter*-Java-Objekts, entsprechend der *ParameterClass* Spezifikation (vgl. Listing 5.1, Zeile 9-12)

3. Falls die durch das *PortClass*-Element referenzierte Java-Klasse noch nicht instanziiert wurde, wird ein Objekt dieser *Port*-Klasse erzeugt

4. Assoziation des *Parameter*-Objektes mit dem entsprechenden *Port*-Objekt

5. Assoziation des *Parameter*-Objektes mit dem entsprechenden *LogEntry*-Knoten des *Snapshot*-DOM

Schritt 1 *DeMML*-Knoten, die keine Parameterbeschreibung enthalten, werden ignoriert.

Schritt 2 Im zweiten Schritt werden für alle *Parameter-DeMML*-Elemente die entsprechenden *Parameter*-Java-Objekte erzeugt und in den Parameter-Index eingetragen. Die referenzierten Java-Klassen müssen hierbei das generische Java-Interface *de.wsi.devin.Parameter*, das in Abschnitt 5.5.2.1 ausführlich beschrieben wird, implementieren. Dies ist die einzige Bedingung an die Klassen, die mittels des *PortClass*-Elements referenziert werden.

Die Parameter Java-Objekte sind zum Einen die Bindeglieder zwischen Konfigurations-DOM und *Snapshot*-DOM und zeichnen sich zum Anderen dafür verantwortlich, einen bestimmten Geräteparameter protokollspezifisch zu identifizieren. Ein CANopen-Parameter wird beispielsweise auf dem Gerät über Index- und Subindex-Werte sowie eine CANopen-Modul-ID identifiziert.

Schritt 3 Im Gegensatz zu den *Parameter*-Objekten wird nicht für jede *Port*-Spezifikation ein entsprechendes Port Java-Objekt erzeugt, sondern jede referenzierte *Port*-Klasse wird genau einmal instanziiert[6]. Dieses Objekt wird in den Port-Index eingetragen und von allen *Parameter*-Objekten, die diese *Port*-Klasse referenzieren, genutzt. Die *Port*-Klassen müssen jeweils das generische Interface *de.wsi.devin.Port* (vgl. Abschnitt 5.5.2.2) implementieren, um von *Device-Investigator* verwaltet werden zu können.

Die *Port*-Objekte bieten Methoden zum protokollspezifischen Zugriff auf die entsprechende Hardware.

Schritt 4 In Schritt 4 wird innerhalb des erzeugten *Parameter*-Objekts der Verweis auf das zugehörige *Port*-Objekt hergestellt. Das *Port*-Objekt wird hierbei als generisches Objekt vom Typ *Port* an die *Parameter*-Klasse übergeben und dort auf einen gerätetypspezifischen, spezialisierteren Typ gecastet. Es ergibt sich nach der Initialisierungsphase eine 1:n Beziehung zwischen *Port*- und *Parameter*-Objekten (vgl. Abbildung 5.1).

[6] Da die entsprechenden *Port*-Objekte nur von der *IndexTraversal*-Klasse erzeugt werden und die Implementierung spezialisierter *Port*-Klassen völlig offen gehalten werden sollte, wurde auf eine Implementierung gemäß dem *Singleton*-Pattern [GHJV95] verzichtet.

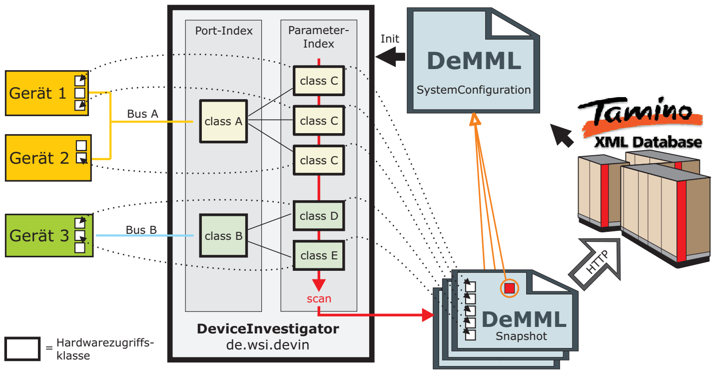

Abbildung 5.1: Generischer Gerätezugriff mit *DeviceInvestigator*

Schritt 5 Jedes *Parameter*-Objekt schreibt seine aktualisierten Werte während eines Scans in den entsprechenden Knoten des *Snapshot*-DOM, das permanent im Hauptspeicher gehalten wird. Die Referenz auf diesen DOM-Knoten wird in diesem Schritt gesetzt.

Die Instanziierung der einzelnen Java-Objekte wird jeweils durch die Methode *createNewInstance()* der *Projection*-Klasse des *XJML*-Java-Pakets zur Laufzeit durchgeführt (vgl. Abschnitt 7.8.2). Diese Methode nimmt als Argument einen *JavaRef*-Knoten und lädt den spezifizieren Java-Byte-Code von der angegeben *Codebase*-URL. Die Klasse wird dann durch die *newInstance()*-Methode der Klasse *java.lang.Class* mittels des Default-Konstruktors instanziiert. Die *Projection*-Klasse wird im Rahmen von *XJM_Eval* in Abschnitt 7.8.2 ausführlich erläutert.

5.5.2 Datenaggregation

Wie im vorigen Abschnitt ausgeführt, sind jedem *DeMML*-Parameter eine *Parameter*- und eine *Port*-Klasse als Hardwarezugriffsklassen zugeordnet. Diese Klassen müssen jeweils die generischen Schnittstellen *de.wsi.devin.Parameter* (vgl. Listing 5.2) bzw. *de.wsi.devin.Port* (vgl. Listing 5.3) implementieren, um von *DeviceInvestigator* gemäß dem für Software-Frameworks typischen *Inversion of Control* Paradigma [FS97] gemanagt werden zu können.

In diesem Zusammenhang kann *DeviceInvestigator* als *XML*-basiertes Software-Framework (vgl. Abschnitt 2.3.1) mit den Hot-Spots *Parameter* und *Port* zur dynamischen Integration gerätetypspezifischer Hardwarezugriffsklassen aufgefasst werden. Um eine zusätzliche Geräteart (z.B. Profibus-Feldbusgeräte) *Device-*

```
   public interface Parameter {
4    // Initialisierung :
     public void setParameterClassNode(Node node);
6    public void setParameterNode(Node node);
     public void setPort(Port port);
8    public void init();
     public void setSnapshotNode(Node node);
10   // Betrieb:
     public boolean update();
12   public String read();
     public boolean write(String value);
14   public String getParameterId();
     public String getDeviceId();
16   // Ende:
     public void cleanup();
18 }
```

Listing 5.2: Die *Parameter*-Schnittstelle

```
public interface Port {
  public void setPortClassNode(Node node);
  public void cleanup();
}
```

Listing 5.3: Die *Port*-Schnittstelle

Investigator und damit dem INSIGHT-System zugänglich zu machen, müssten entsprechende *Parameter*- und *Port*-Implementierungen erstellt und über ein entsprechendes *DeMML*-Konfigurationsdokument eingebunden werden. Entscheidend ist hierbei, dass durch die Einbindung neuer Hardwarezugriffsklassen keine Kompilierung des Gesamtsystems erforderlich wird.

Die Implementierung der Hardwarezugriffsklassen kann völlig unabhängig von *DeviceInvestigator* erfolgen. Es müssen lediglich die abstrakten Methoden der beiden Schnittstellen, die im Folgenden beschrieben werden, gerätetypspezifisch implementiert werden. Alternativ können diese Zugriffsklassen auch von den abstrakten Klassen *BasicParameter* und *BasicPort* abgeleitet werden, die bereits für einen Teil der Schnittstellen eine Default-Implementierung zur Verfügung stellen.

5.5.2.1 Die *Parameter*-Schnittstelle

Dieses abstrakte Java-Interface bildet die Schnittstelle für *DeviceInvestigator* zu einem abstrakten Geräteparameter, unabhängig von Gerätetyp oder Kommunikationsprotokoll. *DeviceInvestigator* ruft die einzelnen Methoden dieses Interfaces (siehe Listing 5.2) teilweise in der Initialisierungsphase (Zeile 5-9) und teilweise während des Betriebs (Zeile 11-15) auf, um einzelne Parameterwerte vom Gerät zu lesen oder auf das Gerät zu schreiben. Bei lesendem Zugriff während eines Scans sind die *Parameter*-Klassen ferner dafür verantwortlich, selbstständig das in der Initialisierungsphase erstellte *Snapshot*-DOM zu aktualisieren.

setParameterClassNode() Dies ist die erste Methode, die nach erfolgter Instanziierung des *Parameter*-Objekts von *DeviceInvestigator* aufgerufen wird. Der Übergabewert *node* ist der *ParameterClass*-DOM-Knoten des für diesen Parameter gültigen *DataAccess*-Elements (vgl. Listing 5.1, Zeile 9-12). Eine *Parameter*-Implementierung hat damit die in diesem Element enthaltenen Informationen, insbesondere das evtl. vorhandene *InitArgs*-Attribut, das zur erweiterten Parametrierung einer *Parameter*-Implementierung dienen kann, zur Verfügung.

setParameterNode() Diese Methode übergibt den *Parameter*-DOM-Knoten dieses Geräteparameters als Übergabewert *node* an die *Parameter*-Klasse. Wie in Abschnitt 4.3.1 ausgeführt, enthält dieses *XML*-Element die Beschreibung des Parameters inklusive Parameter-ID, Datentyp und Zugriffsrechten. Zur eindeutigen Identifikation des Parameters innerhalb der Anlage ist zusätzlich zu der Parameter-ID noch die Geräte-ID notwendig, die innerhalb des *Device*-Elements abgelegt ist. Dieses Element kann, falls nötig, innerhalb dieser Methodenimplementierung z.B. mit der Methode *getNamedParent()* der *DeMMLHelper*-Klasse (vgl. Abschnitt 4.6.1) ermittelt werden.

setPort() Diese Methode setzt eine Referenz innerhalb eines *Parameter*-Objekts auf die für diesen Parameter gültige *Port*-Klasse (vgl. Abbildung 5.1). Die Referenz wird als Übergabeparameter an die *Parameter*-Implementierung übergeben und dort auf einen spezialisierten Klassentyp gecastet (z.B. entsprechend Listing 5.1 auf *HttpImagePort*). Danach kann die *Parameter*-Implementierung hardwarespezifische Funktionen auf dem *Port*-Objekt aufrufen. *DeviceInvestigator* hat keine Informationen über diesen Type-Cast und sieht alle Hardwarezugriffsklassen nur als *Port*- bzw. *Parameter*-Schnittstellen.

init() Die Methode *init()* wird von *DeviceInvestigator* aufgerufen, nachdem die bereits vorgestellten Methoden abgearbeitet wurden. Eine Implementierung der *Parameter*-Schnittstelle kann in dieser Methode auf die oben angeführten Informationen zugreifen und sich selbst, z.B. durch Allokation entsprechender Kommunikationsobjekte, initialisieren.

Die bis jetzt vorgestellten Methoden werden alle innerhalb der *visit()* Implementierung der bereits vorgestellten *IndexTraversal*-Klasse aufgerufen.

setSnapshotNode() Diese Methode wird von *DeviceInvestigator* während der Erstellung des *Snapshot*-DOM auf den einzelnen *Parameter*-Objekten aufgerufen. Als Übergabeparameter wird die Referenz *node* auf den zu einem *Parameter*-Knoten gehörenden *LogEntry*-Knoten übergeben. Das *Parameter*-Objekt ist selbst dafür verantwortlich, den assoziierten *LogEntry*-Knoten bzw. dessen *Value*-Attribut auf Anfrage zu aktualisieren (s.u.).

Die folgenden Methoden werden während des Betriebs von *DeviceInvestigator* aufgerufen und stellen im Wesentlichen die Schnittstelle zum lesenden und schreibenden Zugriff auf Geräteparameter dar.

update() Dies ist die Methode, die von *DeviceInvestigator* während eines Scans auf jedem *Parameter*-Objekt aufgerufen wird. Die Implementierung aktualisiert ihren Status, indem sie auf dem assoziierten *Port*-Objekt protokollspezifische Methoden zum Lesen des jeweiligen Geräteparameters aufruft. Weiterhin aktualisiert sie das *Value*- und das *Valid*-Attribut des assoziierten *LogEntry*-Knotens des *Snapshot*-DOMs, indem sie den Parameterwert in eine textuelle Darstellung serialisiert.

read() Diese Methode sollte den aktuellen Wert des repräsentierten Parameters an *DeviceInvestigator* übergeben. Diese Methode ist für zukünftige Erweiterungen von *DeviceInvestigator* vorgesehen und wird im Moment nicht verwendet.

write() Diese Methode wird von *DeviceInvestigator* aufgerufen, wenn eine Anfrage zum Schreiben eines bestimmten, durch die *DeMML*-Attribute *DeviceId* und *ParameterId* identifizierten Parameters vorliegt. Bei einer solchen Anfrage ermittelt *DeviceInvestigator* zunächst die *Parameter*-Implementierung, die den zu schreibenden Geräteparameter kapselt, und übergibt den zu schreibenden Wert im Übergabeparameter *value*. Da innerhalb von *DeviceInvestigator* alle Daten als textuelle *DeMML*-Dokumente verwaltet werden, sind die zu schreibenden Werte zunächst immer vom Typ String. Die *Parameter*-Implementierung interpretiert diesen String entsprechend ihrer Konfiguration, z.B. falls nötig durch Deserialisierung oder Dekodierung, und ruft entsprechende protokollspezifische Hardwarezugriffsmethoden auf dem assoziierten *Port*-Objekt auf, um den Parameter auf das Gerät zu schreiben.

getParameterId() und getDeviceId() Durch diese Methoden kann ein bestimmtes *Parameter*-Objekt aus dem Parameter-Index eindeutig identifiziert werden. Diese Methoden benutzt *DeviceInvestigator* beispielsweise zum Auffinden einer bestimmten Parameterrepräsentation im Rahmen einer Anfrage zum Schreiben eines bestimmten Geräteparameters.

cleanup() Diese Methode wird von *DeviceInvestigator* aufgerufen, bevor ein *Parameter*-Objekt aus dem Parameter-Index gelöscht und dem Garbage-Collector überlassen werden soll. Das Objekt kann in einer Implementierung dieser Methode belegte (Hardware-) Ressourcen freigeben. Die Benutzung der Java *finalize()*-Methode, die automatisch von der Java-Laufzeitumgebung vor dem Freigeben eines Java-Objekts aufgerufen wird, ist hierfür nicht in allen Fällen ausreichend, da eventuell dynamisch gebundene native Laufzeitbibliotheken erst explizit freigegeben werden müssen, bevor die Laufzeitumgebung das Objekt überhaupt löschen kann.

5.5.2.2 Die *Port*-Schnittstelle

Dieses Interface bildet die abstrakte Schnittstelle zum Erstellen, Konfigurieren und Löschen eines gerätetypspezifischen Kommunikationskanals. Die Implementierungen dieser Schnittstelle stellen Methoden zum Zugriff auf spezifische Hardwaretreiber bereit, die von assoziierten *Parameter*-Implementierungen aufgerufen werden können.

Listing 5.3 auf Seite 77 zeigt die zwei Methoden dieser Java-Schnittstelle, die im Folgenden erläutert werden.

setPortClassNode() Ähnlich wie die Methode *setParameterClassNode()* der *Parameter*-Schnittstelle, wird diese Methode von *DeviceInvestigator* in der Initialisierungsphase aufgerufen, um eine *Port*-Implementierung gemäß eines *DataAccess*-Elements zu konfigurieren. Der Übergabewert *node* ist eine Referenz auf den entsprechenden *PortClass*-Knoten des Konfigurations-DOMs, der die Erzeugung dieses Knotens bewirkt hat. In dem *PortClass*-Knoten ist evtl. ein *InitArgs*-Attribut vorhanden, dessen Wert innerhalb einer Implementierung dieser Methode dazu benutzt werden kann, die *Port*-Implementierung zu konfigurieren. So kann hier z.B. für einen *HttpImagePort* die Basis-URL angegeben werden, von der graphische Bild-Daten einer Kamera via HTTP geladen werden können (vgl. Listing 5.1 auf Seite 74).

cleanup() Diese Methode ist analog zur Methode *cleanup* der *Parameter*-Schnittstelle dafür verantwortlich, belegte (Hardware)-Ressourcen freizugeben, wenn ein *Port*-Objekt nicht länger gebraucht wird.

Durch die vorgestellten Schnittstellen *Port* und *Parameter* ist die Trennungslinie

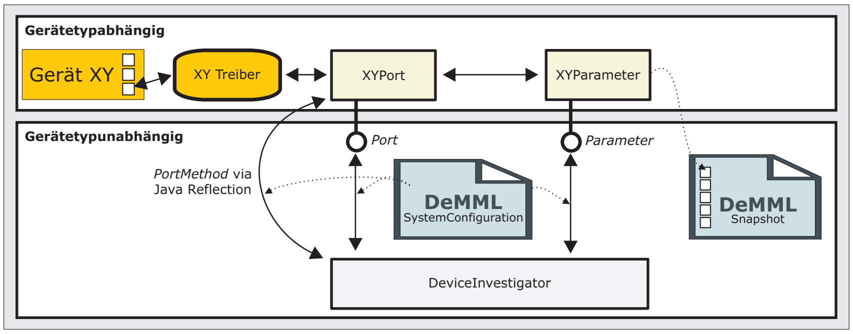

Abbildung 5.2: *DeMML*-definierte Trennung zwischen gerätetypabhängigem und gerätetypunabhängigem Programm-Code

zwischen gerätetypabhängigem und gerätetypunabhängigem Programmcode klar definiert (vgl. Abbildung 5.2). Es gibt keine weiteren Abhängigkeiten über diese Linie hinweg. Zusätzlich sind sämtliche Abbildungen der generischen Schnittstellen auf typspezifische Hardwarezugriffsklassen zur Laufzeit über entsprechende *DeMML*-Elemente konfigurierbar und müssen insbesondere nicht bereits zur Compile-Zeit bekannt sein. Aufgrund dieser losen Koppelung können zusätzliche gerätetypabhängige Hardwarezugriffsklassen völlig unabhängig von *DeviceInvestigator* entwickelt und ohne Rekompilierung des Gesamtsystems in eine bestehende Konfiguration aufgenommen werden.

Um diesen Ansatz halten zu können und dennoch den Zugriff auf gerätetypspezifische Funktionalität einzelner Ports außerhalb eines Scans zu ermöglichen, bietet die *DeviceInvestigator*-Implementierung zusätzlich die Möglichkeit, diese Funktionalitäten über entsprechende *DeMML-PortMethod*-Elemente (vgl. Abschnitt 4.3.1) zu publizieren und zugänglich zu machen. Diese werden dann auf Anfrage dynamisch über *Java Reflection* innerhalb der Methode *invokePortMethod()* (siehe Abschnitt 5.5.3) ausgeführt und verursachen dadurch keine zusätzlichen Anforderungen an die statische Organisation der Hardwarezugriffsklassen.

5.5.2.3 Implementierung der Hardwarezugriffsklassen

In diesem Abschnitt werden kurz einige der bereits implementierten Hardwarezugriffsklassen vorgestellt. Die Abbildungen 5.3 und 5.4 zeigen die Klassenhierarchie dieser *Port*- bzw. der zugehörigen *Parameter*-Implementierungen.

Die abstrakten Klassen *BasicPort* und *BasicParameter* können, müssen aber nicht,

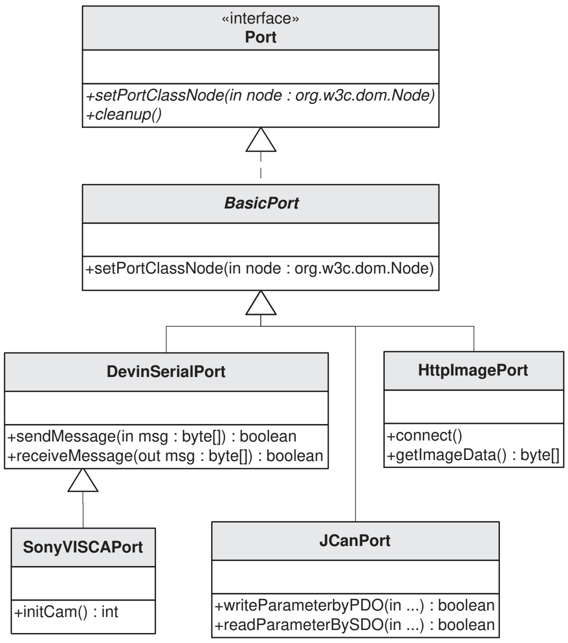

Abbildung 5.3: Ausschnitt aus der Klassenhierarchie einiger *Port*-Implementierungen

als Grundlage spezialisierter Klassen verwendet werden. Sie implementieren ein Default-Verhalten, das entweder direkt übernommen oder überschrieben werden kann. Wie in den entsprechenden Klassendiagrammen angedeutet, implementieren diese Basisklassen nur die Methoden, die während der Initialisierungsphase aufgerufen werden. Sie legen hierbei im Wesentlichen die *DeMML*-kodierten Informationen (z.B. den Wert des *ParameterId*-Attributs) in entsprechenden Java-Variablen ab, so dass spezialisierte Klassen direkt auf diese Variablen zugreifen können.

Die spezialisierten, an einen bestimmten Gerätetyp angepassten Klassen müssen dann insbesondere die Methoden, die während des Betriebs von *Device-Investigator* aufgerufen werden (*update(), write(), cleanup()*), protokollspezifisch implementieren.

Die Klasse *SonyVISCAPort* aus Abbildung 5.3 implementiert Teile eines speziell von der Firma Sony entwickelten Protokolls zur Steuerung von Sony Digital-

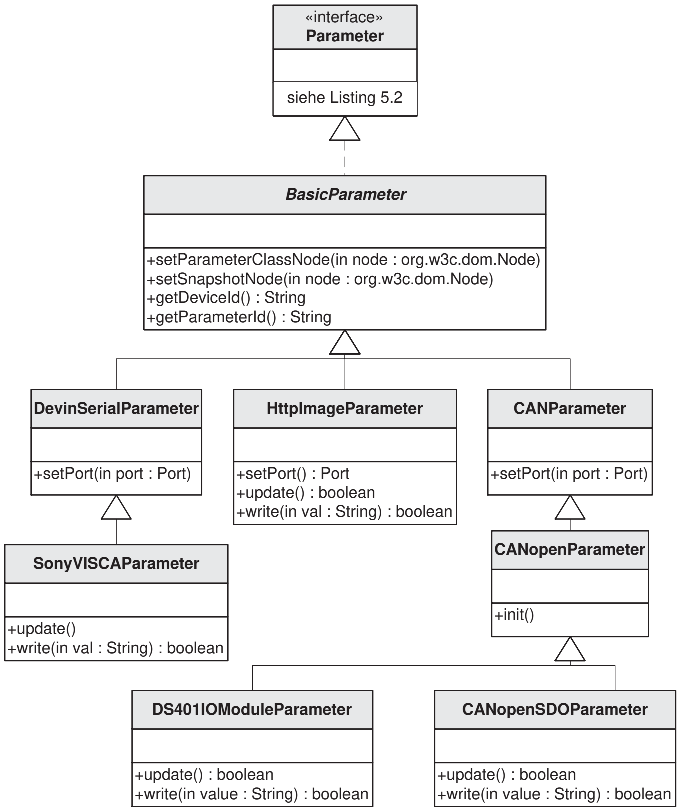

Abbildung 5.4: Ausschnitt aus der Klassenhierarchie einiger *Parameter*-Implementierungen

Kameras über die serielle Schnittstelle. Die Klasse erweitert hierzu die allgemeinere Klasse *DevinSerialPort*, die aufbauend auf der *JavaComm* API [Sunb] eine protokollunabhängige Kommunikation über die serielle Schnittstelle erlaubt.

Die bereits mehrfach erwähnte Klasse *HttpImagePort* erlaubt den Download von JPEG oder GIF kodierter Bilder über das HTTP-Protokoll. Die Klasse *JCanPort* kapselt im Wesentlichen die *CanPort*-Klasse der Java CAN API und erlaubt den Zugriff auf CAN-Geräte beliebiger Art. Die Java CAN API und die *CanPort*-Klasse wird in Kapitel 10 ausführlich vorgestellt.

Die *Parameter*-Implementierungen, die in Abbildung 5.4 exemplarisch aufgeführt

```
    public XmlDocument getDocument() {//...
    public XmlDocument getState() {//...
    public void monitor(long waitMillis,
      String workingDirectory, String configFileName) {//...
    public void monitor(long waitMillis, String taminoXmlUrl,
      String taminoNonXmlUrl, String configFileName) {//...
    public boolean writeParameter(String deviceId,
      String parameterId, String value) {//...
    public void invokePortMethod(String className,
      String codebase, String method, String args) {//...
    public void cleanup() {//...
```

Listing 5.4: Die *DeviceInvestigator*-Klasse

sind, bilden die Gegenstücke zu den entsprechenden *Port*-Klassen. Interessant ist hierbei die Unterscheidung von mehreren *Parameter*-Implementierungen für denselben Port, wie z.B. *CANopenSDOParameter* und *DS401IOModuleParameter* für den *JCanPort*. Ersterer greift auf die Parameter eines CANopen-Gerätes über sog. SDO-Kommunikationsobjekte zu und kann damit sämtliche für ein beliebiges CANopen-Gerät zugängliche Parameter lesen und schreiben. Die *DS401IOModuleParameter*-Klasse ist hingegen auf eine bestimmte CANopen-Geräteart, in diesem Fall auf ein digitales E/A-Moduls das der Spezifikation DS401 [Can96a] gehorcht, spezialisiert. Der Parameter liest ebenfalls bestimmte für diesen Gerätetyp zwingend spezifizierte Parameter und leistet zusätzlich eine Weiterverarbeitung der gelesenen Werte. Beispielsweise werden auf Geräteseite mehrere Kanäle eines DS401 E/A-Moduls zu einem binär kodierten Wert zusammengefasst und als aggregierter Parameter übertragen. Ein *DS401IOModuleParameter* kann zusätzlich für einen bestimmten Kanal des E/A-Moduls definiert werden, um duale Zustände aus der aggregierten Information zu filtern. Ein derartiger Parameter kann beispielsweise verwendet werden, um direkt den jeweiligen Zustand (*high*/*low*) eines an ein E/A-Modul angeschlossenen Lichtsensors zu repräsentieren (vgl. Fallstudie 3 auf Seite 107).

5.5.3 Implementierung von *DeviceInvestigator*

Entsprechend der unterschiedlichen Betriebsarten bietet die Klasse *DeviceInvestigator* u.a. die in Listing 5.4 aufgeführten Methoden an. Die Methoden *getDocument()* und *getState()* erlauben den Zugriff auf das aktuelle Konfigurations- und *Snapshot*-Dokument.

getState() Bei einer Anforderung des aktuellen *Snapshot*-Dokuments mittels der *getState()*-Methode wird zunächst das sich bereits im Hauptspeicher befindliche *Snapshot*-DOM durch einen Scan aktualisiert. Diese Aktualisierung wird durch die Methode *scanDevices()* der *Monitor*-Klasse durchgeführt. In dieser Methode wird innerhalb einer Schleife der Parameter-Index linear durchlaufen, auf allen *Parameter*-Implementierungen die jeweilige *update()*-Methode aufgerufen und damit die aktuellen Parameterwerte in das *Snapshot*-DOM geschrieben (vgl. Abbildung 5.1). Zusätzlich werden die Zeitstempel des Beginns und des Endes der Datenaggregation erfasst und ebenfalls im *Snapshot*-DOM vermerkt. Nach erfolgtem Scan wird das aktualisierte *Snapshot*-DOM in Form eines *org.apache.-crimson.XmlDocument*-Objektes zurückgegeben.

monitor() Die Methode *monitor()* erzeugt eine neue Instanz der Klasse *MonitorThread*, die mit parametrierbarer Frequenz (*waitMillis*) die *scanDevices()*-Methode aufruft. Das aktualisierte *Snapshot*-DOM wird nach jedem Scan entweder in eine neue Datei des Dateisystems serialisiert (Zeile 3-4) oder via HTTP in eine entfernte *Tamino*-Datenbank geschrieben (Zeile 5-6).

writeParameter() Die Methode *writeParameter()* ermöglicht den schreibenden Zugriff auf einen bestimmten Parameter. Der Parameter wird über *deviceId* und *parameterId* eindeutig referenziert. Nachdem die entsprechende *Parameter*-Implementierung im Parameter-Index gefunden wurde, wird die *write()*-Methode dieses Objekts mit dem übergebenen *Value*-Wert aufgerufen. Die *Parameter*-Implementierung ist dann dafür verantwortlich, dass der übergebene Wert korrekt interpretiert und protokollspezifisch auf das Gerät geschrieben wird.

invokePortMethod() Diese Methode erlaubt den generischen, dynamischen Zugriff auf gerätetypspezifische Funktionalität einzelner *Port*-Implementierungen. Da diese je nach Gerätetyp sehr unterschiedlich ausfallen können und die Schnittstelle zwischen gerätetypunabhängigem und gerätetypabhängigem Code so klein wie möglich gehalten werden sollte, wurde hierfür kein spezielles Interface definiert, sondern ein Weg über *Java Reflection* gewählt (vgl. Abbildung 5.2).

Java Reflection erlaubt den dynamischen Aufruf von zur Compile-Zeit unbekannten Methoden in zur Compile-Zeit unbekannten Klassen zur Laufzeit durch eine dynamische Untersuchung des unbekannten Byte-Codes. Listing 5.5 zeigt den entsprechenden Java-Quellcode der *DeviceInvestigator*-Klasse.

Als Erstes wird in Zeile 311 die über *className* und *codebase* identifizierte *Port*-Implementierung (*obj*) aus dem Port-Index ermittelt. Die weiteren Parameter *me-*

```
308 public void invokePortMethod(String className, String codebase,
        String method, String args) {
310   //...
      Object obj = monitor.getPortObject(className, codebase);
312   Class[] argClsArray = new Class[1];
      argClsArray[0] = args.getClass();
314   String[] argArray = new String[1];
      argArray[0] = args;
316   try {
        Method m = obj.getClass().getMethod(method, argClsArray);
318     m.invoke(obj, argArray);
      }
320   //...
```

Listing 5.5: Dynamische Ausführung von Port-Methoden via *Java Reflection*

thod und *args* enthalten den aufzurufenden Methodennamen und den Übergabeparameter (*args*) an diese Methode. Der Übergabeparameter ist immer vom Typ String und muss u.U. innerhalb der *Port*-Implementierung weiter analysiert bzw. deserialisiert werden. Als nächstes wird die aufzurufende Methode in *obj* ermittelt. Die einzelnen Methoden einer Java-Klasse werden über den Methodennamen und die Methodensignatur identifiziert. Die Methode *getMethod()* (Zeile 317) der Klasse *java.lang.Class* zum Suchen einer bestimmten Methode innerhalb einer Klassendefinition nimmt daher den Methodennamen und einen Array von Java-*Class*-Klassen der entsprechenden Methodensignatur (*argClsArray*) als Argumente. Die gefundene Methode wird durch ein Objekt der Klasse *java.lang.reflect.Method* gekapselt und durch die Methode *invoke()* zur Ausführung gebracht.

Welche Methoden ein bestimmter Port zur Verfügung stellt, wird durch entsprechende *DeMML-PortMethod* Subelemente des entsprechenden *PortClass*-Elements festgelegt. Die in Listing 5.1 wiedergegebene Spezifikation meldet z.B. das Vorhandensein einer Methode namens *getStatistics()* an. Ein *PortClass*-Element kann beliebig viele oder gar keine *PortMethod*-Elemente enthalten. Das Anbieten derartiger Funktionalität ist somit zum Einen optional, zum Anderen völlig unabhängig von der *DeviceInvestigator*-Implementierung.

5.5.4 Rekonfigurierung

Die Konfiguration von *DeviceInvestigator* findet während der Laufzeit statt und kann jederzeit während des Betriebs geändert werden. Es können beispielsweise einzelne *Parameter*-Elemente hinzugefügt oder entfernt bzw. stumm geschaltet werden. Zusätzlich ist es möglich, die Hardwarezugriffsklassen auszutauschen, z.B. um eine weitergehende Verarbeitung der erhobenen Geräteparameter zu erreichen. Weiterhin können komplette Gerätebeschreibungen hinzugefügt oder entfernt werden, z.B. wenn ein neues Gerät in die Anlage integriert oder ein altes erneuert wird. Dies alles kann zur Laufzeit des Systems geschehen, was besonders wichtig ist, wenn *DeviceInvestigator* im Rahmen eines Internet-basierten Informationssystems wie INSIGHT betrieben wird. Hier möchte der Benutzer interaktiv über ein Netzwerk die Konfiguration anpassen, um eine flexible Einsicht in die entfernte Anlage zu gewinnen und eine dedizierte Fehlerdiagnose stellen zu können.

Die Rekonfiguration an sich besteht aus der Anpassung des *DeMML*-Konfigurationsdokumentes, z.B. mittels des INSIGHT-*Applet-Client*s, und dem Aufruf der *createDokument()*-Methode, die einen erneuten Durchlauf der Initialisierungsphase bewirkt.

5.6 Fallstudie 1: Monitoring einer CAN-basierten Automatisierungsanlage

In diesem Abschnitt wird entsprechend Use-Case 2 eine Überwachung (*Monitoring*) einer Automatisierungsanlage durch *DeviceInvestigator* beschrieben. Die Daten, die auf diese Weise erhoben wurden, werden im Folgenden als Basis für eine exemplarische Fernwartung dieser Anlage dienen (siehe Abschnitt 7.11).

5.6.1 Geräteaufbau

Die überwachte Anlage (siehe auch [Lum99]) ist eine aus im industriellen Umfeld gängigen Komponenten aufgebaute Demonstrationszelle, die einzelne Werkstücke, in diesem Fall kleine Gummibälle, über mehrere Stationen in einem Kreislauf umherbewegt. Sie besteht aus den folgenden vier Gerätekomplexen:

1. Werkstückvereinzelung (Klappen, Foto-Sensoren, Druckluftdüsen, E/A-Module)
2. Transfersystem (Motor, Foto-Sensoren, Positionierungseinheiten, E/A-Module)
3. Bosch SCARA60 Roboter (4 Motoren/Achsen)
4. Lift (Klappen, Foto-Sensoren, Druckluftdüsen, E/A-Module)

Die einzelnen Geräte kommunizieren über zwei separate Feldbus-Leitungen miteinander, wobei das Feldbussystem *Controller Area Network* (CAN) (siehe Abschnitt 10.1.2) eingesetzt wird. Die vier Antriebsverstärker des SCARA (*Selective Compliance Assembly Robot Arm*) Roboters sind durch den ersten CAN miteinander verbunden und kommunizieren über ein proprietäres Protokoll der Firma

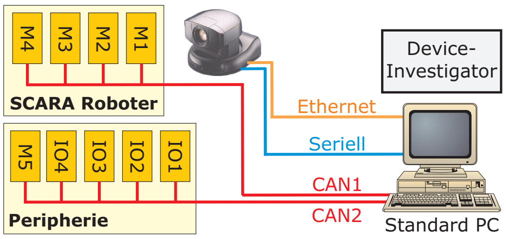

Abbildung 5.5: Hardware Setup der Automatisierungsanlage

Moog Ltd. Die Peripheriegeräte, die, abgesehen vom Motor des Transfersystems, über E/A-Module angesprochen werden, sind mit dem zweiten CAN verbunden und kommunizieren über das CANopen-Protokoll. Zusätzlich ist eine digitale Videokamera installiert, deren Blickwinkel sowie Zoom-Einstellung über die serielle Schnittstelle und deren Bilddaten über das HTTP-Protokoll zugänglich sind.

Die Steuerung dieser Anlage wurde als Java-Applikation mit Hilfe des *Java Fieldbus-based Control Framework* (*JFCF*) (siehe Kapitel 11) entwickelt und auf einem Standard PC unter Windows NT ausgeführt. *DeviceInvestigator* selbst wurde unabhängig davon ebenfalls auf diesem Rechner ausgeführt, was aber keine Voraussetzung für den Einsatz von *DeviceInvestigator* ist, da *DeviceInvestigator* völlig unabhängig von der eingesetzten Steuerungskomponente (z.B. SPS oder IPC) verwendet werden kann. Abbildung 5.5 zeigt ein schematisches Bild des Systemaufbaus.

5.6.2 Konfiguration von *DeviceInvestigator*

Das *DeMML*-Konfigurationsdokument für diese Anlage (siehe Anhang A.6) enthält für jedes der 10 Geräte (vgl. Abbildung 5.5) ein entsprechendes *Device*-Element mit den relevanten Parameterbeschreibungen. Es werden folgende Informationen erfasst:

1. Positionen der 4 Roboterachsen
2. Belegung der Ein- und Ausgänge der E/A-Module
3. Fehlerregister der E/A-Module (soweit unterstützt)
4. CANopen-Knotenzustand der E/A-Module (soweit unterstützt)
5. Zustand des Motors des Transfersystems

6. Blickwinkel und Zoomeinstellung der Kamera

7. Kamerabild

In Anhang A.7 ist ein vollständiges *Snapshot*-Dokument für diese Anlage angegeben. Es ergeben sich vier unterschiedliche Arten von Parametern innerhalb dieser Anlage:

1. CANopen-Parameter (Peripherie)

2. CAN-Parameter (Roboterachsen)

3. Serielle Parameter (Kamerasteuerung)

4. Bild-Parameter

Hinzu kommt eine unterschiedliche Verfügbarkeit der Parameterwerte. Manche Geräte lassen einen expliziten lesenden Zugriff auf einzelne Parameter zu, andere publizieren nur ihre Zustandsänderungen auf dem Bus und unterstützen keinen direkten Zugriff. In letzterem Fall muss eine *Parameter*-Implementierung gewählt werden, die die Auslösung eines Scans vom tatsächlichen Hardwarezugriff entkoppelt, indem sie selbstständig den Busverkehr überwacht und sich zu gegebenen Zeitpunkten selbst aktualisiert. Diese Parameter werden auch als *asynchron* bezeichnet. Die aktualisierten Werte werden dann bei Auslösung eines Scans nur noch in den entsprechenden Knoten des *Snapshot*-DOMs durchgeschrieben. Im Einzelnen werden die folgenden *Port/Parameter*-Paare eingesetzt:

1. *JCanPort/CANopenSDOParameter*, *JCanPort/CANopenUnsigned8PDOParameter*, *JCanPort/CANMsgParameter*

2. *MoogPort/MoogParameter*

3. *HttpImagePort/HttpImageParameter*

4. *SonyVISCAPort/SonyVISCAParameter*

Die *CANopenUnsigned8PDOParameter*- und *CANMsgParameter*-Implementierungen sind asynchrone Parameter, die selbstständig auf CANopen *Process Data Object* (PDO) Nachrichten des CANopen-Protokolls bzw. frei selektierbare CAN-Nachrichten hören.

5.6.3 Ergebnisse

Das in Anhang A.7 angegebene *Snapshot*-Dokument wurde durch *DeviceInvestigator* in eine entfernte *Tamino*-Datenbank abgelegt. Wie aus den Werten der *Snapshot*-Attribute *StartOfQuery* und *EndOfQuery* hervorgeht, dauerte die Aggregation der 27 spezifizierten Parameterwerte rund 90 Sekunden. Der zeitintensivste Teil war hierbei sicher die Erfassung und Übertragung der Bildinformation (ca. 9 KB) der Kamera.

Während weiterer Sitzungen wurden die Daten teilweise auch direkt in das lokale Dateisystem geschrieben. Die Information, wo *DeviceInvestigator* die generierten *Snapshot*-Dokumente ablegen soll, wird als Kommandozeilenparameter beim Start von *DeviceInvestigator* übergeben.

Zusätzlich wurden manuell diverse Fehlerzustände generiert (z.B. durch Abschalten bestimmter Druckluftdüsen), die ebenfalls in den Datenbestand mit aufgenommen wurden. Die *Description*-Elemente der entsprechenden *Snapshot*-Dokumente wurden hierbei jeweils mit Hilfe des INSIGHT-*Applet-Client* (siehe Abschnitt 6.2) mit entsprechenden Diagnoseinformationen für eine spätere automatisierte Fehlerdiagnose angereichert (siehe Kapitel 7).

5.7 Fallstudie 2: Integration von *DeviceInvestigator* in eine Service-Architektur für Magnetresonanztomographen

Im Rahmen einer Kooperation mit *Siemens Medical Solutions*[7] in Erlangen wurde die *DeviceInvestigator*-Komponente prototypisch in die bestehende Service-Softwareinfrastruktur für medizinische Magnetresonanztomographen (vgl. Abbildung 5.6) integriert.

Die Magnetresonanztomographie ist derzeit eines der modernsten diagnostischen bildgebenden Verfahren. Durch Einsatz von elektromagnetischen Wellen werden Wasserstoffatome in den zu untersuchenden Körperregionen angeregt, die daraufhin selbst elektromagnetische Wellen freisetzen. Diese werden gemessen und zur Berechnung entsprechender Bilder verwendet. Diese Diagnosemethode ist besonders geeignet für die Darstellung des Nervensystems (Gehirn und Rückenmark), von Gelenken und Weichteilen sowie von Herz, Kreislaufsystem und den Bauchorganen.

Magnetresonanztomographen sind i.d.R. sehr komplexe und teure (bis zu 2 Mio.

[7] http://www.siemensmedical.com/

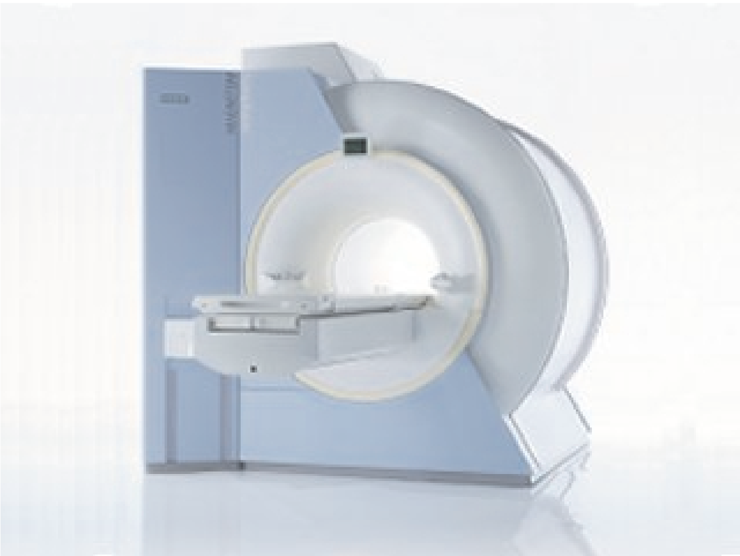

Abbildung 5.6: Ein Magnetresonanztomograph der Firma Siemens

DM) Anlagen, die aus einer Vielzahl unterschiedlicher Gerätekomponenten bestehen.

Folgende Anforderungen wurden in diesem Kontext an *DeviceInvestigator* gestellt:

1. Integration sehr unterschiedlicher Hardwarekomponenten
2. Nahtlose Integration in die bestehende Service-Architektur
   (a) Flexible Ausgabe der Zustandsdaten als *HTML*- und *XML*-Dokumente
   (b) Entfernte Konfigurationsmöglichkeit
   (c) Asynchrone Abbrechbarkeit
3. Plattform: WinNT, Java Runtime Environment 1.1.7b

Die Schnittstelle zu den einzelnen Geräten besteht jeweils aus spezialisierten, in C++ implementierten Proxy-Klassen, die den protokollspezifischen Hardwarezugriff auf die Geräte kapseln. Um diese Proxy-Klassen für *Device-*

Investigator zugänglich zu machen, wurden entsprechende *Port*- und *Parameter*-Klassen in Java sowie entsprechende schlanke Wrapper-DLLs in C++ implementiert. Die Wrapper-DLLs exportieren lediglich die öffentlichen Funktionen der Proxy-Klassen über das *Java Native Interface* [Lia99] in entsprechende Java-*Port*-Klassen. Eine generische *Parameter*-Klasse entnimmt der *DeMML*-Parameterbeschreibung jeweils den Namen und die Parameter der aufzurufenden nativen Proxy-Methode und ruft diese über *Java Reflection* auf dem assoziierten *Port*-Objekt auf.

Um *DeviceInvestigator* in die bestehende CGI/C++-basierte Service-Architektur nahtlos integrieren zu können, wurde ein kleines C++-Rahmenprogramm entwickelt, das die virtuelle Maschine der installierten Java-Laufzeitumgebung ermittelt, konfiguriert, instanziiert und die *main()*-Methode der *DeviceInvestigator*-Klasse aufruft. Dieses Rahmenprogramm bindet dann auch eine vorhandene C++-Bibliothek ein, die die Basis für die kontrollierte asynchrone Abbrechbarkeit der *DeviceInvestigator*-Komponente darstellt.

Um das Ausgabeformat von *DeviceInvestigator* flexibel variieren zu können, wurde die freie *XSLT*-Engine der *Xalan*-Bibliothek [Apad] in *DeviceInvestigator* integriert. Dadurch kann über externe *XSL*-Dokumente das Ausgabeformat von *DeviceInvestigator* in das bestehende *HTML*-Format oder zukünftige *XML*-Formate der Service-Software transformiert werden.

Das Konzept der entfernten Konfiguration von Service-Routinen über Zugriff auf *XML*-Konfigurationsdokumente war bereits in der Service-Architektur vorhanden, so dass die entfernte Konfiguration von *DeviceInvestigator* über Modifikation des *DeMML*-Konfigurationsdokuments problemlos realisiert werden konnte.

5.7.1 Ergebnisse

Es hat sich gezeigt, dass die *DeviceInvestigator*-Komponente aufgrund ihrer Offenheit und Skalierbarkeit problemlos in das bestehende System integrierbar war. Die Eigenschaften, dass keine mächtigen Datenerfassungssysteme (z.B. OPC-Server) vorausgesetzt werden und als Datenformat konsequent *XML* in Verbindung mit *XSLT* eingesetzt wird, erwiesen sich hierbei als besonders vorteilhaft. Ebenso hat sich gezeigt, dass die *XML*-Applikation *DeMML* ihrem universellen Anspruch gerecht wurde, da keinerlei Spracherweiterungen zur Repräsentation der Proxy-Klassen nötig waren. Die zusätzliche Integration von *XSLT*-Transformationen in *DeviceInvestigator* ermöglicht weiterhin die automatisierte Abbildung auf Siemens-interne und weitere Datenformate.

Kapitel 6

Client/Server-Architektur

Das INSIGHT-System integriert das generische *DeviceInvestigator*-Werkzeug in eine Client/Server-Architektur, bestehend aus dem INSIGHT-Server und zwei unterschiedlichen Internet-Schnittstellen für den INSIGHT-Applet-Client und gewöhnliche Web-Browser. Das so entstehende Internet-basierte Informationssystem unterstützt die dynamische Generierung, Übertragung und Visualisierung von Prozesszuständen in unterschiedlichen Sichten, den schreibenden Zugriff auf einzelne Geräteparameter sowie die Integration eines *XML*-Datenbanksystems.

Abbildung 6.1 zeigt die wichtigsten Komponenten der realisierten Client/Server-Architektur. Der INSIGHT-Server umhüllt die *DeviceInvestigator*-Komponente und erlaubt den entfernten Zugriff auf ihre Funktionalität über *Java Remote Method Invocation* (RMI) [Sun98]. Um unabhängig von Java-RMI auch direkt mit einem Web-Browser auf das System zugreifen zu können, ist dem INSIGHT-Server zusätzlich ein entsprechendes Servlet vorgeschaltet, das direkt eine HTTP-Schnittstelle nach außen anbietet.

6.1 Der INSIGHT-Server

Die zentrale Schaltstelle der INSIGHT-Server-Implementierung ist die Klasse *de.-wsi.insight.server.InsightServerImpl*. Wie in Abbildung 6.2 dargestellt, implementiert diese Klasse drei von *java.rmi.Remote* abgeleitete Schnittstellen: *Insight-Interface*, *InsightServletInterface* und *KeyArbitration*. Die ersten zwei Schnittstellen umhüllen die *DeviceInvestigator*-Komponente für direkten Zugriff über Java-RMI sowie über eine Servlet-Zwischenstufe und erlauben so den entfernten Zugriff auf ihre Funktionalität. Die *main()*-Methode erzeugt hierzu eine statische In-

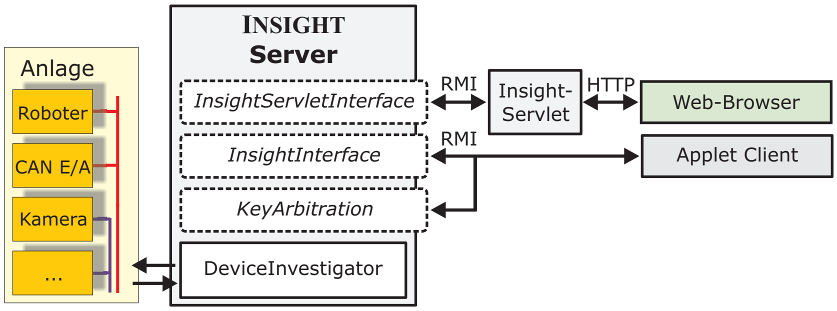

Abbildung 6.1: Vereinfachte Client/Server-Architektur des INSIGHT-Systems

stanz der *DeviceInvestigator*-Klasse und startet deren Initialisierungsphase (vgl. Abschnitt 5.4.1) durch die Übergabe eines *DeMML*-Konfigurationsdokuments.

Die *KeyArbitration*-Schnittstelle erlaubt die Implementierung eines zeitscheibenbasierten Sitzungsmanagements. Sie ist Bestandteil des *ServerArbitration*-Frameworks, das im nächsten Abschnitt vorgestellt wird.

6.1.1 Das *ServerArbitration*-Framework

Dieses Software-Framework, das durch das Paket *de.uni_tuebingen.ServerArbitration* realisiert ist, unterstützt ein Sitzungsmanagement für Java-RMI-Architekturen. Das Ziel ist es, ein zeitscheibenbasiertes Server-Zuteilungsverfahren bereitzustellen, das die Server-Zugriffe konkurrierender Clients synchronisiert. Das Problem hierbei ist, einzelne entfernte Methodenaufrufe bestimmten Client-Instanzen zuzuordnen, da auf Server-Seite innerhalb des Methodenrumpfes einer via RMI zugänglichen Methode durch die Java-Klassenbibliothek in der Version 1.1 nur der Zugriff auf die Client-IP, nicht aber auf den benutzten Socket-Port unterstützt wird. Es ist demnach für den Server zunächst unmöglich, zwei unterschiedliche Clients, die auf demselben Rechner ausgeführt werden, zu unterscheiden.

Zur Lösung dieses Problems muss ein Client zunächst eine innerhalb des *ServerArbitration*-Frameworks festgelegte Login-Prozedur abarbeiten, bevor er Zugang zu dem entsprechenden RMI-Server erteilt bekommt. Diese Login-Prozedur kann automatisch durch ein Objekt der Klasse *LoginThread* (vgl. Abbildung 6.3) des *ServerArbitration*-Frameworks ausgeführt werden. Dieses Objekt ruft über die *KeyArbitration*-Schnittstelle wiederholt die Methode *login()* (vgl. Abbildung 6.2) auf, bis diese einen gültigen Schlüssel (*Session-Ticket*) in

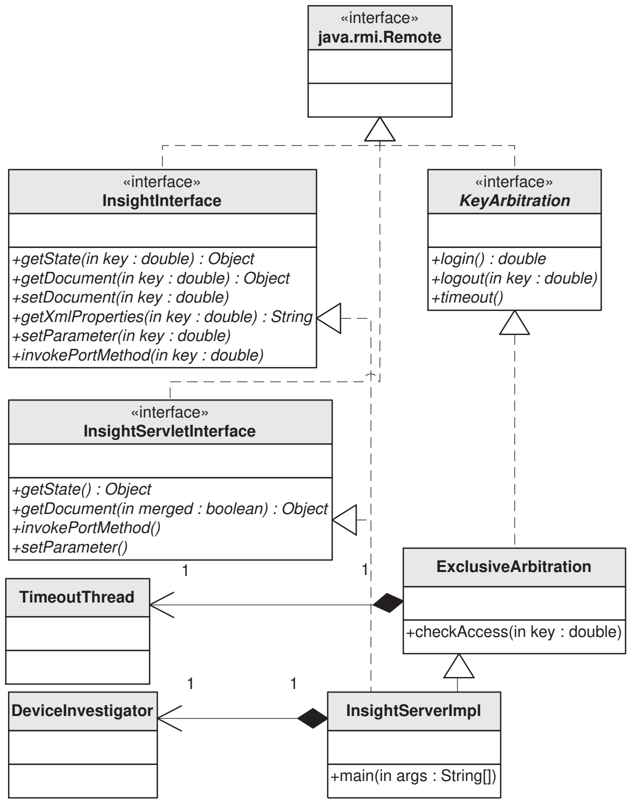

Abbildung 6.2: Die zentrale *InsightServerImpl*-Klasse

Form eines eindeutigen *Double*-Wertes zurückliefert. Innerhalb einer Implementierung der *login()*-Methode muss daher zunächst geprüft werden, ob der Server momentan belegt ist oder ein neuer, für eine beschränkte Zeit gültiger Schlüssel zurückgegeben werden kann. Die Klasse *TimeoutThread* ist ein Java-Thread, der nach einer parametrierbaren Zeit die Methode *timeout()* der *KeyArbitration*-Schnittstelle aufruft, um den momentan gültigen Schlüssel zu invalidieren. Der entsprechende Client muss sich dann erneut beim Server anmelden. Die Klasse *ExclusiveArbitration* bietet eine mögliche Implementierung der *KeyArbitration*-Schnittstelle und realisiert ein exklusives Ausschlussverfahren, welches jeweils für die Dauer einer spezifizierten Zeitspanne nur einem bestimmten Client exklusiven Zugriff auf den Server erlaubt.

Anhand des eindeutigen Schlüssel-Wertes kann der RMI-Server nun einzelne Clients identifizieren und voneinander unterscheiden. Voraussetzung hierfür ist allerdings, dass die Clients diesen Schlüssel bei jedem Methodenaufruf als Parameter übergeben müssen. Dies ist durch die Definition der *InsightInterface*-Schnittstelle (siehe Abbildung 6.2) gewährleistet. Die *InsightServletInterface*-Schnittstelle, die von der *InsightServlet*-Klasse angesprochen wird, benötigt kein explizites Sitzungsmanagement (und damit keine Schlüsselparameter), da dieses bereits vom Web-Server geleistet wird.

Auf Client-Seite benötigt der bereits erwähnte *LoginThread* neben der Referenz auf die für ihn entfernte *KeyArbitration*-Schnittstelle eine Referenz auf eine Implementierung der *LoginInterface*-Schnittstelle, um beliebige Clients über den momentanen Status der Login-Prozedur unterrichten zu können (siehe Abbildung 6.3).

6.2 Der INSIGHT-*Applet-Client*

Der INSIGHT-*Applet-Client* basiert auf direktem Zugriff auf den INSIGHT-Server über Java-RMI. Die übergebenen Zustandsinformationen werden durch unterschiedliche GUI-Komponenten visualisiert und erlauben den interaktiven Zugriff auf einzelne Geräteparameter, Management-Funktionalität und die Konfiguration des INSIGHT-Server bzw. der *DeviceInvestigator*-Komponente.

6.2.1 Kommunikation

Die Klasse *InsightApplet* (siehe Abbildung 6.3) ist die von *java.applet.Applet* abgeleitete Basisklasse dieses Java-Applets. Sie ermittelt die entfernten Implementierungen der *KeyArbitration* und *InsightInterface*-Schnittstellen über die Java-Registry. Diese Referenzen werden an ein Objekt der Klasse *KeyHolder* weitergereicht, das im Folgenden für die Verwaltung des Sitzungsschlüssels und für die Abwicklung der gesamten Client/Server-Kommunikation verantwortlich ist.

In der Initialisierungsphase des Applets startet das *KeyHolder*-Objekt zunächst einen neuen *LoginThread*, der nach einer erfolgreichen Server-Anmeldung den von *InsightSeverImpl* erhaltenen Schlüssel über die *LoginInterface*-Schnittstelle (vgl. Abbildung 6.3) an das *KeyHolder*-Objekt übergibt. Zum Abschluss der Initialisierungsphase wird die momentane Konfiguration der *DeviceInvestigator*-Komponente über die *getDocument()*-Methode der *InsightInterface*-Schnittstelle angefordert.

Wenn ein Benutzer im Folgenden den momentanen Zustand der entfernten Anlage

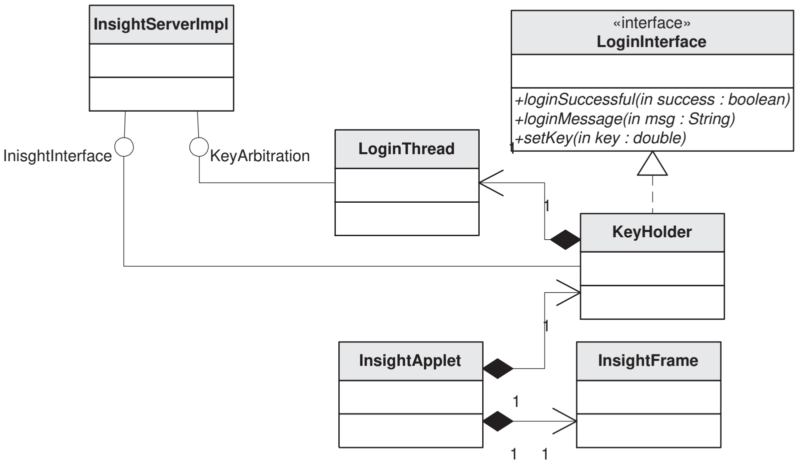

Abbildung 6.3: Der INSIGHT-*Applet-Client*

anfragt, wird über die Methode *getState()* die Erzeugung eines neuen *Snapshot*-Dokuments innerhalb von *DeviceInvestigator* veranlasst. Das generierte *Snapshot*-Dokument wird dann als Java-Objekt an den Client übergeben. Der Client parst das erhaltene Java-Objekt wieder in ein *DeMML*-DOM und mischt die darin enthaltenen Parameterwerte in das in der Initialisierungsphase erhaltene Konfigurations-DOM. Daraufhin werden sämtliche betroffenen Sichten auf die Daten gemäß dem Observer-Pattern [GHJV95] aufgefordert, sich zu aktualisieren (vgl. Abschnitt 4.6.2).

Wenn der Benutzer das Konfigurations-DOM modifiziert, wird bei der nächsten Zustandsanfrage zunächst das modifizierte Konfigurationsdokument vom Client an den Server übertragen, woraufhin der Server eine entsprechende Rekonfigurierung der *DeviceInvestigator*-Komponente (vgl. Abschnitt 5.5.4) durchführt. Danach wird dann der Scan gemäß der neuen Konfiguration durchgeführt und das Ergebnis wie oben beschrieben an den Client zurückgegeben.

6.2.2 Benutzerschnittstellen

Zusätzlich zu der *KeyHolder*-Klasse instanziert die *InsightApplet*-Klasse ein Objekt der Klasse *InsightFrame*, welches die einzelnen GUI-Komponenten des Applets mit Hilfe eines *DeMMLDataModel*-Objekts (siehe Abschnitt 4.6.2) verwal-

Sicht	Beschreibung	Interaktion
Tree	Hierarchische Darstellung der kompletten *DeMML*-Information	Editieren der Konfiguration
Parameter	Tabellarische Darstellung der definierten *Parameter*-Elemente	Schreibender Zugriff auf einzelne Parameter. Editieren der Parameterkonfiguration
Monitoring	Tabellarische Darstellung der *DataAccess* Konfiguration	Editieren der *DataAccess* Konfiguration
Configuration	Darstellung der Anlagenbeschreibung	Editieren der Anlagenbeschreibung
Device	Darstellung der Gerätebeschreibungen	Editieren der Gerätebeschreibungen
Port	Generische Darstellung der definierten Kommunikationskanäle	Zugriff auf Management Funktionalität der einzelnen Ports (*PortMethod*s)
DeMML	Textuelle Darstellung der *DeMML*-Konfiguration	-
Snapshot	Textuelle Darstellung des aktuellen *Snapshot*-Dokuments	-

Tabelle 6.1: Die einzelnen Sichten des INSIGHT-*Applet-Client*s

tet und koordiniert. Hierzu werden die bereits vorgestellten GUI-Komponenten des *DeMML*-Java-Pakets in ein entsprechendes *JTabbedPane*-Objekt der *Swing*-Bibliothek eingefügt (vgl. Abbildung 4.7 auf Seite 61). Die einzelnen Sichten sind dadurch direkt über die entsprechenden Registerkarten zugänglich. Tabelle 6.1 listet die einzelnen Sichten zusammen mit den jeweils unterstützten Interaktionsmöglichkeiten auf.

Die meisten Sichten unterstützen eine Volltextsuche durch eine sichtspezifische Implementierung der Schnittstelle *de.wsi.misc.SearchDialogHandler*. Die in Tabelle 6.1 aufgelisteten Editierungsmöglichkeiten beruhen alle auf der in Abschnitt 4.6.2 vorgestellten *ElementNodePanel*-Klasse. Sämtliche Veränderungen des zu Grunde liegenden *DeMML*-Dokuments werden jeweils den anderen Sichten gemäß dem Observer-Pattern mitgeteilt und somit konsistent gehalten.

Die bereits in Abbildung 4.7 auf Seite 61 gezeigte Parameter-Sicht erlaubt den direkten schreibenden Zugriff auf einzelne Parameter. Der Benutzer kann den zu schreibenden Wert direkt in die entsprechende Zelle der Tabelle eintragen und damit auf das Gerät durchschreiben. Abbildung 6.4 zeigt die Port-Sicht, die den Zugriff auf durch *DeMML-PortMethod*-Elemente publizierte Manage-

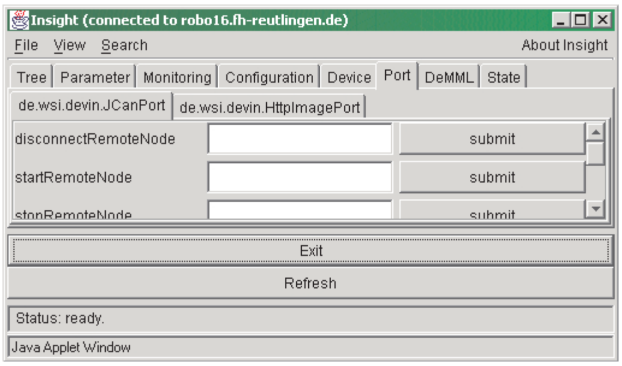

Abbildung 6.4: Die *Port*-Sicht des INSIGHT-*Applet-Client*s

ment Funktionen (siehe Abschnitt 5.5.3) einzelner Ports unterstützt. Der in Abbildung 6.4 gezeigte *JCanPort* Port erlaubt beispielsweise so den Zugriff auf CANopen-spezifische *Network Management* (NMT) [Can96d] Funktionalität. Als Parameter kann jeweils eine einzelne Zeichenkette angegeben werden.

Zu jedem Zeitpunkt kann der Benutzer ein neues *Snapshot*-Dokument anfordern, wodurch die einzelnen Sichten gemäß dem aktuellen, entfernten Anlagenzustand aktualisiert werden.

6.2.3 Datenbankintegration

Zur persistenten Archivierung von *DeMML*-Konfigurations- und *Snapshot*-Dokumenten unterstützt der INSIGHT-*Applet-Client* den Zugriff auf entfernte Datenbanken via HTTP und mit Einschränkungen via *Java Database Connectivity* (JDBC) [WFC+99].

6.2.3.1 Anbindung relationaler Datenbanksysteme via JDBC

Das Vorgängersystem CANINSIGHT benutzte zunächst ausschließlich eine JDBC-Schnittstelle, um *XML*-Daten in einem relationalen Datenbanksystem (IBM DB2) abzulegen [BK00]. Hierzu wurde eine Abbildung der *CANopen Markup Language* [Büh00], die eine Vorstufe der *Device Management Markup Language* bildete (vgl. Abschnitt 4.1), auf ein relationales Datenmodell definiert. Abbildung 6.5 zeigt das *Entity-Relationship* (ER) [Che88] Diagramm[1] dieses *XML*-Formats.

[1] Die verwendete Darstellung des ER Diagramms ist analog zu [Dat95].

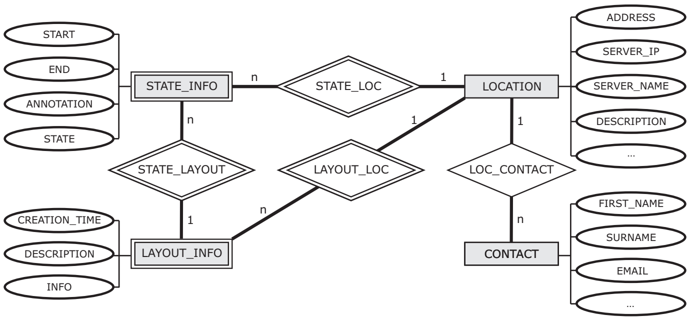

Abbildung 6.5: Das relationale Datenmodell des CANINSIGHT-Systems

Um ein effizientes und dennoch möglichst flexibles Data-Retrieval zu unterstützen, musste ein Kompromiss zwischen der vollständigen Abbildung der hierarchischen *XML*-Struktur auf ein entsprechendes relationales Model und der Verwaltung der ganzen *XML*-Struktur in jeweils einem einzelnen *Character Large Object* (CLOB) gewählt werden. Es wurden daher bestimmte *XML*-Fragmente, die häufig in Datenbankanfragen verwendet wurden, auf separate Attribute entsprechender Entities (z.B. das *CREATION_TIME*-Attribut der *LAYOUT_INFO* Entity) abgebildet. Die restlichen Teile wurden dann jeweils als CLOBs verwaltet, die durch gewöhnliche *XML*-Parser wieder in ein entsprechendes DOM transformiert werden konnten.

6.2.3.2 Anbindung eines nativen *XML*-Datenbanksystems via HTTP

Im Zuge der Weiterentwicklung des CANINSIGHT-Systems hin zum universellen INSIGHT-System wurde parallel zu der JDBC-basierten Datenbankanbindung eine Anbindung des *XML*-Datenbanksystems *Tamino* über das HTTP-Protokoll implementiert. Die transparente Schnittstelle zu beiden Zugangswegen bildet die *de.wsi.insight.client.DatabaseAccessManager*-Schnittstelle zusammen mit den Implementierungen *JDBCAccessHandler* und *TaminoAccessHandler* (vgl. Abbildung 6.6). Die Klasse *DatabaseHandler* kann somit je nach Bedarf auf *Tamino*-Datenbanken oder relationale Datenbanken zugreifen.

Tamino-Datenbankanfragen werden in Form von sog. *X-Query Statements* als Parameteranteil von entsprechenden URLs an das *Tamino*-System gegeben und dort ausgewertet. *Tamino X-Query* ist ein auf *XQL* [RLS98], *XPath* [CD99] und *XML Query* [MF00] basierendes Anfrageformat zur Selektion von spezifischen *XML*-Fragmenten aus spezifischen *XML*-Dokumenten. Diese Selektion ist im Wesentlichen durch die Angabe von einzelnen Pfaden in die *XML*-Baumstruktur definiert.

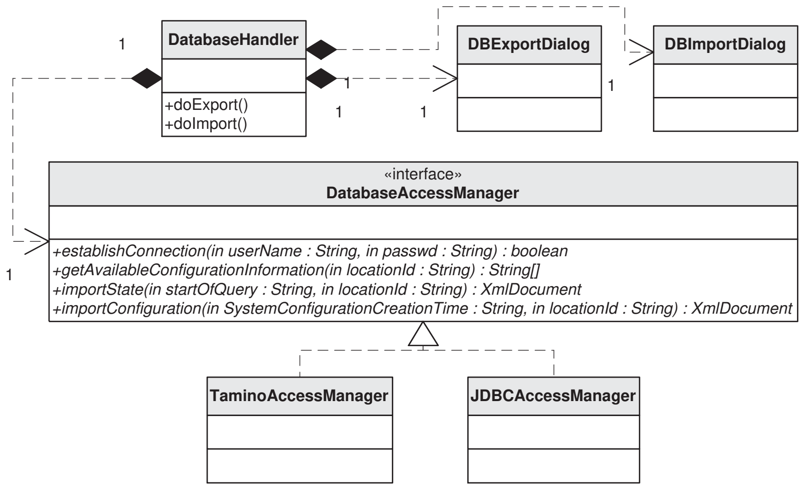

Abbildung 6.6: Ein Ausschnitt aus der *DatabaseAccessHandler*-Schnittstelle

Die Pfade können hierbei in Analogie zu SQL-*WHERE*-Klauseln zusätzlich mit Prädikaten verknüpft werden (vgl. Abschnitt 7.3.3).

Abbildung 6.7 zeigt den Internet Explorer beim Zugriff auf eine *Tamino*-Datenbank. Die Query ist in der URL, die im Adressfeld des Browsers zu sehen ist, enthalten. Diese spezielle Query selektiert beispielsweise alle *DeMML*-Dokumente der Collection `data` der Datenbank `DeMML`, die ein `Snapshot` Subelement mit einer `LocationId` von `0042` haben.

Tamino ist zusätzlich ein sog. natives *XML*-Datenbanksystem und verwaltet *XML*-Daten tatsächlich auch in hierarchischen Datenstrukturen. Dies verspricht eine höhere Performance als relationale Datenbanksysteme mit *XML*-Erweiterungen, wie z.B. eine IBM DB2-Datenbank mit XML-Extender. Indexstrukturen können über die Definition eines entsprechenden Datenbankschemas erstellt und konfiguriert werden, wobei auf die logische Struktur der zu erwartenden *XML*-Dokumente Bezug genommen werden kann. Es können beispielsweise volltext- oder datentypbasierte Indexstrukturen über einzelne Attributwerte oder Elemente angelegt werden.

Für *DeMML* wurde ein entsprechendes Schema entworfen und in der Datenbank installiert. Dieses Schema definiert insbesondere Indexstrukturen für die Attribute,

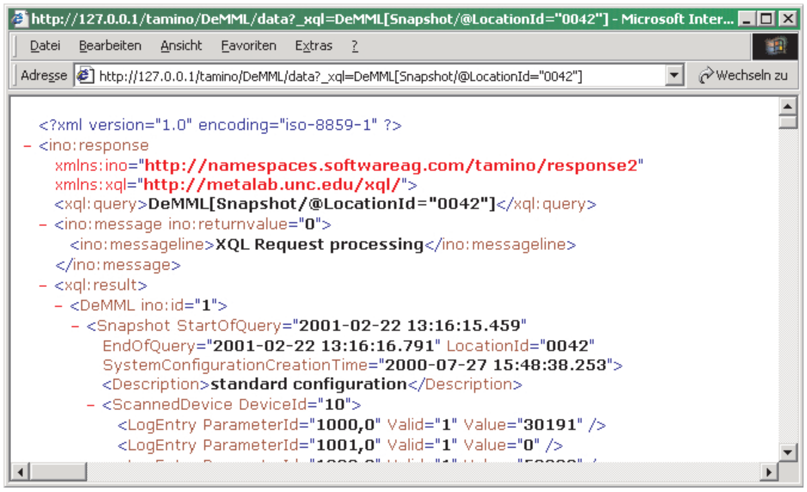

Abbildung 6.7: Ergebnis einer *Tamino*-X-Query

die als Fremdschlüssel in weitere Dokumente verwendet werden (z.B. *LocationId* und *SystemConfigurationCreationTime*). Im Gegensatz zu relationalen Datenbanken werden die durch die konzeptionelle Verwendung von Fremdschlüsseln entstehenden Integritätsanforderungen (*Referential Integrity*) jedoch meist von *XML*-Datenbanken nicht überwacht.

Die Verwendung eines nativen *XML*-Datenbanksystems in Verbindung mit einer HTTP-Schnittstelle bietet diverse Vorteile gegenüber der JDBC-Schnittstelle:

1. Die Abbildung des hierarchischen *XML*-Datenmodells auf ein relationales Modell wird obsolet

2. Die Flexibilität des Datenmodells wird erhöht

3. HTTP-Unterstützung ist Bestandteil der Standard-Java-Klassenbibliothek im Gegensatz zu datenbankabhängigen JDBC-Treibern

4. Jeder Web-Browser ist automatisch ein voll funktionsfähiger Datenbank-Client

5. Unterstützung von effizientem *XML*-spezifischem Daten-Retrieval durch eine X-Query-Engine auf Datenbankseite

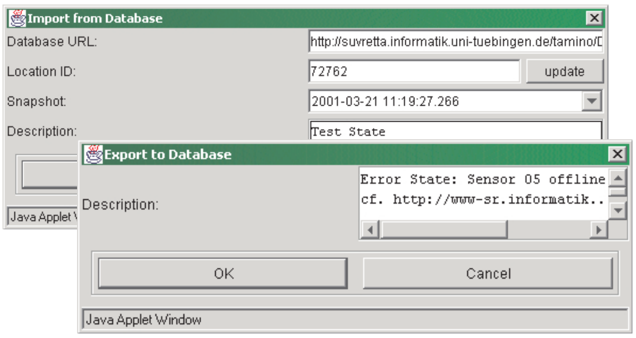

Abbildung 6.8: Export/Import-Dialoge des INSIGHT-*Applet-Client*

Aufgrund dieser Vorteile unterstützt das INSIGHT-System die JDBC-Schnittstelle nur noch rudimentär und favorisiert den HTTP/X-Query-basierten Ansatz. Insbesondere erhöht die HTTP-Datenbankschnittstelle auch die Skalierbarkeit des Systems, da auf Datenerfassungsseite die Standard-Java-Klassenbibliothek genügt, um erfasste Daten persistent in entfernten Datenbanken abzulegen.

Implementierung Die Implementierung der *Tamino*-spezifischen HTTP-Erweiterungen wurde zu einem Java-Paket (*de.wsi.tamino*) zusammengefasst, das unter Verwendung der *Tamino*-Java-Unterstützung u.a. die folgenden Anforderungen erfüllt:

1. Einfügen und Löschen von *XML*-Dokumenten und multimedialen Non-*XML*-Daten

2. Daten-Retrieval über X-Query/*XQL* Statements

3. Unterstützung eines abstrakten Ergebnis-Cursors zum Traversieren großer Ergebnismengen

Der INSIGHT-*Applet-Client* nutzt dieses Paket zum Importieren und Exportieren von *DeMML*-Konfigurations- und *Snapshot*-Dokumenten. Abbildung 6.8 zeigt die entsprechenden Dialoge, die die Parametrierung des Datenbankzugriffs erlauben.

Exportieren von *DeMML*-Information Wie in Abschnitt 4.2.1 beschrieben, sind *DeMML*-Konfigurationsdokumente durch die Attribute *LocationId* und *CreationTime* eindeutig identifizierbar. Jedes der einen Anlagen-Zustand kodierende

Snapshot-Dokumente referenziert das verantwortliche Konfigurationsdokument über diese beiden Attribute. Wenn ein aktueller Zustand in ein Datenbanksystem exportiert werden soll, prüft der Client zunächst, ob das entsprechende *DeMML*-Konfigurationsdokument bereits in der Datenbank vorliegt. Ist dies der Fall, wird nur das *Snapshot*-Dokument in die Datenbank geschrieben, andernfalls wird vorher zusätzlich das entsprechende *DeMML*-Konfigurationsdokument exportiert.

Bevor der Zustand exportiert wird, kann der Benutzer noch den Text des entsprechenden *Description*-Elements (vgl. Abschnitt 4.2.2) anpassen, um beispielsweise, wie in Abbildung 6.8 gezeigt, auf zusätzliche externe Informationen über diesen Zustand zu verweisen. Wenn der angegebene Verweis eine URL ist, kann das Ziel wieder in einer *Tamino*-Datenbank liegen und z.B. eine Fehlerbeschreibung für einen Fehlerzustand darstellen.

Importieren von *DeMML*-Information Beim Importieren von *DeMML*-Zustandsinformationen muss der Benutzer zunächst die Basis-URL der *Tamino*-Datenbank und den Wert des *LocationId*-Attributes angeben. Basierend auf diesen beiden Angaben werden die *StartOfQuery*-Attribute und *Description*-Elemente sämtlicher *Snapshot*-Dokumente, die die spezifizierte *LocationId* tragen, aus der angegebenen Datenbank importiert und in entsprechenden GUI-Komponenten abgelegt (vgl. Abbildung 6.8). Der Benutzer kann daraufhin über die *StartOfQuery* Zeitstempel oder die Zustandsbeschreibungen ein spezielles *Snapshot*-Dokument zum Import auswählen.

Beim Importieren des selektierten *Snapshot*-Dokuments prüft der Client zunächst, ob das referenzierte *DeMML*-Konfigurationsdokument bereits im Client vorliegt. Falls dies nicht der Fall ist, wird dieses Dokument noch vor dem *Snapshot*-Dokument importiert.

6.3 Der HTTP-Client

Ein INSIGHT-HTTP-Client implementiert den zweiten Zugangsweg zu einem INSIGHT-System. Als HTTP-Clients können alle gängigen Web-Browser verwendet werden. Im Gegensatz zum INSIGHT-*Applet-Client* setzt dieser Zugangsweg keine Java-Laufzeitumgebung auf der Client-Seite voraus, sondern delegiert die Generierung unterschiedlicher Sichten an die Server-Seite. Die einzelnen Sichten sind jeweils durch entsprechende *XSLT*-Dokumente auf der Server-Seite definiert. Diese *XSLT*-Dokumente legen die sichtspezifische Transformation der *DeMML*-Dokumente in entsprechende *HTML*-Dokumente fest, die dann direkt im Browser visualisiert werden können. Der Browser fordert hierzu die entsprechenden Dokumente über herkömmliche HTTP-Requests an, die durch den Web-Server an das

INSIGHT-System weitergeleitet werden.

6.3.1 Kommunikation

Wie bereits in Use-Case 1 beschrieben, laufen die einzelnen HTTP-Zustandsanfragen des HTTP-Clients über die folgenden Stationen:

1. Web-Server
2. Cocoon
3. INSIGHT-*Servlet*
4. INSIGHT-Server

Definiert ist dieser Ablauf durch die URL-Parameter der HTTP-Anfrage in Verbindung mit der Konfiguration des Web-Servers (Apache, www.apache.org), der Servlet-Engine (JServ, java.apache.org/jserv) und des Cocoon-Servlets (xml.apache.org/cocoon). Eine derartige Anfrage kann beispielsweise wie folgt aussehen:

```
http://suvretta.informatik.uni-tuebingen.de/\
insight.xml?producer=insight&extended=true&\
style=styles/parameter.xsl
```

Diese URL fordert zunächst das Dokument *insight.xml* auf dem angegeben Rechner an. Innerhalb der Konfiguration des Web-Servers ist festgelegt, dass Anfragen an Dokumente mit der Endung *.xml* grundsätzlich nicht durch Apache selbst, sondern durch das Servlet Cocoon bearbeitet werden. Cocoon [Apab] ist ein frei verfügbares Java-Servlet zur Auswertung von *XSL*-Transformationen auf *XML*-Dokumenten. Cocoon analysiert den Parameterteil der erhaltenen Anfrage und ermittelt gemäß dem *producer*-Parameter, dass das angefragte Dokument von einer weiteren Java-Klasse, die durch das Alias *insight* identifiziert ist, geliefert werden wird. Innerhalb der Konfiguration von Cocoon wird das Alias *insight* auf die Klasse *de.wsi.xsl.InsightProducer* abgebildet, die von der abstrakten Klasse *AbstractProducer* des *Cocoon*-Java-Pakets abgeleitet ist (vgl. Abbildung 6.9). Cocoon ruft auf dieser Klasse die Implementierung der Methode *getStream()* auf und liest von dem gelieferten *Reader*-Objekt das angefragte *XML*-Dokument.

Die *InsightProducer*-Klasse analysiert innerhalb der *getStream()*-Methode die obige Anfrage URL und gibt aufgrund dieser Analyse einen Stream auf die folgende URL zurück:

```
http://127.0.0.1/servlets/insight?\
extended=true&style=styles/parameter.xsl
```

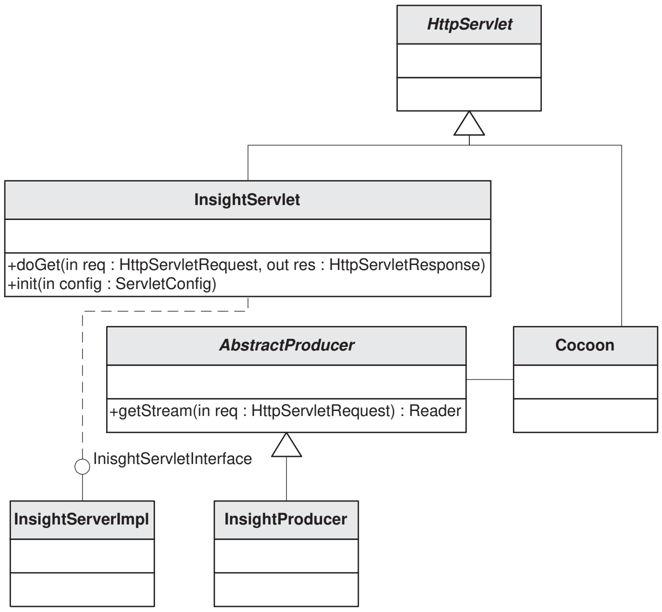

Abbildung 6.9: Die *InsightServlet*-Klasse

Diese Anfrage wird wieder von Apache entgegengenommen und gemäß dem Alias *servlets* an die Servlet-Engine JServ weitergereicht. Innerhalb der Konfiguration von JServ wird dann das Alias *insight* auf die Klasse *InsightServlet* (vgl. Abbildung 6.9) abgebildet und der Request in Form eines *HttpServletRequest*-Objekts an die Methode *doGet()* dieses Servlets weitergereicht.

Das INSIGHT-*Servlet*, implementiert durch die Klasse *InsightServlet*, analysiert die einzelnen URL-Parameter und erfragt beim INSIGHT-Server über das *InsightServletInterface* den aktuellen Anlagenzustand. Falls der Parameter *extended* auf den Wert *true* gesetzt ist, wird zunächst ein neuer Scan gestartet, das generierte *Snapshot*-Dokument in das Konfigurationsdokument gemischt und dann das Ergebnis an das *InsightServlet*-Objekt zurückgegeben. Falls der Parameter den Wert *false* trägt, wird nur das generierte *Snapshot*-Dokument übertragen. In jedem Fall wird in das erhaltene *DeMML*-Dokument eine Referenz auf ein *XSLT*-Dokument gemäß dem *style* Parameter (in unserem Beispiel wäre dies *styles/parameter.xsl* vgl. Anhang A.9) eingefügt. Das so generierte *DeMML*-Dokument wird anschließend über das *HttpServletResponse*-Objekt der *doGet()*-Methode als Ergebnis an

Cocoon zurückgeliefert.

Das Cocoon-Servlet prüft das gelieferte *XML*-Dokument auf Referenzen auf *XSLT*-Dokumente und wertet die dort enthaltenen Transformationsanweisungen auf dem erhaltenen *DeMML*-Dokument aus. Zu jeder durch den HTTP-Client unterstützten Sicht existiert ein entsprechendes *XSLT*-Dokument, das die Transformation des *DeMML*-Dokuments in ein entsprechendes *HTML*-Dokument spezifiziert. Das Ergebnis dieser Transformation wird als *HTML*-Dokument an den Web-Browser zurückgegeben und dort dargestellt. Für den Web-Browser ist damit die Verwendung von *XML* als Datenformat völlig transparent. Abbildung 6.10 zeigt ein Ergebnis dieses Aufrufs, dargestellt in Netscape Navigator.

6.4 Fallstudie 3: Integration eines Telelabors in das INSIGHT-System

Die Geräte, die im Rahmen des VVL-Projekts (vgl. Kapitel 12) innerhalb einer Internet-basierten Lehr-/Lernumgebung verwendet werden, wurden zusätzlich über das INSIGHT-System zu einem Internet-basierten Informationssystem zusammengefasst. Die Homepage dieses Systems ist über *http://www-sr.informatik.uni-tuebingen.de/Insight/* rund um die Uhr zugänglich. Im Einzelnen sind die folgenden Geräte integriert:

1. Selectron CANopen Dioc711 E/A-Modul (Werkstückvereinzelung)
2. SelectronCANopen Dioc711 E/A-Modul (Lichtsteuerung)
3. SelectronCANopen Dioc711 E/A-Modul (Leuchtschrift)
4. Beckhoff CANopen Bk5110 E/A-Modul
5. Sony EVI D31 (steuerbare Digitalkamera)

Die verwendeten Hardwarezugriffsklassen entsprechen im Wesentlichen den in Abschnitt 5.5.2.3 vorgestellten Beispielen. Das durch das INSIGHT-System geschaffene Portal ermöglicht damit einen schnellen Überblick über den momentanen Zustand der Vereinzelungseinheit (siehe Abschnitt 12.3), der Leuchtschrift (siehe Abschnitt 12.2), über den aktuellen Blickwinkel und die Zoom-Einstellung der Kamera sowie das aktuelle Kamerabild.

Zur zusätzlichen Einbindung der Roboterzelle aus Fallstudie 1 (vgl. Abschnitt 5.6) in das System ist lediglich das Einfügen der entsprechenden Gerätebeschreibungen (d.h. der *Device-DeMML*-Elemente) in das Konfigurationsdokument notwendig. Da die Roboterzelle bei unsachgemäßer Bedienung jedoch potentielle Risiken für

Abbildung 6.10: Die *Parameter* Sicht eines HTTP-Clients

Geräte und Menschen birgt, wurde sie nicht in die Standardkonfiguration aufgenommen.

6.5 Alternative Ansätze

6.5.1 *Simple Network Management Protocol* (SNMP)

Das *Simple Network Management Protocol* (SNMP) wurde entwickelt, als 1990 immer deutlicher wurde, dass mit der zunehmenden Verbreitung des Internet und der damit einhergehenden steigenden Zahl an Netzwerkkomponenten (Router, Switches etc.) ein standardisiertes Konzept zur Administration und Überwachung dieser Geräte erforderlich wurde [Tan96]. Der SNMP-Standard ist durch die drei Versionen SNMPv1 (RFC[2] 1157, 1155), SNMPv2 (RFC 1441 - 1452) und SNMPv3 (RFC 2570 - 2575) festgelegt. Die Versionen 1 und 2 definieren die grundlegende Architektur von SNMP, während SNMPv3 Sicherheits- und Administrationskonzepte (z.B. Verschlüsselungsverfahren und Benutzergruppenverwaltung) hinzufügt.

Das Ziel des SNMP-Ansatzes ist es, standardisierte Schnittstellen, Datenprotokolle und Managementservices zu definieren, die ein einheitliches Management von Geräten unterschiedlicher Hersteller auf unterschiedlichen Plattformen erlauben. Viele kommerzielle Werkzeuge sind mittlerweile im Handel erhältlich, die jeweils einheitliche Management-Konsolen für unterschiedliche SNMP-fähige Geräte anbieten.

Das SNMP-Modell für das Management eines Netzwerks besteht aus vier Teilen:

1. Managed Nodes
2. Management Stations
3. Management Information
4. Management Protocol

Unter *Managed Nodes* versteht man hierbei die zu verwaltenden Geräte, wie z.B. Hosts, Bridges oder Drucker. Damit ein Gerät via SNMP gemanagt werden kann, muss es einen entsprechenden SNMP-Agenten ausführen, der den Zugriff auf die einzelnen Geräteparameter verwaltet. Eine *Management Station* kann dann Kontakt zu einzelnen SNMP-Agenten aufnehmen und über das SNMP-Protokoll mit diesen kommunizieren, um bestimmte Parameterwerte zu erfragen oder zu setzen. Die einzelnen Informationen, die durch einen Agenten verwaltet werden, bilden die *Management Information* eines Geräts und sind jeweils in einer sog. *Management Information Base* (MIB) definiert.

[2] Die entsprechenden *Request for Comments* (RFC) Dokumente sind beispielsweise unter http://www.ietf.org/rfc/ online zugänglich.

Ein großer Teil des SNMP-Standards befasst sich mit der Spezifikation dieser MIBs und der entsprechenden Datentypen bzw. Datentypkodierungen. Dies ist insbesondere deshalb wichtig, weil Geräte unterschiedlicher Hersteller auf dieselbe Informationsrepräsentation abgebildet werden sollen. Ein Router sollte beispielsweise unabhängig von Hersteller und CPU-Architektur die Information, wieviele TCP Pakete seit einem bestimmten Zeitpunkt verloren gingen, allgemein verständlich darstellen können. Das Konzept zur Spezifikation dieser Informationen wird *Structure of Management Information* (SMI, RFC 1442) genannt und basiert auf einer erweiterten Teilmenge der von der OSI entwickelten *Abstract Syntax Notation One* (ASN.1, OSI 8824).

Prinzipiell ist SNMP nicht auf das Management von Netzwerkkomponenten beschränkt, wird aber dennoch hauptsächlich in diesem Bereich eingesetzt. Dies liegt teilweise auch in der Definition der standardisierten Informationsstrukturen, die einen starken Fokus in Richtung Netzwerkinformationen haben, begründet.

6.5.1.1 Abgrenzung

Das Ziel von SNMP ist die Standardisierung von Schnittstellen und Datenprotokollen auf syntaktischer und semantischer Ebene zur Erleichterung des Datenaustauschs über Herstellergrenzen und Plattformgrenzen hinweg. Eine ähnliche Zielsetzung verfolgen auch Teile des INSIGHT-Systems, jedoch mit einigen entscheidenden Unterschieden. INSIGHT bedient sich an Stelle einer großen Menge von hardwareabhängigen Spezifikationen (in [SNM01] wird von ca. 10 000 standardisierten Parameterspezifikationen gesprochen) eines universellen *XML*-basierten Datenformats, welches sämtliche Parameterwerte in textueller Form verwaltet und die Logik zur (Rück-)Transformation dieser Werte jeweils durch Referenzen auf externe Java-Klassen realisiert. Dieser Ansatz erlaubt die Identifizierung und Interpretation einzelner Geräteparameter zwar nicht global, sondern immer nur in Bezug auf einen bestimmten Konfigurationskontext, ist dadurch aber sowohl flexibler als auch skalierbarer.

INSIGHT kann deswegen beispielsweise eine einfache universelle Datenbankintegration anbieten, deren Datenbankschema auf dem universellen Datenformat beruht und für sämtliche Anlagenkonfigurationen sofort verwendet werden kann. Das *XML*-Format beinhaltet weiterhin automatisch ein einfaches Datenprotokoll zur Übertragung der Daten über Netzwerke auf unterschiedlichen Plattformen, da alle Werte als serialisierte Zeichenketten verwaltet werden. Die SNMP-SMI ist im Gegensatz dazu als ein auf ASN.1 basierendes Konzept sehr komplex und schwer zu handhaben [Tan96]. Die Verwendung von *XML* allgemein bringt zusätzlich die bereits in Abschnitt 2.2 erwähnten Vorteile mit sich.

Da die Transformation von hardwarespezifisch kodierten Geräteparameterwerten in textuelle Darstellungen an externe Java-Klassen delegiert wird, kann feingranular entschieden werden, welche Datentypen und Kodierungen für ein (evtl. sehr kleines) System unterstützt werden sollen, wodurch die Skalierbarkeit des Systems erhöht wird. INSIGHT delegiert diese Transformation an die Hardwarezugriffsklassen (vgl. Abschnitt 5.5.1), die unabhängig von INSIGHT entwickelt und integriert werden können. Die Trennung von plattformabhängiger Software und plattformunabhängiger Infrastruktur ist damit klar definiert und somit flexibel erweiterbar. In einem SNMP-System muss die Bedeutung und die interne Kodierung jedes Parameters dem gesamten System bekannt sein, was den Zugriff auf semantische Information erleichtert, aber gleichzeitig die Komplexität des Systems erhöht und die Skalierbarkeit und Offenheit einschränkt.

Aufgrund der Offenheit des INSIGHT-Ansatzes ist es des Weiteren kein Problem, SNMP-fähige Geräte zusammen mit nicht SNMP-fähigen Geräten in ein INSIGHT-System einzubinden. Es muss hierfür nur einmal eine entsprechende SNMP-*Port*- und SNMP-*Parameter*-Klasse (siehe Abschnitt 5.5.2) erstellt werden.

6.5.2 *Java Management Extension* (JMX)

Die *Java Management Extension* (JMX) [Sun00] ist eine neue Erweiterung der *Java 2 Enterprise Edition* (J2EE) zum dynamischen Management entfernter Ressourcen (Software, Geräte oder Services). Insbesondere das Management von autonomen Services innerhalb von verteilten dynamischen Service-Architekturen, die spontan entstehen und wieder verschwinden können (vgl. [SNB00]), ist hierbei eine Herausforderung. Die JMX-Spezifikation definiert hierfür einen Satz von standardisierten Java-Interfaces und -Services, die dieses Ziel unter Berücksichtigung von

- Plattformunabhängigkeit,
- Protokollunabhängigkeit und
- Unabhängigkeit von der Informationsrepräsentation

ermöglichen sollen.

Die zwei grundlegenden Konzepte des JMX-Ansatzes sind *Managed Bean* (MBean) und MBean-Server. Eine MBean ist eine *Java Bean*-Komponente (vgl. [Hoq98]), die zusätzlichen JMX-spezifischen Restriktionen genügt und als Abstraktion eines SNMP-Agenten verstanden werden kann. Sie hat die Aufgabe, bestimmte Managementfunktionalität einer Ressource dem JMX-System auf standardisierte Weise zugänglich zu machen. Das entsprechende Gegenstück ist der MBean-Server, der eine Menge von MBeans verwalten kann und eine einheitliche

Schnittstelle für Managementkonsolen unterschiedlicher Art über sog. *Connectors* und *Protocol Adapters* bereitstellt. Diese oberste Schicht eines JMX-Systems befindet sich allerdings noch in der Standardisierungsphase. Entscheidend ist hierbei, dass die JMX-Architektur unabhängig von den verwendeten Kommunikationsprotokollen ist. Geplant sind insbesondere Implementierungen über HTTP, SNMP und IIOP (JSR-000070[3]).

Die Beschreibung der verfügbaren Managementfunktionalität einer MBean erfolgt über den für *Java Beans* üblichen Weg der *Introspection* (basierend auf *Java Reflection*) sowie den *BeanInfo*-Schnittstellen und/oder zusätzlichen JMX-spezifischen Schnittstellen.

JMX definiert weiterhin eine Reihe von Services (z.B. Monitoring bestimmter Parameter, asynchrone Benachrichtigungen (Events), Timer-Funktionalität ...) die aufgrund der standardisierten Schnittstellen auf allen entsprechenden MBean Komponenten zur Verfügung stehen.

6.5.2.1 Abgrenzung

Das Pendant zu INSIGHT wäre ein JMX-System mit genau einer MBean, die die Rolle von *DeviceInvestigator* übernimmt und *DeMML* über Java-RMI als Daten- bzw. Kommunikationsprotokoll verwendet. Abgesehen von den JMX-Managementservices, die in dieser Form momentan nicht in INSIGHT implementiert sind, liegt der wesentliche technische Unterschied zwischen INSIGHT und JMX in der Repräsentation der verfügbaren Managementinformation. In JMX liegt diese Information in Form von Java-Byte-Code der MBean-Komponenten vor, der die in JMX definierten Schnittstellen implementiert, während INSIGHT eine externe Spezifikation über die *DeMML*-Sprache verwendet. Eine Änderung der JMX-Konfiguration bedeutet daher immer eine erneute Kompilierung der MBeans, es sei denn die Änderung wird direkt über entsprechende Schnittstellen als Managementfunktionalität exportiert, was die Schnittstellen erheblich vergrößern kann.

Die Verwaltung von bestimmten Anlagenkonfigurationen wird in INSIGHT über die Einbindung einer *XML*-Datenbank realisiert. In JMX müssten die einzelnen MBeans in serialisierter Form als *Binary Large Objects* (BLOB) in einer Datenbank abgelegt werden. In dieser Form wäre allerdings eine Recherche in bestehenden Anlagenkonfigurationen sehr aufwendig, da für jede zu untersuchende Konfiguration die entsprechende MBean aus der Datenbank geladen, deserialisiert und analysiert werden müsste. Die Einbettung von *DeviceInvestigator* in eine MBean Komponente stellt dennoch eine interessante Perspektive für die Zukunft dar. Die Einbindung eines MBean-Servers in ein INSIGHT-System wäre wieder aufgrund

[3]http://java.sun.com/aboutJava/communityprocess/jsr/jsr_070_jmxcorba.html

der Offenheit von INSIGHT problemlos möglich.

Auf konzeptioneller Seite unterscheiden sich beide Ansätze insbesondere auch durch ihren Anspruch bzgl. der unterstützten Funktionalität. INSIGHT versteht sich weniger als Managementsystem (zum automatisierten Starten, Überwachen, Stoppen usw. von Ressourcen/Services), sondern vielmehr als Diagnose-Tool zur flexiblen Verwaltung, Visualisierung und Bewertung von entfernten Prozesszuständen.

6.5.3 Industrielle Fernwartungslösungen

Es werden drei Arten von Fernwartungslösungen (Teleservice) unterschieden [Sch98b]:

1. PC-basiertes „Remote-Login"
2. Spezielle Systemlösungen
3. Videokonferenzsysteme

PC-basiertes „Remote-Login" Dieser Ansatz bedient sich (i.d.R. Microsoft-basierter) kommerzieller Softwarelösungen, wie z.B. PC-Anywhere, um einen entfernten Wartungs-Server-PC vor Ort fernzusteuern. PC-Anywhere spiegelt beispielsweise die komplette Videoinformation und die Mausbewegungen des entfernten Bildschirms auf den lokalen Service-PC. Sämtliche Wartungssoftware, die auf dem Wartungs-PC vorhanden ist, ist damit aus der Ferne bedienbar.

Dieser Ansatz setzt einen mächtigen Server-PC mit entsprechender Software auf der Geräteseite voraus. Zusätzlich wird eine große Netzwerkbandbreite beansprucht, da anstatt von Parameterwerten oder Messdaten die graphischen Informationen des kompletten Bildschirms übertragen werden. Eine direkte automatische Integration der Wartungsinformation in eine zentrale Datenbasis ist nicht möglich, da innerhalb des Wartungssystems die Daten nur als graphische Informationen vorliegen. Eine Integration kann erst in einem zweiten Schritt durch die Interpretation eines menschlichen Benutzers/Experten geschehen.

Spezielle Systemlösungen Dieser Ansatz beruht auf speziellen Software-Lösungen für bestimmte Gerätefamilien wie z.B. dem GLASS-System für Komponenten der Firma ABB (siehe Abschnitt 5.2.2) oder dem *TeleService Adapter V5.1* für SPS-S7-Steuerungen der Firma Siemens. I.d.R. werden fest vorgegebene, gerätespezifische Funktionalitäten unterstützt, die über ISDN-Verbindungen und spezielle Applikationsprotokolle auch von der Ferne aus zugänglich sind.

Diese Lösungen bieten einen komfortablen und Bandbreite schonenden Zugriff auf die entsprechenden Geräte bzw. Steuerungskomponenten. Dieser Komfort geht allerdings automatisch mit einer Reduzierung der Generizität sowohl hinsichtlich der unterstützten Services als auch der verwendeten Datenformate einher. Diese Systeme sind nicht erweiterbar und können jeweils nur in spezifischen Kontexten eingesetzt werden.

Videokonferenzsysteme Diese Systeme (z.B. Lotus Sametime) erlauben die Übertragung von Audio- und Videodaten und ermöglichen so eine verbesserte Kommunikation des entfernten Personals mit einem lokalen Experten. Der Experte kann dem Personal Anweisungen geben, z.B. die Kamera auf bestimmte Anlagenteile zu richten, um so einen Eindruck von einer entfernten, fehlerhaften Anlage zu bekommen.

Dieser Ansatz ist zwar für viele Fernwartungsprobleme anwendbar, bietet aber keinen direkten elektronischen Zugriff auf einzelne Zustandsparameter, die wieder von Hand erfasst werden müssen, was sowohl zeitaufwendig als auch fehlerträchtig sein kann.

Kapitel 7

XML Recherche mit *XJM_Eval*

7.1 Motivation

Moderne computerbasierte Systeme produzieren während ihres Betriebs eine wachsende Menge von Informationen über sich selbst (Log-Dateien, Monitoring-Daten), die u.a. zur Qualitätssicherung, Feinabstimmung und Analyse von Fehlerzuständen verwendet werden. Das Problem besteht hierbei oft nicht darin, dass die jeweils erforderlichen Informationen nicht erfasst wurden, sondern vielmehr darin, diese aus der Menge an erfasster Information herauszufiltern.

Innerhalb der INSIGHT-Infrastruktur und in zunehmendem Maß auch in industriellen Monitoring-Systemen werden Monitoring-Informationen in Form von *XML*-Dokumenten abgelegt, wodurch die Handhabbarkeit der Informationen gesteigert wird (vgl. Abschnitt 2.2). Eine auf Ähnlichkeitsmessung basierende Recherchekomponente für *XML*-Dokumente kann daher eine wertvolle Informationsfilterfunktion für diese Systeme darstellen. Eine Ähnlichkeitsmessung kann beispielsweise dazu verwendet werden, Fehlerzustände von entfernten Anlagen mit einer Datenbasis von bereits bekannten und behobenen Fehlerzuständen zu vergleichen und so eine Diagnoseunterstützung zur Fernwartung von entfernten Anlagen zu realisieren (vgl. Use-Case 4 auf Seite 39). Weiterhin könnte diese Ähnlichkeitsmessung in Zukunft beispielsweise dazu verwendet werden, gefährliche Zustände oder spezifischen Verschleiss bestimmter Anlagenkomponenten während der Überwachung der Anlage zu erkennen und zu melden.

Unabhängig von der Verwendung innerhalb von INSIGHT kann eine derartige Recherchekomponente zum Auffinden und Bewerten von Ähnlichkeiten innerhalb von beliebigen *XML*-Dokumenten eingesetzt werden.

7.2 Anforderungen

Folgende Anforderungen an die Recherchekomponente wurden definiert:

$a1$ Flexibilität in Bezug auf Parametrierung, Auswahl und Erstellung einzelner Abstandsfunktionen

$a2$ Einfache Integration bereits bestehender Ähnlichkeitsbewertungssoftware

$a3$ Verwaltung der Abstandsmaße in Datenbanken mit entsprechender Recherchemöglichkeit

$a4$ Verwendung offener, verbreiteter Standards

Die Retrieval Komponente sollte zusätzlich auch den besonderen Charakteristika der Dokumente, die innerhalb eines INSIGHT-Systems generiert werden[1], Rechnung tragen:

- Große strukturelle Ähnlichkeit der Dokumente untereinander
- Die Dokumente enthalten serialisierte Daten, wie z.B. Sensor-Informationen, Ein-/Ausgangsbelegungen von E/A-Modulen, Winkelpositionen von Schrittmotoren, Kamerabilder usw.

Daraus ergaben sich die folgenden zusätzlichen Anforderungen:

$a5$ Unterstützung insbesondere von inhaltsbezogener Ähnlichkeitsbewertung (im Gegensatz zu struktureller Ähnlichkeitsbewertung)

$a6$ Deserialisierung von *XML*-Fragmenten in getypte Daten

7.3 Stand der Forschung

Die Entwicklung des *XJM_Eval*-Systems [BK01] wurde beeinflusst von den Ergebnissen der Feature-basierten Ähnlichkeitsbewertung, der Ähnlichkeitsbewertung für Baumstrukturen und der Entwicklung unterschiedlicher *XML*-Querysprachen.

[1]Derartige Dokumente treten z.B. auch unabhängig von INSIGHT immer dann auf, wenn Zustände *XML*-basiert überwacht werden (siehe z.B. [SHKK00]).

Metrik	Abstandsfunktion
City-Block-Metrik (L_1-Metrik)	$d_{L_1}(x,y) := \sum_{i=1}^{n} \mid x_i - y_i \mid$
Euklidische Metrik (L_2-Metrik)	$d_{L_2}(x,y) := \sqrt{\sum_{i=1}^{n} \mid x_i - y_i \mid^2)}$
L_p-Metrik	$d_{L_p}(x,y) := (\sum_{i=1}^{n} \mid x_i - y_i \mid^p)^{\frac{1}{p}}$
χ^2-Metrik	$d_{\chi^2}(x,y) := \sum_{i=1}^{n} \frac{(x_i-y_i)^2}{x_i+y_i}$

Tabelle 7.1: Abstandsfunktionen einiger Metriken für zwei Vektoren $x = (x_1, ..., x_n)$ und $y = (y_1, ..., y_n)$ (vgl. [Ues01])

7.3.1 Feature-basierte Ähnlichkeitsbewertung

Dieser Ansatz, der vor allem auch mit dem zunehmenden Angebot an Internet-basierten Multimedia-Datenbanken an Bedeutung gewinnt, beschäftigt sich allgemein mit der Ähnlichkeitsbewertung von hochdimensionalen Datenstrukturen z.B. im Kontext von Bilderkennung [FBF+94], Vergleich von Volumendaten [ENR97] oder Videodatenbanken [JK00]. Die entsprechenden Algorithmen basieren auf der Abbildung bestimmter Aspekte (Features) der komplexen Datenstrukturen auf Elemente eines entsprechend hochdimensionalen Vektorraums (sog. Feature-Vektoren) und der Auswertung von Abstandsfunktionen (z.B. die Euklidische Abstandsfunktion (vgl. Tabelle 7.1)) auf den einzelnen Vektorenpaaren. Zur Effizienzsteigerung können Indexstrukturen und Parallelisierungsansätze integriert werden (z.B. [BBB+97]). Insbesondere wenn die verwendeten Abstandsfunktionen metrische Funktionen sind, lassen sich besonders effiziente Indexstrukturen erstellen, die eine schnelle Ähnlichkeitsbewertung erlauben (z.B. [Yia93, Cla97]).

Zur Parametrierung der Ähnlichkeitsmaße stehen i.d.R. Möglichkeiten zur Auswahl der zu berücksichtigenden Features und deren Gewichtung in der Vektorraum-Abstandsfunktion zur Verfügung. Ein abstraktes theoretisches Framework zur Spezifikation von derartigen anwendungsspezifischen Ähnlichkeitsmaßen wird in [JMM95] vorgeschlagen. Das Framework besteht aus drei Komponenten: Einer Pattern-Sprache zur Repräsentation der Datenobjekte (z.B. Reguläre Ausdrücke für textuelle Daten), einer Transformationssprache zur Bewertung der Ähnlichkeit von je zwei Objekten (z.B. über das Aufsummieren der Kosten einzelner Transformationen, die zur Transformation des ersten Objekts in das zweite nötig sind) und einer Querysprache.

7.3.2 Ähnlichkeitsbewertung für Baumstrukturen

Da die logische Struktur eines jeden *XML*-Dokuments die eines geordneten Wurzelbaumes ist, können Forschungsergebnisse aus dem Bereich des *Tree Matching* auch auf *XML*-Datenstrukturen übertragen werden. Ein gut untersuchtes Maß zur Bewertung der strukturellen Ähnlichkeit von Bäumen ist hierbei z.B. die sog. *Edit Distance* [Tai79, ZWS95]. Zur Berechnung der *Edit Distance* zwischen zwei Bäumen werden zunächst drei Operationen definiert: *Relabeling*, *Insert* und *Delete*, die eine schrittweise Transformation des einen Baums in den anderen erlauben. Je weniger Operationen nötig sind, um den einen Baum in den anderen zu transformieren, desto ähnlicher sind sich die Bäume. Zusätzlich können die einzelnen Operationen noch mit unterschiedlichen Kosten belegt werden.

Es wurden diverse Verfahren entwickelt, um die Berechnung der *Edit Distance* für zwei Bäume möglichst effizient durchzuführen (z.B. [ZS89]). Oommen et al. stellen in [OZL96] ein abstraktes Ähnlichkeitsmaß für die Struktur zweier Bäume vor, das zusätzlich zur *Edit Distance* auf weitere Ähnlichkeitsmaße (z.B. Größe des größten gemeinsamen Teilbaums) spezialisiert werden kann.

Das *Approximate-Tree-By-Example*-System [WZJS94] integriert in die *Edit Distance*-basierte Bewertung der Ähnlichkeit von Bäumen zusätzlich eine Ähnlichkeitsbewertung der jeweiligen Knoteninhalte durch Berechnung der *Edit Distance* für flache Zeichenketten. Es wird nicht zwingend versucht, eine exakte Transformation des einen Baumes in den anderen zu finden, sondern es werden bestimmte Unterschiede toleriert. Zur Spezifikation entsprechender Anfragen wurde eine spezielle Querysprache entwickelt, die innerhalb des Systems auch graphisch unterstützt wird. Ein weiteres Tool [CPRW97], das auf diesem Ansatz basiert, unterstützt direkt die Verwendung von *SGML*-Dokumenten und visualisiert ähnliche und nicht-ähnliche Teile von *SGML*-Dokumenten durch entsprechende Texthervorhebungen in einer graphischen Benutzerschnittstelle. Das Tool *XML Diff*[2] von IBM bietet ähnliche Funktionalität für *XML*-Dokumente.

7.3.3 *XML*-Querysprachen

Eine ganze Reihe unterschiedlicher *XML*-Querysprachen (z.B. *Lorel* [AQM+97], *XQL* [RLS98], *XML-QL* [DFF+98], *XML-GL* [CCD+99], *XPath/XML Query* [MF00, CD99]) wurden bereits entwickelt oder befinden sich gerade in der Standardisierungsphase. Ein Vergleich unterschiedlicher *XML*-Querysprachen findet sich beispielsweise in [BC00].

Die genannten Querysprachen erlauben die Spezifikation von bestimmten Pfaden

[2] Kostenfrei erhältlich unter http://www.alphaworks.ibm.com/tech/xmldiffmerge

in die hierarchische Struktur eines oder teilweise auch mehrerer *XML*-Dokumente, um bestimmte *XML*-Fragmente zu selektieren. Zusätzlich können diese Pfade mit Prädikaten assoziiert werden, die bestimmte Pfade verwerfen und andere akzeptieren.

Die Query aus Abbildung 6.7 auf Seite 102, die sowohl eine gültige *XQL*-Query als auch einen gültigen *XPath*-Ausdruck darstellt, lautet beispielsweise:

```
DeMML[Snapshot/@LocationId="0042"]
```

Diese Anfrage selektiert zunächst alle *DeMML*-Elemente aus einem gegebenen *XML*-Datenbestand. Das in eckigen Klammern eingeschlossene Prädikat selektiert daraufhin bestimmte Elemente aus der Ergebnismenge. Das Prädikat fordert in diesem Fall, dass die *DeMML*-Elemente jeweils ein Subelement *Snapshot* beinhalten, welches wiederum ein Attribut *LocationId* mit Wert *0042* trägt.

Schlieder et al. [SN00] schlagen die Sprache *approXQL* als Abwandlung von *XQL* zur Spezifikation von unsicheren Anfragen an inhomogene *XML*-Datenbestände vor. Zur Formulierung von *approXQL*-Queries sind keine Kenntnisse über die exakte logische Struktur der *XML*-Datenbestände nötig. Eine Anfrage wird auf einen entsprechenden Template-Baum abgebildet, der dann entsprechend dem *Tree Inclusion*-Ansatz (siehe z.B. [KM93]) in die Vergleichsdokumente eingebettet wird. Je nach Erfolg und Aufwand der Einbettung kann der Grad der Übereinstimmung mit der Query gemessen werden.

Navarro und Baeza-Yates untersuchen in [NBY97] Ansätze zur Formulierung und Auswertung von Anfragen an Textdatenbanken unter Berücksichtigung von Struktur und Inhalt. Sie schlagen hierfür ein erweiterbares Framework vor, das unabhängig von der Manifestation der Strukturierung (z.B. *SGML* oder LaTeX) der Textdaten anwendbar ist. Weiterhin ist eine zukünftige Integration von multimedialen Datentypen bereits im Framework berücksichtigt.

7.4 Der *XJML*-Ansatz

Das *XJM_Eval*-System erweitert die *XML*-Querysprachen durch die Integration der Programmiersprache Java und ermöglicht dadurch eine flexible, inhaltsbezogene Ähnlichkeitsbewertung für *XML*-Dokumente. Ein bestimmtes Ähnlichkeitsmaß des *XJM_Eval*-Systems wird hierbei selbst durch ein *XJML* (*XML to Java Mapping Language*) [BK01] Dokument in *XML* spezifiziert.

XJM_Eval unterstützt insbesondere eine *Nearest Neighbor*-Suche (NN-Suche) (siehe z.B. [RKV95]) für einzelne *XML*-Anfragedokumente innerhalb eines *XML*-Datenbestandes. Ein Datenbestand kann hierbei durch ein Verzeichnis des

Dateisystems oder eine Query an ein *XML*-Datenbanksystem definiert sein. Die einzelnen Abstandsfunktionen sind jeweils als Java-Methoden implementiert und werden dynamisch in *XJM*-Ähnlichkeitsmaße eingebunden. Sie arbeiten jeweils nur auf bestimmten, durch *XQL*- oder *XPath*-Statements selektierten Fragmenten der zu vergleichenden *XML*-Dokumente. Die einzelnen *XML*-Fragmente können hierbei nicht nur als rein textuelle Informationen interpretiert und verarbeitet werden, sondern auch als textuelle Repräsentationen bestimmter Datentypen (z.B. Integer, Double, CANopenNodeState ...). Als Datentypen können prinzipiell alle von *java.lang.Object* abgeleiteten Java-Klassen verwendet werden, solange sie eine textuelle Repräsentation des entsprechenden Wertes unterstützen (vgl. Tabelle 7.2). Neue Abstandsfunktionen können völlig unabhängig von *XJM_Eval* entwickelt und über entsprechende Referenzen und *Java Reflection* auf einfache Weise in das System integriert werden.

XJM_Eval verbindet damit auf flexible Weise die Stärken von *XML* und der *XML*-Querysprachen mit der Mächtigkeit der Java-Programmiersprache.

7.5 Abgrenzung

Die z.T. in Abschnitt 7.3 beschriebenen bestehenden Systeme erwiesen sich für die definierten Anforderungen als zu unflexibel. Dies liegt hauptsächlich daran, dass die Anforderung $a6$, also die Deserialisierung von *XML*-Fragmenten in getypte Daten, wenn überhaupt, jeweils nur unzureichend unterstützt wird. Die Verwaltung von Parameterwerten als serialisierte Zeichenketten in Form von *XML*-Fragmenten mit extern spezifiziertem Datentyp ist aber ein zentrales Konzept von INSIGHT und sollte deshalb auch von der Recherchekomponente entsprechend unterstützt werden.

Die Feature-basierten Ansätze sind i.d.R. auf bestimmte Anwendungsgebiete und Datentypen (z.B. typische Multimedia-Daten) beschränkt. Weiterhin werden bestimmten Features auch bestimmte Abstandsfunktionen statisch zugeordnet (z.B. in [FBF$^+$94]) was unvereinbar ist mit $a1$.

Das theoretische Framework aus [JMM95] ist aufgrund seiner Abstraktheit prinzipiell dazu in der Lage, die meisten der gestellten Anforderungen zu modellieren, erschien aber als Grundlage für dieses System aus zwei Gründen als eher unnatürlich: die zwingende Rückführung von Ähnlichkeitsfunktionen auf gewichtete Sequenzen von Transformationen und die im *XML*-Umfeld eher exotische Syntax der Querysprachen. Immerhin legt $a4$ eine Rückführung auf allgemeine, verbreitete Standards nahe.

Ein Großteil der Ähnlichkeitsmessung für Baumstrukturen befasst sich nur mit der Bewertung der strukturellen Ähnlichkeit von Bäumen und berücksichtigt für den

Vergleich der einzelnen Labels (also den Dateninhalten der einzelnen Knoten) nur das binäre Prädikat gleich oder ungleich bzw. die Ähnlichkeit der Zeichenketten an sich. Dieser Ansatz wird den Anforderungen $a5$ und $a6$ nicht gerecht, da innerhalb des INSIGHT-Systems vor allem ein inhaltlicher Vergleich von Dokumenten wichtig ist.

Die Semantik der erwähnten *XML*-Querysprachen (vgl. Abschnitt 7.3.3) ist darauf ausgelegt, bestimmte Fragmente aus *XML*-Dokumenten zu selektieren und als Ergebnis zurückzuliefern. Anspruchsvollere Recherchefunktionalitäten, wie z.B. ähnlichkeitsbasiertes Suchen werden nicht unterstützt. Die *approXML* Querysprache berücksichtigt nur strukturelle und keine inhaltsbezogene Ähnlichkeit ($a5$).

Der allgemeine Ansatz des *Approximate-Tree-By-Example*-Systems [WZJS94] ermöglicht auf der einen Seite wertvolle Einsichten in das generelle Prinzip des struktur- und inhaltsbezogenen Auffindens von Information aus strukturierten Dokumenten, erscheint aber auf der anderen Seite als eher unhandlich in dem sich stark durchsetzenden spezielleren *XML*-Umfeld. Zudem ist das System stark zeichenkettenorientiert ($a6$) und benützt statt offener *XML*-Standards ($a4$), wie z.B. *XPath*, eine spezielle API [ARS92] zur Selektion bestimmter Textfragmente. Die Implementation erscheint zudem als monolithisch, was eine ohnehin aufwändige Erweiterung des Systems für neue Datentypen (Erstellung entsprechender Querysprachen und Indizierungskomponenten) zusätzlich erschwert ($a1$, $a6$).

7.6 Begriffsbestimmungen und Definitionen

7.6.1 Überblick

Abbildung 7.1 gibt einen Überblick über das *XJM_Eval*-System, das aus den folgenden Komponenten besteht:

1. *XJM*-Kernsystem
   (a) *XML to Java Mapping* (*XJM*) Konzept
   (b) *XML to Java Mapping Language* (*XJML*)
   (c) *XJM_Eval*-Evaluationswerkzeug
2. Menge von Abstandsfunktionen implementiert durch entsprechende Java-Klassen und -Methoden
3. *XML*-Datenbestand (Datenbank oder Dateisystem)

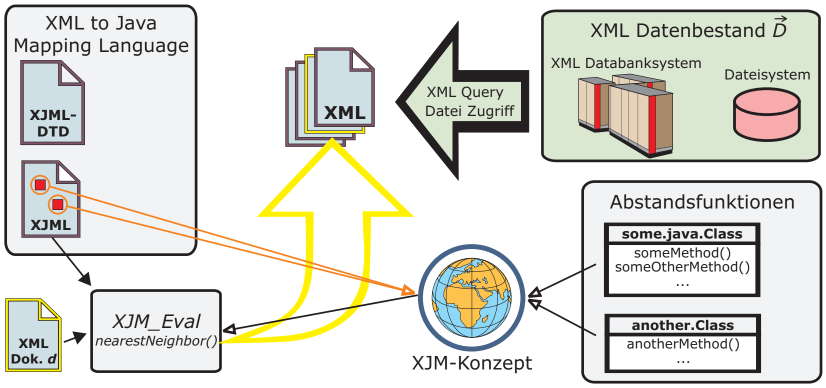

Abbildung 7.1: Überblick über das *XJM_Eval*-System

Das *XJM*-Kernsystem bildet die Infrastruktur zur Definition und Auswertung von *XJM*-basierten Ähnlichkeitsmaßen. Es ist realisiert als *XML*-basiertes Software-Framework, das durch dynamische Integration von spezieller Java-Logik über *Java Reflection* auf spezifische Rechercheanforderungen maßgeschneidert werden kann.

XJM_Eval ist die aktive Komponente des Systems. Zur Berechnung einer NN-Suche werden *XJM_Eval*, wie in Abbildung 7.1 dargestellt, ein Anfragedokument und ein *XJML*-Dokument übergeben. Das *XJML*-Dokument enthält die Spezifikation des Suchraums, der beispielsweise durch die Query an ein Datenbanksystem definiert sein kann. Die eigentliche Berechnung der Ähnlichkeit zweier Dokumente besteht aus der Berechnung und Zusammenführung einer Folge von sog. Ähnlichkeitsaspekten (siehe Abschnitt 7.6.3), die ebenfalls innerhalb des *XJML*-Dokuments durch Verweise auf externen Java-Code angegeben sind. Dieser Java-Code wird dann zur Laufzeit dynamisch über Java-Reflection ausgeführt.

7.6.2 Das *XJM*-Konzept

Das *XJM*-Konzept ermöglicht es, innerhalb eines *XML*-Dokuments Java-Klassen und Methoden auf dem Internet zu referenzieren und diese Methoden durch *XML*-Fragmente (s.u.) bzw. DOM-Knoten zu parametrieren. Die Identifikation der Java-Klassen erfolgt über die *XML*-basierte Angabe der Basis-URL des Java-Byte-Codes, den Java-Klassennamen und ggf. den Methodennamen. Dieser Ansatz kann als Analogon zu dem COM/DCOM-Konzept der UUID zur global eindeutigen

Identifizierung von Software-Komponenten betrachtet werden, mit dem Vorteil, dass die Identifikation selbst bereits Protokoll und Lokation der Klasse innerhalb des Internet enthält. Diese Art der Identifikation gewährleistet, dass Programme, die auf dem *XJM*-Konzept aufbauen, die identifizierte Java-Logik jederzeit dynamisch laden und instanziieren können.

Instanziiert und aufgerufen werden die entsprechenden Klassen bzw. Methoden letztendlich jeweils über *Java Reflection*. Während das *DeviceInvestigator*-Tool aus Kapitel 5 dieses Konzept einsetzt, um in *XML*-Konfigurationsdokumenten Hardwarezugriffsklassen zu spezifizieren, wird dieses Konzept innerhalb des *XJM_Eval*-Systems dazu verwendet, in Java kodierte Abstandsfunktionen bestimmten *XML*-Fragmenten zuzuordnen, um die Ähnlichkeit der entsprechenden *XML*-Dokumente zu bewerten.

7.6.3 Definitionen

Bevor die Arbeitsweise des *XJM_Eval*-Systems im Detail vorgestellt wird, werden zunächst die im weiteren Verlauf wichtigen Begriffe *XJM*-Ähnlichkeitsaspekt und *XJM*-Ähnlichkeitsmaß etwas formaler eingeführt. Ein *XJM*-Ähnlichkeitsaspekt ist eine durch ein *XJML*-Dokument spezifizierte Funktion, die den Grad der Ähnlichkeit einer bestimmten Eigenschaft (Feature) zweier *XML*-Dokumente bewertet. Ein *XJM*-Ähnlichkeitsmaß berechnet aus einer Folge von *XJM*-Ähnlichkeitsaspekten ein Endergebnis, das die Gesamtähnlichkeit zweier *XML*-Dokumente widerspiegelt.

Definition 7.1 *XML*-Fragment:
Ein *XML*-Fragment ist eine beliebige Teilzeichenkette eines wohlgeformten *XML*-Dokuments (z.B. ein Attributwert).

Definition 7.2 *XJM*-Referenz:
Eine *XJM*-Referenz identifiziert eine Java-Klasse bzw. eine Methode einer bestimmten Klasse auf dem Internet durch Angabe der folgenden drei Informationen:

1. Voll qualifizierter Java-Klassenname
2. URL des Java-Byte-Codes (optional)
3. Methodenname (optional)

Falls keine URL zur Lokalisierung des Java-Byte-Codes angegeben wird, so wird der lokale Java-Klassenpfad angenommen. Falls kein Methodenname angegeben wird, wird der Konstruktor dieser Klasse referenziert.

Definition 7.3 ***XJM*-Ähnlichkeitsaspekt**:
Sei

- $\mathbb{O}$ die Menge der Java-Instanzen vom Typ *java.lang.Object*
- $\mathbb{J}$ die Menge der *XJM*-Referenzen
- $\mathbb{D}$ die Menge der wohlgeformten *XML*-Dokumente und $\mathbb{F}$ die Menge aller *XML*-Fragmente
- $d, d' \in \mathbb{D}$ *XML*-Dokumente, $n \in \mathbb{N}$
- $\vec{S} = (s_1, ..., s_n)$ ein Vektor von *XML*-Selektionen, mit $s : \mathbb{D} \to \mathbb{F}$
- $\vec{J} = (j_1, ..., j_n)$ ein Vektor von *XJM*-Referenzen
- $\vec{M} = (m_1, ..., m_n)$ ein Vektor von Abbildungen von *XML*-Fragmenten auf Java-Klassen, mit $m : \mathbb{F} \times \mathbb{J} \to \mathbb{O}$
- $a_{\vec{S}\vec{J}\vec{M}}$ eine Funktion mit $a_{\vec{S}\vec{J}\vec{M}} : \mathbb{D} \times \mathbb{D} \to \mathbb{O}$,
$a_{\vec{S}\vec{J}\vec{M}}(m_1(s_1(d), j_1), m_1(s_1(d'), j_1), ..., m_n(s_n(d), j_n), m_n(s_n(d'), j_n)) = o$

dann heißt $a_{\vec{S}\vec{J}\vec{M}}$ Ähnlichkeitsaspektfunktion und das Tupel
$A = (\vec{S}, \vec{J}, \vec{M}, a_{\vec{S}\vec{J}\vec{M}})$ *XJM*-Ähnlichkeitsaspekt.

Ein Ähnlichkeitsaspekt definiert also eine Funktion, die, unter Berücksichtigung einer Reihe von *XML*-Selektionen (z.B. *XPath*-Ausdrücke) und *XJM*-Referenzen, zwei *XML*-Dokumenten ein Java-Objekt zuordnet. Mit der verkürzten Schreibweise $A(d, d')$ sei im Folgenden immer die Auswertung von

$$a_{\vec{S}\vec{J}\vec{M}}(m_1(s_1(d), j_1), m_1(s_1(d'), j_1), ..., m_n(s_n(d), j_n), m_n(s_n(d'), j_n))$$

im obigen Sinne gemeint.

Ein Ähnlichkeitsaspekt kann beispielsweise die prozentuale Abweichung des Wertes eines bestimmten *DeMML*-Parameters in Bezug auf eine Maximalabweichung sein. Ausgewertet würde dieser Aspekt durch die Selektion der entsprechenden *XML*-Fragmente in d und d', die Verpackung dieser Fragmente in die referenzierten Java-Klassen und die Berechnung des Ähnlichkeitsaspektes a dieser Java-Objekte durch eine entsprechende Java-Methode.

Definition 7.4 ***XJM*-Ähnlichkeitsmaß**:
Sei

- $d, d' \in \mathbb{D}$ *XML*-Dokumente
- $\vec{A} = (A_1, ...A_u)$ ein Vektor von *XJM*-Ähnlichkeitsaspekten
- $\vec{W} = (w_1, ...w_u)$ ein Vektor von Gewichten
- e eine Evaluationsfunktion mit
  $e(A_1(d, d'), ..., A_u(d, d'), w_1, ..., w_u) \rightarrow \mathbb{O}|_{java.lang.Double}$

dann heißt das Tupel $E = (\vec{A}, \vec{W}, e)$ *XJM*-Ähnlichkeitsmaß.

Ein *XJM*-Ähnlichkeitsmaß definiert einen reellen Abstandswert für je zwei *XML*-Dokumente durch Zusammenführen mehrerer Ähnlichkeitsaspekte unter Berücksichtigung der durch $\vec{W}$ gegebenen Gewichtung der einzelnen Ähnlichkeitsaspekte. Mit der verkürzten Schreibweise $E(d, d')$ sei im Folgenden immer die Auswertung von

$$e(A_1(d, d'), ..., A_u(d, d'), w_1, ..., w_u)$$

im obigen Sinne gemeint. Eine mögliche Implementierung dieser Funktion wäre beispielsweise die Berechnung des arithmetischen Mittels für Integer-Zahlen.

Definition 7.5 *XJM* Nearest Neighbor Problem:
Sei

- $\vec{D} = (d_1, ...d_t), d_i \in \mathbb{D}$ ein *XML*-Datenbestand
- $d \in \mathbb{D}$ ein *XML*-Dokument
- E ein *XJM*-Ähnlichkeitsmaß

dann definiert das Tupel $P = (\vec{D}, d, E)$ ein *XJM* Nearest Neighbor Problem (NN-Problem).
Für die Lösung dieses Problems *XJM_Eval*$(P) = r \in \mathbb{N}$ gilt:
$E(d, d_r) \leq E(d, d_i) \forall 1 \leq i \leq t$.

Die Lösung eines *XJM*-NN-Problems identifiziert demnach das Dokument aus $\vec{D}$, das bezüglich eines *XJM*-Ähnlichkeitsmaßes den geringsten Abstand zu d hat.

7.7 Die *XML to Java Mapping Language* (*XJML*)

Die *XML to Java Mapping Language* (*XJML*) ist eine durch die *XJML*-DTD (siehe Anhang A.5) definierte *XML*-Applikation zur Spezifikation von

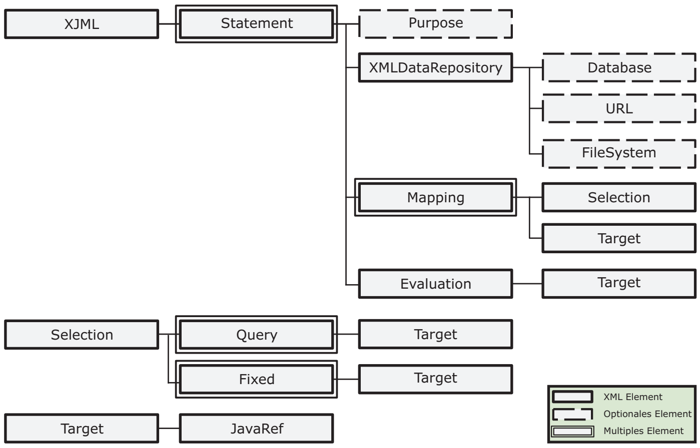

Abbildung 7.2: Logische Struktur der *XJML*-DTD

XJM-Referenzen, Ähnlichkeitsaspekten, Ähnlichkeitsmaßen und letztendlich NN-Problemen. Abbildung 7.2 gibt einen Überblick über die logische Struktur eines *XJML*-Dokuments, und Listing 7.1 und 7.2 zeigen ein einfaches *XJML*-Beispieldokument für den partiellen Zustandsvergleich eines digitalen E/A-Moduls der CANopen-Demonstrationszelle aus Fallstudie 1 (vgl. Abschnitt 5.6).

Das *XJML*-Element ist das obligatorische Wurzelelement zur Identifikation von *XJML*-Dokumenten. Das *Statement*-Element dient zur Definition eines *XJML*-Statements, z.B. eines NN-Problems[3]. Dieses Element ist in die DTD integriert worden, um zukünftige weitere Problemstellungen in der *XJML*-Sprache repräsentieren zu können.

Innerhalb des optionalen *Purpose*-Subelements des *Statement*-Elements kann eine natürlichsprachliche Beschreibung des spezifizierten *XJML*-Statements abgelegt werden. Diese Information kann insbesondere dazu verwendet werden, um basierend auf einer Volltextsuche nach speziellen Ähnlichkeitsmaßen innerhalb eines *XJML*-Datenbestandes (z.B. einer *Tamino*-Datenbank) zu recherchieren. Denkbar wären hier z.B. Ähnlichkeitsmaße für Fehlerzustände bestimmter Anlagentypen

[3]Das *XJM*-System und *XJM_Eval* sind nicht auf die Auswertung derartiger Probleme beschränkt, sondern können prinzipiell für beliebige weitere Auswertungen auf *XML*-Dokumenten verwendet werden.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE XJML SYSTEM
'http://www-sr.informatik.uni-tuebingen.de/Insight/XJML.dtd' [
  <!ENTITY suvretta "http://suvretta.informatik.uni-tuebingen.de/classes">
  <!ENTITY base "de.wsi.xjml"> ]>
<XJML>
  <Statement Problem="Nearest␣Neighbor">
    <Purpose>CANopen Robot Cell Error State Detection ... </Purpose>
    <XMLDataRepository RootElement="Snapshot">
      <Database Query="http://suvretta.informatik.uni-tuebingen.de/tamino\
/DeMML/data/DeMML?_xql=DeMML/Snapshot"/>
    </XMLDataRepository>
    <Mapping Weight="1">
      <Selection>
        <Query XPath="ScannedDevice[@DeviceId='0F']/LogEntry\
[@ParameterId='6000,1']/@Value">
          <Target><JavaRef Codebase="&suvretta;" Class="&base;.IntegerHex"/>
          </Target>
        </Query>
        <Fixed Value="8">
          <Target><JavaRef Class="java.lang.Integer"/></Target>
        </Fixed>
      </Selection>
      <Target><JavaRef Codebase="&suvretta;" Class="&base;.eval.Methods"
        Method="hammingDistance"/></Target>
    </Mapping>
    ...
```

Listing 7.1: Ein *XJML*-Dokument für CANopen-Prozessdaten (Teil 1: Berechnung des Hamming-Abstands der Eingangsbelegung eines digitalen E/A-Moduls)

aufgrund von:

- Verschleiß
- Spezifischer Fremdeinwirkung
- Überlastung

Das *XMLDataRepository*-Element spezifiziert den Datenbestand $\vec{D}$ für ein *XJM-NN*-Problem (vgl. Definition 7.5). Es erlaubt die Verwendung der folgenden drei *XML*-Datenquellen (vgl. Abbildung 7.2):

1. Ergebnismenge einer Anfrage an eine (entfernte) *XML*-Datenbank
2. Ein *XML*-Dokument auf dem Internet durch Angabe einer entsprechenden URL
3. Das Dateisystem durch Angabe eines Verzeichnisses und eines Dateinamen-Filters

```
    ...
    <Mapping Weight="1">
      <Selection>
        <Query XPath="ScannedDevice[@DeviceId='0F']/LogEntry\
  [@ParameterId='1002,0']/@Value">
          <Target>
            <JavaRef Codebase="&suvretta;"
              Class="de.uni_tuebingen.can.CanProtocol.CANopenNodeState"/>
          </Target>
        </Query>
      </Selection>
      <Target>
        <JavaRef Codebase="&suvretta;" Class="&base;.eval.Methods"
          Method="CANopenNodeStateDistance"/>
      </Target>
    </Mapping>
    <Evaluation>
      <Target><JavaRef Codebase="&suvretta;" Class="&base;.eval.Methods"
        Method="doubleAverage"/></Target>
    </Evaluation>
  </Statement>
</XJML>
```

Listing 7.2: Ein *XJML*-Dokument für CANopen-Prozessdaten (Teil 2: Vergleich des CANopen-Knotenzustandes und Referenzierung der Evaluationsfunktion)

Diese Datenquellen werden jeweils durch die Subelemente *Database*, *URL* bzw. *FileSystem* repräsentiert. Wenn das optionale Attribut *RootElement* gesetzt ist, werden die jeweils referenzierten *XML*-Dokumente zusätzlich gemäß dem angegebenen Tag-Namen unterteilt.

Das *XJML*-Dokument aus Listing 7.1 definiert beispielsweise in Zeile 9-12 zunächst das Ergebnis der *X-Query DeMML/Snapshot* (vgl. Abschnitt 6.2.3.2) an die *Tamino*-Datenbank mit Namen *DeMML* auf dem Rechner *suvretta.-informatik.uni-tuebingen.de* als Suchraum. Diese Ergebnismenge wird durch das *RootElement*-Attribut zusätzlich auf die Menge aller *Snapshot*-Element der Ergebnismenge spezialisiert.

Das *Mapping*-Element repräsentiert einen *XJM*-Ähnlichkeitsaspekt (vgl. Definition 7.3). Wie in Abbildung 7.2 dargestellt, besteht ein *Mapping*-Element aus genau einem *Selection*-Element und einem *Target*-Element. Das optionale Attribut *Weight* des *Mapping*-Elements spezifiziert das Gewicht w (vgl. Definition 7.4), das das Ergebnis dieses Ähnlichkeitsaspekts in der Evaluationsfunktion haben soll.

Das *Selection*-Element definiert die Vektoren $\vec{S}$, $\vec{J}$ und $\vec{M}$ aus Definition 7.3. Die Selektionen aus $\vec{S}$ sind entweder vom Typ *Query* oder *Fixed*. Das Element *Query* repräsentiert entweder eine *XQL* Anfrage durch das Attribut *Xql* oder einen *XPath*-Ausdruck durch das Attribut *XPath*. Das *Fixed*-Element kann als direkte Selektion auf den Wert des zugehörigen *Value*-Attributs aufgefasst werden. Das

Fixed-Element aus Zeile 20 in Listing 7.1 selektiert also z.B. direkt das *XML*-Fragment *8*. Es dient hauptsächlich zur Parameterübergabe unveränderlicher Werte an die Funktion, die den Wert dieses Ähnlichkeitsaspektes berechnet.

Jedem *Query* bzw. *Fixed*-Element ist genau ein *Target*-Element zugeordnet, welches die entsprechende *XJM*-Referenz (vgl. Definition 7.2) durch ein *JavaRef*-Element spezifiziert. Die zusätzliche Indirektionsstufe über das *Target*-Element wurde im Hinblick auf zukünftige Erweiterungen für andere Programmiersprachen eingeführt. Das *JavaRef*-Element enthält die Informationen bzgl. des Klassennamens, der Basis-URL des Java-Byte-Codes und des Methodennamens in entsprechenden Attributen. Das *JavaRef*-Element aus Zeile 24-25 in Listing 7.1 referenziert beispielsweise die Methode *hammingDistance()* der Klasse *de.wsi.xjml.-eval.Methods*, deren Java-Byte-Code von der URL *http://suvretta.informatik.uni-tuebingen.de/classes* geladen werden kann.

Die jeweilige Zuordnung der *Query*- bzw. *Fixed*-Elemente zu ihrem *Target* Subelement definiert letztendlich die Vektoren $\vec{J}$ und $\vec{M}$ aus Definition 7.3 und ordnet jedem selektierten *XML*-Fragment eine Java-Klasse zu. Das *Target* Subelement des *Mapping*-Elements referenziert wieder über ein entsprechendes *JavaRef*-Element die Ähnlichkeitsaspektfunktion a, die die Auswertung dieses *XJM*-Ähnlichkeitsaspektes mit den durch das *Selection*-Element definierten Java-Objekten als Parametern implementiert.

Die Folge von *Mapping*-Elementen zusammen mit dem abschließenden *Evaluation*-Element definiert ein *XJM*-Ähnlichkeitsmaß (vgl. Definition 7.4). Die Evaluationsfunktion e wird wieder über ein *Target*-Element referenziert. Der Vektor $\vec{w}$ von Gewichten für die einzelnen Ähnlichkeitsaspekte wird über das optionale *Weight*-Attribut der *Mapping*-Elemente festgelegt. Eine Gewichtung muss innerhalb des *XJM*-Systems nicht explizit angegeben werden, sondern kann alternativ auch fest in die Evaluationsmethode implementiert werden. Eine externe Verwaltung der Gewichte innerhalb des *XJML*-Dokuments erhöht jedoch die Flexibilität des definierten Ähnlichkeitsmaßes.

Die in Zeile 18-19 aus Listing 7.2 referenzierte Evaluationsfunktion *doubleAverage()* berechnet beispielsweise das arithmetische Mittel der durch die beiden definierten Ähnlichkeitsaspekte definierten Ähnlichkeitswerte durch die folgende Formel:

$$E(d,d') = \frac{\sum_{i=1}^{u} java.lang.Double(A_i(d,d')) \cdot w_i}{\sum_{i=1}^{u} w_i}$$

Ein *XML*-Dokument zusammen mit einem *XJML*-Dokument definieren demnach ein spezielles NN-Problem (vgl. Definition 7.5). Das in Listing 7.1 und 7.2 abgedruckte *XJML*-Dokument spezifiziert beispielsweise ein Ähnlichkeitsmaß, das die unterschiedliche Belegung der einzelnen Eingänge des CANopen-E/A-Moduls mit

DeviceId `0F` und den CANopen-Knotenzustand dieses Knotens zu gleichen Teilen berücksichtigt. Es ist auf eine Auswertung von *DeMML-Snapshot*-Dokumenten zugeschnitten, wie sie während des Monitorings der in Abschnitt 5.6 beschriebenen CANopen-Demonstrationszelle anfallen. Für eine aussagekräftigere Ähnlichkeitsbewertung dieser Dokumente können weitere der erfassten Parameter (z.B. die Winkelpositionen der Roboterachsen) in das Ähnlichkeitsmaß mit aufgenommen werden (vgl. Listing A.8 auf Seite 242).

Die Semantik eines *XJML*-Dokuments ist damit verteilt auf das *XJML*-Dokument selbst und die global eindeutig referenzierten Java-Klassen. Die Identifikation der Java-Klassen über das *XJM*-Konzept gewährleistet hierbei sowohl die Vermeidung von Mehrdeutigkeiten als auch die Möglichkeit, die nötigen Java-Klassen zur Laufzeit dynamisch via Internet zu laden und zu instanziieren (siehe nächster Abschnitt). Ein *XJML*-Dokument ist daher unabhängig von den lokal vorhandenen Java-Klassen eine sinnvolle Spezifikation eines Ähnlichkeitsmaßes.

7.8 Arbeitsweise des *XJM_Eval*-Tools

Das *XJM_Eval* Tool implementiert u.a. die Lösung von *XJM*-NN-Problemen. Wie bereits in Abbildung 7.2 dargestellt und im vorigen Abschnitt beschrieben, benötigt das Tool hierfür eine Referenz auf das Anfragedokument d und ein *XJML*-Dokument, das den *XML*-Datenbestand und das *XJM*-Ähnlichkeitsmaß definiert.

Die Auswertung eines *XJM*-NN-Problems zerfällt in die folgenden drei Phasen:

1. Initialisierung
2. Auswertung:
   (a) Auswertung der *XJM*-Ähnlichkeitsaspekte
   (b) Auswertung des *XJM*-Ähnlichkeitsmaßes
3. Ergebnisausgabe

7.8.1 Initialisierung

In der Initialisierungsphase wird zunächst das *XJML*-Dokument geparst und der entsprechende DOM-Baum aufgebaut. Das Tool instanziiert daraufhin den *XML*-Datenbestand, der über die Klasse *de.wsi.xjml.XmlDataRepository* gekapselt wird. Diese Klasse erlaubt über die Methode *getNextNode()* eine Iteration über alle relevanten *XML*-Dokumente dieses Repositories unabhängig davon, ob es sich um

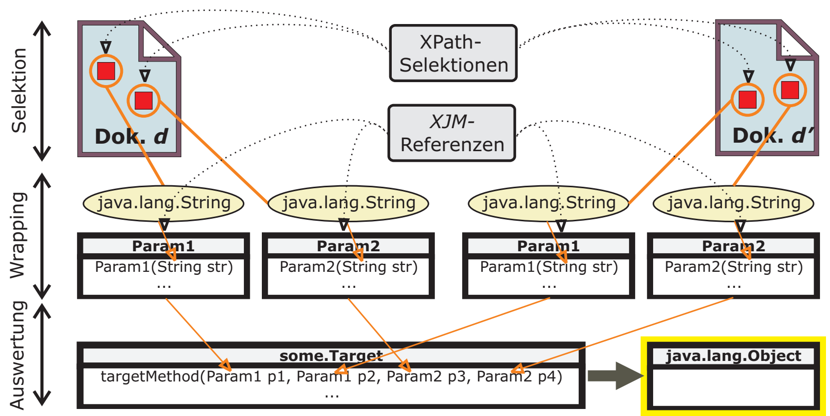

Abbildung 7.3: Auswertung eines *XJM*-Ähnlichkeitsaspektes für zwei *XML*-Dokumente d und d'

eine Ergebnismenge einer *XML*-Datenbank-Anfrage, eine URL oder ein Verzeichnis des Dateisystems handelt. Parametriert wird die Instanziierung dieser Klasse durch Übergabe des *Statement*-DOM-Knotens.

Das Laden von durch *JavaRef*-Elemente referenziertem Java-Byte-Code geschieht ebenfalls in der Initialisierungsphase und für jede Klasse genau ein Mal. Hierfür wurde ein spezieller Java-Klassenlader entwickelt, der alle nötigen (z.B. auch evtl. vorhandene Basisklassen) von der angegebenen URL über das HTTP-Protokoll lädt.

7.8.2 Auswertung der *XJM*-Ähnlichkeitsaspekte

Abbildung 7.3 stellt die Auswertung eines *XJM*-Ähnlichkeitsaspektes dar. Diese Auswertung definiert eine Abbildung von zwei *XML*-Dokumenten auf ein Java-Objekt das den Grad der Ähnlichkeit der *XML*-Dokumente angibt. Die Berechnung dieses Wertes zerfällt in drei Phasen:

1. Selektion
2. Wrapping
3. Auswertung

```
36 public Object createNewInstance(ElementNode javaRefNode, String initValue) {

38   // custom class loading according to codebase and classname:
     // Class cls = ...
40
     Constructor[] ctors = cls.getConstructors();
42   Object[] initArgs = new Object[1];
     initArgs[0] = initValue;
44   for (int i=0; i<ctors.length; i++) {
       Class[] params = ctors[i].getParameterTypes();
46     if (params.length != 1) continue;
       if (!params[0].getName().equals("java.lang.String")) continue;
48     return ctors[i].newInstance(initArgs);
     }
50   //...
   }
```

Listing 7.3: Dynamisches Parameter-Wrapping der *Projection*-Klasse

Selektion Die Selektion der *XML*-Fragmente ist durch entsprechende *Query*- bzw. *Fixed*-Elemente in *XJML* definiert. Wie bereits erwähnt, ergibt eine Selektion durch ein *Fixed*-Element direkt den Wert des entsprechenden *Value*-Attributs dieses Elements. Die Auswertung eines *Query*-Elements ergibt sich aus der Auswertung der darin enthaltenen *XQL*- oder *XPath*-Selektion auf den Dokumenten d und d'. Die Selektion wird symmetrisch auf beiden Dokumenten ausgeführt und ergibt zwei *XML*-Fragmente, die als Java-Objekt vom Typ *String* verwaltet werden. Falls die Selektionen mehr als jeweils ein einzelnes *XML*-Fragment selektieren, wird jeweils nur das erste Fragment betrachtet. Zur Auswertung der *XQL*-Ausdrücke wird die *GMD-IPSI XQL-Engine* [GMD] verwendet, die für nicht-kommerzielle Anwendungen kostenfrei verfügbar ist. *XPath*-Ausdrücke werden durch die kostenfreie *Xalan*-Bibliothek [Apad] berechnet.

Wrapping Wie in Abbildung 7.3 gezeigt, werden die selektierten *XML*-Fragmente in Form von Java-String-Objekten an die Konstruktoren derjenigen Klassen gegeben, die durch die zugehörigen *JavaRef*-Elemente referenziert wurden. Diese dynamische Instanziierung wird durch die *Projection*-Klasse des *XJML*-Java-Pakets implementiert.

Die *Projection*-Klasse (vgl. Listing 7.3) lädt innerhalb der *createNewInstance()*-Methode zunächst die durch *javaRefNode* (Zeile 36) referenzierte Java-Klasse mit Hilfe einer Instanz der Klasse *de.wsi.misc.MyClassLoader*. Das Ergebnis wird in der Variablen *cls* vom Typ *Class* abgelegt. In Zeile 44-47 wird die Referenz auf den Konstruktor dieser Klasse gesucht, der genau einen Parameter vom Typ *String* akzeptiert. Wird dieser gefunden, so wird dieser Konstruktor in Zeile 48 mit dem Argument *initValue* über *Java Reflection* aufgerufen. Die Variable *initValue* enthält hierbei das zuvor selektierte *XML*-Fragment.

An dieser Stelle wird eine erste Anforderung an *XJM*-Wrapper-Klassen deutlich: Die Wrapper-Klasse muss einen Konstruktor implementieren, der genau einen *String*-Wert als Übergabeparameter akzeptiert. Tabelle 7.2 listet sämtliche Anforderungen dieser Art auf. Innerhalb des Konstruktors einer *XJM*-Wrapper-Klasse kann das übergebene *XML*-Fragment geparst und weiter verarbeitet werden. Die Klasse *CANopenNodeState* (vgl. Listing 7.2, Zeile 8) parst beispielsweise den übergebenen String zunächst in eine Integer-Zahl und interpretiert diesen als eine CANopen-spezifische Knotenzustandskodierung gemäß [Can96b]. Insbesondere bei einer Verwendung von *XJM_Eval* innerhalb des INSIGHT-Systems können diese Wrapper-Klassen die gerätetypspezifische Serialisierung von Parameterwerten in *DeMML*-Fragmente umkehren, und so wieder die eigentlichen Parameterwerte einer Verarbeitung in Java, z.B. einer Ähnlichkeitsbewertung zugänglich machen.

Auswertung Die Auswertung eines *XJM*-Ähnlichkeitsaspektes besteht aus dem Ausführen einer durch eine *XJM*-Referenz identifizierten Methode mit den in der Wrapping-Phase erzeugten Wrapping-Objekten. Referenziert wird diese Methode durch die *Target*- bzw. *JavaRef*-Elemente des zugehörigen *Mapping*-Elements (vgl. Listing 7.1, Zeile 24 u. 25). Implementiert wird diese Funktionalität durch die Methode *invokeMethod()* der bereits erwähnten *Projection*-Klasse. Parametriert wird diese Methode wieder durch Übergabe des entsprechenden *JavaRef*-Knotens und einem Array vom Typ *Object*, das die zuvor instanziierten Wrapping-Objekte enthält. Wie in Abbildung 7.3 angedeutet, stammen die einzelnen Wrapping-Objekte ($p1,...,p4$) abwechselnd aus Selektionen aus dem ersten ($p1,p3$) und dem zweiten Dokument ($p2,p4$). Die *invokeMethod()*-Methode sucht wieder zunächst die Referenz auf das Objekt vom Typ *java.lang.reflect.Method* der referenzierten Klasse, das eine Signatur entsprechend der Objekte des übergebenen Parameter Arrays aufweist und ruft diese über *invoke()* auf. Das Resultat dieses Aufrufs ist wieder ein von *java.lang.Object* abgeleitetes Java-Objekt.

Die in Listing 7.1 referenzierte Ähnlichkeitsaspektfunktion *hammingDistance()* berechnet beispielsweise die Anzahl der Bits, in denen sich die Binärkodierung zweier hexadezimal kodierter Integer-Zahlen *hi1* und *hi2* (vgl. Abbildung 7.4) voneinander unterscheiden. Der *XPath*-Ausdruck in Listing 7.1, (Zeile 15-16) selektiert hierfür die durch INSIGHT erfasste Belegung der acht Eingangskanäle eines digitalen CANopen-E/A-Moduls zu zwei unterschiedlichen Zeitpunkten. Der Parameter *len1* bzw. *len2* (vgl. Abbildung 7.4), der durch ein *Fixed*-Element selektiert wurde (Zeile 20), gibt die Anzahl der zu untersuchenden Bit-Stellen, angefangen beim *Least Significant Bit*, an. Ein typisches Merkmal von Ähnlichkeitsaspektfunktionen sind diese „verdoppelten“ Signaturen für abwechselnd Parameter aus dem ersten und dem zweiten Dokument.

7.8.3 Auswertung des *XJM*-Ähnlichkeitsmaßes

Ein *XJM*-Ähnlichkeitsmaß wird laut Definition 7.4 durch einen Vektor $\vec{A}$ von *XJM*-Ähnlichkeitsaspekten und $\vec{W}$ von Gewichten sowie eine Evaluationsfunktion festgelegt. Die Evaluationsfunktion konsolidiert hierbei die Teilergebnisse der Ähnlichkeitsaspektfunktionen, die innerhalb der einzelnen *Mapping*-Elemente spezifiziert waren. Die Evaluationsfunktion wird durch das *Evaluation*-Element, das sich innerhalb eines *XJML*-Dokumentes an die Liste der *Mapping*-Elemente anschließt, referenziert (siehe Abbildung 7.2). Die Referenz wird wieder über ein entsprechendes *Target* bzw. *JavaRef*-Element festgelegt. Die referenzierte Methode muss als Parameter ein Java-Array vom Typ *java.lang.Object* akzeptieren, wobei die einzelnen *Object*-Objekte innerhalb der Methodenimplementierung auf spezialisierte Typen gecastet werden können. Dieses Array wird während der Auswertung eines *XJM*-Ähnlichkeitsmaßes mit den Ergebnisobjekten der einzelnen Ähnlichkeitsaspektfunktionen in der Reihenfolge ihrer Spezifikation gefüllt. Falls die referenzierte Evaluationsmethode zusätzlich ein Array von *double*-Werten als zweiten Parameter akzeptiert, werden in diesem Array die gemäß $\vec{W}$ zu den einzelnen Ergebnissen assoziierten Gewichte übergeben. Der Ergebnistyp der Evaluationsfunktion wurde auf den Java-Datentyp *Double* festgelegt, der den Grad der Ähnlichkeit (Abstandswert) widerspiegelt.

7.8.4 Ergebnisausgabe

Die Methode *nearestNeighborSearch()* der Klasse *de.wsi.xjml.Evaluation* löst ein *XJM*-NN-Problem, indem sie das *XJM*-Ähnlichkeitsmaß für alle relevanten Dokumentenpaare berechnet, den minimalen Abstandswert ermittelt und das Ergebnis als Objekt der Klasse *de.wsi.xjml.SearchResult* zurückgibt.

7.9 Erweiterbarkeit und Flexibilität

Die funktionale Erweiterbarkeit des Systems basiert auf dem *XJM*-Konzept (vgl. Abschnitt 7.6.2) bzw. der Realisierung als *XML*-basiertes Software-Framework. Es können nahezu beliebige in Java kodierte Ähnlichkeitsaspekt- und Evaluationsfunktionen frei auf dem Internet referenziert und damit in ein Ähnlichkeitsmaß integriert werden, ohne dass das *XJM*-Kernsystem neu übersetzt werden muss. Bei der Erweiterung des Systems um neue Methoden gelten jedoch bestimmte Anforderungen an die neuen Java-Klassen und Methoden, die im nächsten Abschnitt erläutert werden.

7.9.1 Integration neuer *XJM*-Ähnlichkeitsaspekte

Die Integration neuer *XJM*-Ähnlichkeitsaspekte erfolgt durch Referenzierung entsprechender neuer Java-Klassen durch *JavaRef*-Elemente. Die Klassen können hierbei, wie bereits erwähnt, frei auf dem Internet lokalisiert sein. Es ist beispielsweise möglich, Klassen von dritter Seite, die an einer zentralen Stelle bereitgestellt werden, in einem *XJM_Eval*-System zu verwenden. Auf diese Weise kann Java-Logik für komplexe *XJM*-Ähnlichkeitsaspekte (z.B. Bildverarbeitungsmethoden, Genom-Analysemethoden oder auch *XML*-Strukturvergleiche[4]) von zentraler Stelle gepflegt und auf dem Internet für laufende *XJM_Eval*-Systeme dynamisch zur Verfügung gestellt werden. Wichtig ist hierbei, dass aufgrund der Einbindung über *Java Reflection* die Integration neuer Klassen keine erneute Übersetzung des *XJM_Eval* Kern-Systems erfordert. Dies ist insbesondere dann von Vorteil, wenn, wie in einem Data-Mining-Prozess üblich, mit verschiedenen Ähnlichkeitsmaßen und Konfigurationen experimentiert werden muss. Zusätzlich könnten Implementierungen der in Abschnitt 7.3 vorgestellten Systeme über minimale Wrapper-Klassen in ein *XJM*-Ähnlichkeitsmaß integriert werden.

Die Entwicklung neuer Funktionalität kann völlig unabhängig vom *XJM_Eval* Kernsystem erfolgen, jedoch müssen hierbei einige Richtlinien befolgt werden. Tabelle 7.2 gibt eine Übersicht über die Anforderungen an *XJM*-Wrapper-Klassen, Ähnlichkeitsaspekt- und Evaluationsmethoden.

Gemäß Tabelle 7.2 müssen die einzelnen Ähnlichkeitsaspekt- und Evaluationsmethoden als öffentliche, statische Klassenmethoden implementiert werden, wobei der Rückgabewert der Evaluationsmethoden zusätzlich auf den Typ *Double* festgelegt wurde. Diese Anforderung stellt i.d.R. keine Einschränkung des Entwicklungsprozesses für neue *XJM* Funktionalität oder die Integration bestehender Abstandsfunktionen über schlanke Wrapper-Klassen dar. Auch die Festlegung der Signatur einer *XJM*-Evaluationsmethode auf einen Parameter vom Typ *java.lang.-Object[]* ist keine wirkliche Einschränkung, da das Java-Array beliebige Java-Objekte enthalten kann. Diese Verpackung der einzelnen Parameter in ein Java-Array hat Implementationsgründe und beschleunigt hauptsächlich die *invoke()*-Methode der Klasse *java.lang.reflect.Method*.

Von entscheidender Wichtigkeit sind jedoch die dynamischen Typabhängigkeiten der einzelnen Methoden. Diese Abhängigkeiten kommen dadurch zustande, dass

[4]Denkbar wäre hier z.B. eine Implementierung der *Tree Edit Distance* Funktion.

	Aufgabe	Rückgabewert	Restriktionen
Wrapper-Klasse	Transformation eines *XML*-Fragments in ein Java-Objekt	Neue Instanz dieser Klasse	Konstruktor muss einen *String*-Parameter akzeptieren
Ähnlichkeitsaspektmethode	Berechnung eines *XJM*-Ähnlichkeitsaspektes	Beliebiges Java-Objekt	Eine Signatur muss zu definierten Wrapper-Klassen passen Methode muss *public static* deklariert sein
Evaluationsmethode	Konsolidierung der Ergebnisse der Ähnlichkeitsaspektfunktionen	*java.lang.-Double*	Methode muss ein Java-Array von *Object*-Objekten akzeptieren Methode kann zusätzlich ein Array ($\vec{W}$) von *double*-Werten akzeptieren (Gewichtung) Methode muss *public static* deklariert sein

Tabelle 7.2: Anforderungen an *XJM*-Klassen und -Methoden

im Verlauf der Auswertung eines *XJM*-NN-Problems die einzelnen dynamisch geladenen Java-Klassen und deren Methoden miteinander kommunizieren, indem sie Werte weiterreichen bzw. konsolidieren. Hierbei müssen jeweils die Rückgabewerte und Signaturen kompatibel sein. Abbildung 7.4 zeigt beispielsweise die Typabhängigkeiten der Methoden des *XJML*-Dokuments aus Listing 7.1 und 7.2.

Es liegt in der Verantwortung des Benutzers, bei der Erstellung eines *XJML*-Dokuments darauf zu achten, dass diesen Typabhängigkeiten Rechnung getragen wird, da sämtliche Aufrufe der Methoden über *Java Reflection* erfolgen und somit etwaige Typinkonsistenzen erst zur Laufzeit erkannt werden. Es gibt jedoch keinerlei Typabhängigkeiten bzgl. neuer Klassen und dem *XJM_Eval* Kernsystem.

7.9.2 Flexibilität durch *XJML*-Dokumente

Ähnlich wie bei der Konfiguration eines INSIGHT-Systems durch ein *DeMML*-Dokument bietet der *XML*-bzw. *XJML*-basierte Ansatz zur Definition von Ähnlichkeitsmaßen für *XML*-Dokumente einen hohen Grad an Flexibilität. Die *XJML*-Dokumente können mit speziellen *XJML*-Tools oder allgemeinen *XML*-Werkzeugen und Programmbibliotheken auf einfache Weise erstellt, angepasst und

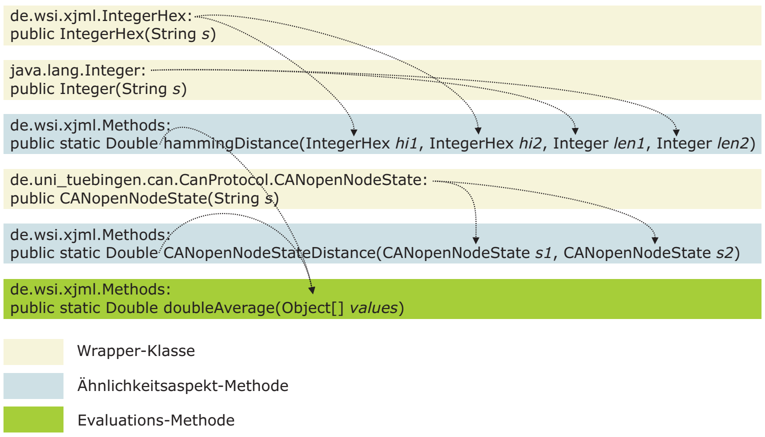

Abbildung 7.4: Typabhängigkeiten für das *XJML*-Dokument aus Listing 7.1 und 7.2

verwaltet werden. Sie können als Dokumente direkt in einem zentralen *XML*-Datenbanksystem abgelegt und zur Verfügung gestellt werden. Ein Unternehmen kann somit einen Fundus von Ähnlichkeitsmaßen für spezielle Anwendungen (vgl. Abschnitt 7.7) pflegen und den Benutzern effizient (z.B. über eine HTTP-basierte Queryschnittstelle auf dem Internet) zur Verfügung stellen.

Dieser offene Ansatz ist insbesondere deshalb sinnvoll, weil die einzelnen Java-Klassen, die in einem *XJML*-Dokument referenziert werden, nicht lokal auf dem Evaluationssystem vorhanden sein müssen, sondern erst zur Laufzeit vom *XJM_Eval*-Kernsystem dynamisch über das Netz geladen werden. Es können also beliebige (auch fremde) *XJML*-Dokumente getestet werden, ohne zuvor neue Java-Pakete installieren zu müssen.

In einem verteilten *XML*-basierten Informationssystem, wie z.B. INSIGHT, integrieren sich *XJML*-Dokumente besonders gut, da dort i.d.R. bereits eine auf *XML*-Dokumente ausgerichtete Infrastruktur vorhanden ist (*XML*-Datenbanken, Servlets, Style-Sheets usw.).

7.9.3 *XJML*-basierte Auswertung von *XML*-Dokumenten

XJM_Eval kann prinzipiell zusätzlich zur Berechnung der Ähnlichkeit von *XML*-Dokumenten auch zur Auswertung des Inhalts jeweils allein stehender *XML*-Dokumente verwendet werden. So wäre es beispielsweise möglich, Test-Verfahren durch *XJML*-Dokumente zu spezifizieren. Im Kontext der INSIGHT-Infrastruktur könnte ein derartiges Test-Verfahren beispielsweise berechnen, ob sich die momentan gemessene Temperatur eines Meßfühlers zwischen zwei Temperatur-Schwellwerten befindet oder nicht.

Aufgrund der Realisierung von *XJM_Eval* als *XML*-basiertes Software-Framework bestehen konzeptionell keine Einschränkungen bzgl. der Berechnungsfunktionen, da diese jeweils dynamisch über entsprechende *XJM*-Referenzen eingebunden werden können.

7.10 Integration von *XJM_Eval* in die INSIGHT-Infrastruktur

Eine exemplarische Integration von *XJM_Eval* in die INSIGHT-Infrastruktur konnte mit minimalem Aufwand durch die Erstellung eines entsprechenden Java-Servlets realisiert werden. Dieses Servlet generiert dynamisch ein *HTML*-Formular (siehe Abbildung 7.5), das eine entfernte Parametrierung und Ausführung von *XJM_Eval* erlaubt.

Das *HTML*-Formular unterstützt insbesondere direkt die Verwendung des aktuellen Systemzustandes einer entfernten Anlage als Anfragedokument d und das Resultat einer Query an ein entferntes Tamino-Datenbanksystem als Suchraum $\vec{D}$. Aufgrund der vorhandenen HTTP-Schnittstellen des INSIGHT-Systems und der Tamino-Datenbank kann sowohl d als auch $\vec{D}$ einfach durch Angabe entsprechender URLs spezifiziert werden. Die zu verwendenden *XJML*-Dokumente können aus dem lokalen Dateisystem oder über eine URL (also wieder auch aus einer *Tamino*-Datenbank) in das Textfeld geladen und bei Bedarf angepasst werden. Wenn die Suchanfrage durch Drücken der *Search*-Taste gestartet wird, wird das definierte Ähnlichkeitsmaß auf dem entfernten Server ausgewertet und das Ergebnis als neue *HTML*-Seite dargestellt.

7.11 Fallstudie 4: *XJML*-basierte Fehlersuche für eine Automatisierungsanlage

Evaluiert wurde die Leistungsfähigkeit der entwickelten Konzepte anhand der Daten, die bei der Überwachung der in Kapitel 5 vorgestellten Demonstrationsanlage durch das INSIGHT-System erhoben wurden. Zielsetzung war es, ein *XJML*-Dokument und entsprechende Ähnlichkeitsaspektfunktionen zu erstellen, die die in Fallstudie 1 manuell erzeugten Fehlerzustände zuverlässig aus der Masse der

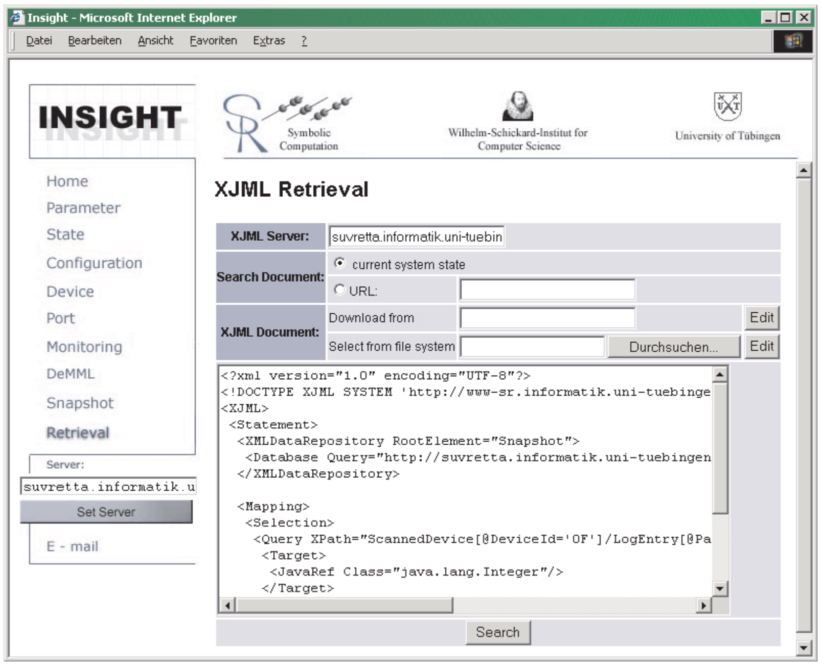

Abbildung 7.5: Integration von *XJM_Eval* in das INSIGHT-System

Monitoring-Daten herausfiltern (vgl. Abschnitt 5.6).

Ein *XJML*-Dokument, das diese Aufgabe zuverlässig erfüllt, ist in Listing A.8 (S. 242) vollständig angegeben. Im Einzelnen werden in diesem Dokument folgende Ähnlichkeitsaspekte berücksichtigt:

1. Fehlerregister der CANopen-Knoten
2. Anzahl aufgetretener Fehler innerhalb der CANopen-Knoten
3. Hamming-Distanz der digitalen Ein-/Ausgangsbelegungen der E/A-Module
4. Knotenzustand der CANopen-Knoten
5. Zustand des Transfersystems
6. Position des Robotergreifers im Raum

Der Vergleich des Fehlerregisters (vgl. Listing A.8, Zeilen 23-38) benützt die Klasse *java.lang.Integer* als Wrapper-Klasse und vergleicht die ganzzahligen CANopen Fehlercodes lediglich bzgl. gleich oder ungleich. Beim Vergleich der Anzahl der aufgetretenen Fehler (z.B. Zeilen 54-69) wird die Methode *integerDistance()* referenziert, die neben den entsprechenden *Integer*-Werten mit einem *Fixed*-Wert der Maximalabweichung parametriert wird. Die Methode *hammingDistance()* zur Berechnung der Hamming-Distanz wurde bereits in Abschnitt 7.7 vorgestellt. Die Methode *CANopenNodeStateDistance()* (vgl. Zeilen 50 u. 51) löst den Grad der Ähnlichkeit von CANopen-Knotenzuständen (*operational*, *pre-operational* ... [Can96d]) über die Wrapper-Klasse *CANopenNodeState* (vgl. Zeilen 43-46) und eine statische Abstandstabelle auf.

Der Zustand des Transfersystems (transfer-nach-links, transfer-nach-rechts, stop) ist als binär-kodierter Datenanteil von entsprechenden CAN-Nachrichten (8 Byte) im Automatisierungssystem zugänglich. Diese CAN-Nachrichten wurden durch INSIGHT mit Hilfe der Klasse *CANMsgParameter* (siehe Abschnitt 5.6.2) erfasst und serialisiert. Die *XJML*-Wrapper-Klasse *CanMessage* kehrt diese Serialisierung um und macht die 8 CAN-Datenbytes wieder zugänglich, die dann von der Methode *CANMsgDistance()* miteinander verglichen werden (Zeilen 232-246).

Die Methode *SCARASpatialDistance* berechnet aufgrund der unterschiedlichen Achspositionen des SCARA-Roboters die Vorwärtskinematik des Roboterarms und ermittelt daraus die räumliche Entfernung der beiden Greiferpositionen (Zeilen 266-301). Die Geometrieinformationen des verwendeten Roboters werden durch ein *Fixed*-Element mit assoziierter Wrapper-Klasse *SCARAGeometry* an die Ähnlichkeitsaspektfunktion übergeben (Zeilen 290-295).

Die generierten Fehlerzustände konnten über diese *XJM_Eval*-Konfiguration zuverlässig diagnostiziert, d.h. aus dem bestehenden Datenbestand gefiltert werden. Aufgrund der Beschränktheit des Systems und der Zahl der generierten und ermittelten Fehlerzustände erscheinen zusätzliche Fallstudien sinnvoll, die weiteren Aufschluss über die Einsetzbarkeit/Anpassbarkeit des Systems geben können.

7.12 Zusammenfassung und Bewertung

Der *XJM*-Ansatz beruht auf der flexiblen Verbindung von *XML*-Querysprachen mit der Java-Programmiersprache ($a4$). Die Verbindung an sich wird durch ein *XML*-Dokument der *XML to Java Mapping Language* definiert und dynamisch während der Evaluation einer NN-Suche über *Java Reflection* hergestellt. Es lässt sich nahezu beliebige Java-Logik in die Definition von Ähnlichkeitsmaßen für *XML*-Dokumente integrieren, ohne spezielle Kenntnisse über die Implementierung des *XJM*-Kernsystems zu besitzen ($a2$). Die *XML*-Querysprachen leisten die

Selektion der *XML*-Fragmente (Features), die für die referenzierte Java-Logik relevant sind. Die *XML*-basierte Spezifikation des Zusammenspiels der einzelnen Java-Klassen und *XML*-Selektionen unterstützt zusätzlich die Flexibilität und Offenheit des Systems, da die einzelnen *XJML*-Dokumente innerhalb einer verteilten *XML*-Infrastruktur auf bequeme Weise verwaltet und verwendet werden können ($a1$, $a3$, $a4$).

Insbesondere in dem intendierten Anwendungsgebiet, der Suche nach dem Dokument, das zu einem gegebenen Dokument (z.B. die *XML*-basierte Repräsentation eines Fehlerzustandes einer industriellen Automatisierungssanlage) die größte Ähnlichkeit aufweist, ist die Definition der Ähnlichkeit nicht von vorneherein klar vorgegeben, sondern muss vom Benutzer i.d.R. interaktiv erforscht oder über andere Methoden ermittelt werden. In diesem Kontext ist die oben beschriebene Flexibilität besonders vorteilhaft.

Ein wichtiger Aspekt ist die Offenheit des Systems bzgl. serialisierter Datentypen und Datenkodierungen. So ist das Kernsystem nicht auf die Verwendung bestimmter Datenformate beschränkt, sondern kann ohne erneute Übersetzung beliebige Datentypen durch Referenzierung entsprechender Wrapper-Klassen integrieren ($a5$). Das INSIGHT-System aus Fallstudie 1 (vgl. Abschnitt 5.6) legt beispielsweise im JPG-Format kodierte Bilder (siehe auch Abbildung 4.7 auf Seite 61) als *XML*-Fragmente in entsprechenden *DeMML*-Dokumenten ab. Für diese *XML*-Fragmente können problemlos entsprechende *XJM*-Wrapper-Klassen und -Ähnlichkeitsaspektfunktionen erstellt werden, die mit Hilfe von Graphikbibliotheken die Ähnlichkeit der Bilddaten bewerten ($a2$). So könnte beispielsweise die Ähnlichkeit der Kamerabilder einer Anlage, die zu zwei unterschiedlichen Zeitpunkten entstanden sind, ermittelt werden.

Weitere interessante Datentypen wären beispielsweise Audiodatentypen, um die akustischen Frequenzstrukturen einer laufenden Anlage zu vergleichen. Dies ist insbesondere dann sinnvoll, wenn bei bestimmten Geräten (z.B. Motoren) ein erhöhter Verschleiß akustisch wahrnehmbar wird und diese akustische Veränderung sonst nur von Experten erkannt wird.

Das *XJM_Eval*-System wurde gemäß $a5$ speziell darauf ausgelegt, inhaltsbezogene Ähnlichkeitsaussagen besonders gut zu unterstützen. Für die Definition von Ähnlichkeitsmaßen sollte daher eine gewisse Kenntnis über die logische Struktur der zu bewertenden Dokumente vorhanden sein. Diese Einschränkung ist immer dann problemlos, wenn Dokumente mit bekannten DTDs verglichen werden, für die die logische Struktur klar definiert ist. Insbesondere die zunehmende Verwendung von *XML* bei der Prozessüberwachung und Generierung von Log-Dateien allgemein resultiert in einem erhöhten Aufkommen derartig strukturierter Dokumente.

Die zusätzliche Integration struktureller Ähnlichkeitsaspekte, z.B. der *Edit Distance* für Baumstrukturen oder auch des *XML Diff* Werkzeugs (vgl. Abschnitt 7.3), als *XJM*-Ähnlichkeitsaspektfunktion könnte, falls dies gewünscht ist, den Grad dieser Ähnlichkeit in ein *XJM*-Ähnlichkeitsmaß integrieren. Andernfalls werden strukturell unterschiedliche *XML*-Dokumente evtl. als zu unterschiedlich bewertet, da die *XML*-Selektionen u.U. leere oder unerwünschte *XML*-Fragmente selektieren.

Ein generischer, selbstorganisierender Ansatz für beliebig strukturierte *XML*-Dokumente, der beispielsweise ein a priori unbekanntes Ähnlichkeits-Clustering durchführt, müsste auf die expliziten *XML*-Selektionen verzichten und diese dynamisch generieren.

Die durchgeführte Fallstudie hat belegt, dass es mit *XJM_Eval* problemlos möglich ist, die Ähnlichkeit von serialisierten Prozesszuständen einer industriellen Automatisierungsanlage zu bewerten, sofern die Prozesszustände in *XML*-Format vorliegen. Der *XJM*-Ansatz und *XJM_Eval* sind aufgrund ihrer Offenheit und Flexibilität weiterhin dazu in der Lage, Ähnlichkeitsbewertungen für beliebige *XML*-basierte Datenrepräsentationen zu definieren und auszuwerten.

Kapitel 8

Zusammenfassung

Das INSIGHT-System kann in seiner Gesamtheit aufgefasst werden als eine flexible, erweiterbare Infrastruktur zur Integration von Geräten in offene Informationssysteme für das World Wide Web. Als universelles Datenformat zur Konfiguration des Systems und zur Repräsentation von aggregierten Gerätedaten wird die *XML*-Applikation *DeMML* verwendet, die gleichzeitig auch das Datenbankschema zur persistenten Speicherung von Prozesszuständen begründet. Als Datenbanksystem wird primär das native *XML*-Datenbanksystem *Tamino* eingesetzt, das über eine schlanke HTTP-Query- und Managementschnittstelle eingebunden wird.

Die Datenaggregation wird durch das *XML*-basierte Software-Framework *DeviceInvestigator* koordiniert, das die flexible Einbindung von unterschiedlichsten Datenerfassungskonzepten über generische Schnittstellen unterstützt. Durch die Realisierung als *XML*-basiertes Java-Software-Framework zeichnet sich das *DeviceInvestigator*-Konzept durch seine Plattformunabhängigkeit, Offenheit, Skalierbarkeit, Flexibilität und die standardisierte Datenbankanbindung aus. Insbesondere die gute Skalierbarkeit des Systems ermöglicht auch den Einsatz dieses Systems auf kostengünstigen Embedded-Systemen.

Geräteparameterwerte werden innerhalb von INSIGHT grundsätzlich als serialisierte Zeichenketten in Form von *DeMML*-Fragmenten verwaltet. Jedem Parameterwert wird über die Angabe einer weiteren Zeichenkette ein Datentypname zugeordnet, der jedoch nicht einem allgemein akzeptierten und gepflegten Verzeichnis von gültigen Datentypnamen entnommen werden muss, sondern frei gewählt werden kann. Die Semantik des gewählten Datentyps steckt in der eindeutigen Zuordnung einer Java-Hardwarezugriffsklasse zu jeder Parameterbeschreibung. Diese Java-Klasse enthält nicht nur die Logik zum protokollspezifischen Zugriff auf den beschriebenen Geräteparameter, sondern auch das Verfahren zur Serialisierung des Parameters[1]. Es bestehen daher prinzipiell keine Einschränkungen bzgl. der verwendeten Datentypen (z.B. Video- oder Audiodatentypen).

Identifiziert werden die einzelnen Java-Harwarezugriffsklassen durch das *XML to Java Mapping* Konzept bzw. Elemente der entsprechenden *XML*-Applikation *XJML* (*XML to Java Mapping Language*). Die zur Identifikation verwendete Kombination von URL und Java-Klassennamen kann als Analogon zu dem COM/DCOM-Konzept der UUID zur global eindeutigen Identifizierung von Software-Komponenten betrachtet werden, mit dem Vorteil, dass die Identifikation selbst bereits Protokoll und Lokation der Klasse innerhalb des Internet enthält. Diese Art der Identifikation gewährleistet, dass *DeMML*-Parameterbeschreibungen immer eindeutig sind, unabhängig davon, ob die entsprechenden Hardwarezugriffsklassen auf dem Evaluationssystem lokal vorhanden sind oder nicht. Bei einer Auswertung einer derartigen *DeMML*-Konfiguration lädt *DeviceInvestigator* ggf. die spezifizierten Hardwarezugriffsklassen automatisch von der angegebenen URL.

Die Client/Server-Architektur des INSIGHT-Systems bindet die *DeviceInvestigator*-Komponente in ein offenes Internet-basiertes Informationssystem ein. Es wird sowohl direkte Kommunikation über Java-RMI als auch über HTTP und Java-Servlets zur Überquerung von Firewalls unterstützt. Ein Benutzer kann über ein mächtiges Java-Client-Applet oder direkt mit dem Web-Browser über eine HTTP-Schnittstelle lesend und schreibend auf das System zugreifen. Insbesondere wird die Visualisierung des aktuellen Parameterzustands der entfernten Anlage in unterschiedlichen Sichten, der schreibende Zugriff auf einzelne Parameterwerte und der Zugriff auf frei konfigurierbare protokollspezifische Managementfunktionalität einzelner Geräte unterstützt.

Als Recherchekomponente wurde das *XJM_Eval*-System zur flexiblen Ähnlichkeitsbewertung von *XML*-Dokumenten entwickelt. Diese Funktionalität ermöglicht innerhalb von INSIGHT die Realisierung eines Diagnoseunterstüzungssystems für Fernwartungsaufgaben. Ein Benutzer kann ortstransparent den in *DeMML* kodierten Zustand einer entfernten, defekten Anlage mit den Datenbeständen einer entfernten, zentralen *DeMML*-Datenbank von bereits behobenen Fehlerzuständen vergleichen und so Diagnosevorschläge erhalten. Wichtig ist hierbei eine hohe Flexibilität der Ähnlichkeitsmaßdefinition, die durch die Realisierung von *XJM_Eval* als *XML*-basiertes Software-Framework erreicht wird.

Die Konzeption und Realisierung von *XJM_Eval* stellte insbesondere deshalb eine

[1]Diese Serialisierung kann später in einem Recherchekontext durch das *XJM_Eval*-System wieder rückgängig gemacht werden.

Herausforderung dar, weil der Vergleich der *XML*-Dokumente sowohl die Struktur als auch den Inhalt und die Typisierung der durch die Hardwarezugriffsklassen serialisierten Daten (Floats, Integers, CANopen-Knotenzustände, Bit-Strings, JPGs, ...) berücksichtigen sollte. Gelöst wurde dieses Problem wiederum durch das *XJM*-Konzept, das in diesem Fall zur Integration von Java-Logik in *XML*-Querysprachen eingesetzt wurde.

Die *XML*-Applikation *XJML* erlaubt in diesem Kontext die flexible Definition von struktur- und inhaltsbezogenen Ähnlichkeitsmaßen für beliebige *XML*-Dokumente und insbesondere für solche, die serialisierte Daten enthalten. Die Deserialisierung wird hierbei an durch das *XJM*-Konzept eindeutig identifizierte Klassen delegiert, die durch *XML*-Querysprachen selektierte *XML*-Fragmente wieder in tatsächliche Werte des angegebenen Datentyps deserialisieren können. Diese Werte können dann mit Hilfe von ebenfalls durch das *XJM*-Konzept referenzierten Java-Klassen typspezifisch miteinander verglichen werden. Der *XJM_Eval* Ansatz erlaubt so auch die einfache Integration von spezialisierter Ähnlichkeitsbewertungssoftware (z.B. graphisches *Pattern Matching*) in *XML*-Query-Statements.

Zur Validierung der entwickelten Konzepte und Implementierungen wurden mehrere Fallstudien durchgeführt. Im Rahmen der ersten Fallstudie wurde eine aus industriellen CAN-Feldbusgeräten aufgebaute Automatisierungsanlage durch eine *DeviceInvestigator*-Komponente überwacht. Die aggregierten *Snapshot*-Dokumente, sowohl Fehlerzustände als auch Zustände des normalen Betriebs, wurden in einer entfernten *XML*-Datenbank abgelegt. Einzelne Fehlerzustände konnten im Rahmen einer weiteren Fallstudie über ein entsprechendes *XJML*-Ähnlichkeitsmaß sicher aus dem *DeMML*-Datenbestand gefiltert und somit identifiziert werden.

Im Rahmen der zweiten Fallstudie wurde gezeigt, wie *DeviceInvestigator* und *DeMML* flexibel auf neue Gerätetypen und Anlagen angewendet werden können. Konkret wurde das bestehende Internet-basierte Informationssystem für Magnetresonanztomographen der Firma Siemens mit Hilfe von *DeviceInvestigator* und *DeMML* um die Zugriffsmöglichkeit auf einzelne Geräteparameter erweitert. Es hat sich gezeigt, dass die komplette *DeviceInvestigator*-Infrastruktur übernommen werden konnte und nur sehr einfache Java-Wrapper-Klassen zur Einbindung der bereits vorhandenen Geräte-Proxy-Klassen erstellt werden mussten.

Die dritte Fallstudie ist ein INSIGHT-System für diverse industrielle CANopen-Geräte und eine steuerbare Digitalkamera, das die INSIGHT-basierte Einbindung dieser Geräte in das World Wide Web demonstriert. Das System ist rund um die Uhr zugänglich und erlaubt einen schnellen Überblick über die integrierten Geräte in unterschiedlichen Sichten sowie schreibenden Zugriff auf einzelne

Geräteparameter.

Teil II

Integration von externen Komponenten in Internet-basierte Lehr-/Lernumgebungen

Kapitel 9

Einführung

Die Ergebnisse, die in diesem Teil der Arbeit vorgestellt werden, sind hauptsächlich im Rahmen des Projektes *Automatisierte Anlagen und Informatik virtueller Systeme* entstanden und bilden Vorstufen oder Spin-Offs der INSIGHT-Entwicklung (siehe Teil 1). Die Grundthematik ist hierbei die möglichst offene Integration industrieller Geräte und Anlagen in Internet-basierte Lehr-/Lernumgebungen. Der übergeordnete Rahmen dieses Projekts ist das Verbundprojekt *Verbund Virtuelles Labor* (VVL) bzw. das Förderprogramm *Virtuelle Hochschule* des Landes Baden-Württemberg (siehe Abbildung 9.1).

In Kapitel 13 werden zusätzlich Ergebnisse, die im Rahmen des Projekts *Mathematik für BioInform@tiker* erzielt wurden, vorgestellt. Die Grundthematik ist hier die möglichst offene Integration von Computeralgebrasoftware in Internet-basierte Lehr-/Lernumgebungen.

Unter „externen Komponenten“ werden in diesem Teil der Arbeit also sowohl industrielle Anlagen als auch eigenständige Softwarepakete verstanden. Da der Schwerpunkt dieser Arbeit auf der Entwicklung offener Integrationskonzepte liegt, wird das Themengebiet *Internet-basierte Lehr-/Lernumgebungen* nur am Rande behandelt. Im Wesentlichen werden diese Umgebungen lediglich als eine spezielle Art von Internet-basierten Informationssystemen aufgefasst.

9.1 Das Projekt *Verbund Virtuelles Labor* (VVL)

Der spezielle Fokus des VVL-Projekts ist die Integration von industriellen Geräten und Anlagen (Automatisierungsanlagen, Robotern, Anlagen zur 2D- und 3D-Messtechnik ...) in Internet-basierte Lehr-/Lernumgebungen. Der Begriff „virtuell“

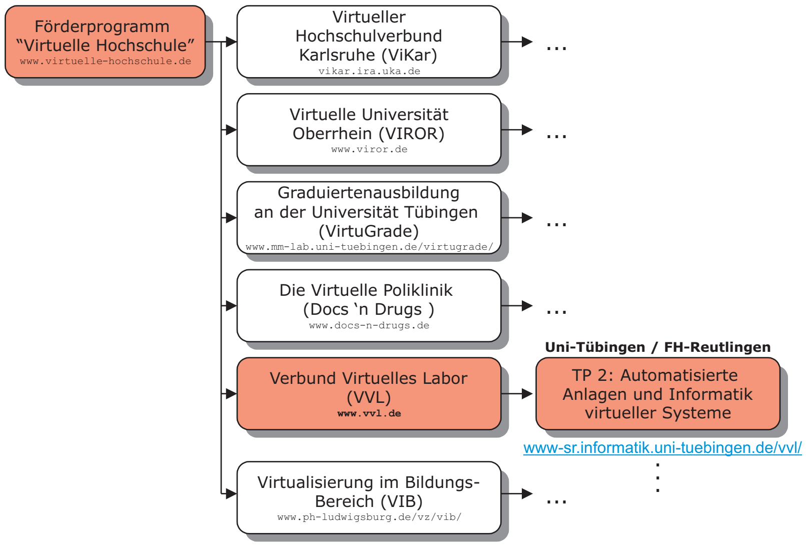

Abbildung 9.1: Die virtuelle Hochschule des Landes Baden-Württemberg

wird in diesem Zusammenhang so verstanden, dass auf reale Geräte via Internet orts- und zeittransparent zugegriffen werden kann. Der Hauptvorteil dieses Ansatzes ist, dass teure Geräte entfernter Hochschulen in lokale Praktikumsveranstaltungen oder reine Online-Angebote eingebunden werden können. Zusätzlich kann die entsprechende Internet-basierte Zugangssoftware Sicherheitsaspekte überwachen, konkurrierende Zugriffe auf die Geräte synchronisieren und so gleichzeitig das für die Geräte verantwortliche Personal entlasten und die Verfügbarkeit der Geräte für die Studenten erhöhen.

Diese Zielsetzung erfordert einen beträchtlichen Aufwand hinsichtlich der Erstellung von Konzepten zur Integration der Geräte, da insbesondere ein hohes Maß an Interaktion unterstützt werden soll, um dem Benutzer ein möglichst authentisches Gefühl für die Geräte zu geben. Der Schwerpunkt dieser Arbeit liegt daher auch primär auf der Entwicklung und Realisierung von Integrationskonzepten und nicht auf der didaktischen Aufbereitung und Verwaltung von Lerninhalten.

9.2 Anforderungen und Zielsetzung

Die Zielsetzung dieser Arbeit im Kontext des VVL-Projekts war die Erstellung von Konzepten und deren Implementierungen zur möglichst offenen Integration von CAN-Feldbusgeräten in eine zu erstellende rudimentäre Internet-basierte Lehr-/Lernumgebung. Wie in Abschnitt 1.1 bereits aufgelistet, waren hierfür folgende konkrete Zielsetzungen relevant:

$\mathbb{B}1$ Entwurf und Implementierung einer objektorientierten API für den CAN Feldbus in Java

$\mathbb{B}2$ Entwicklung und Implementierung eines offenen Konzepts zur Integration von CANopen-Feldbusgeräten in das World Wide Web basierend auf $\mathbb{B}1$

$\mathbb{B}3$ Entwurf und Implementierung eines objektorientierten Software-Frameworks zur einfachen Realisierung von feldbusbasierten Gerätesteuerungen

$\mathbb{B}4$ Konzeption und Realisierung einer rudimentären Internet-basierten Lehr-/Lernumgebung

$\mathbb{B}5$ Konzeption und Realisierung von Übungsaufgaben für die Lehr-/Lernumgebung basierend auf den Ergebnissen aus $\mathbb{B}1$-$\mathbb{B}3$ aus den folgenden Themengebieten

1. CANopen-Geräteprofile
2. Client/Server-Architekturen für Teleservice-Systeme
3. Objektorientierte Gerätesteuerung

Zusätzlich wurden als nicht funktionale Anforderungen definiert:

$\mathbb{B}6$ Zugang zu den Versuchen rund um die Uhr

$\mathbb{B}7$ Die Versuche setzen keine menschliche Interaktion auf Geräteseite voraus

$\mathbb{B}8$ Die Systemanforderungen auf Client-Seite beschränken sich auf einen Standard-Web-Browser

$\mathbb{B}9$ Erstellung von möglichst plattformunabhängigen, universell einsetzbaren Komponenten

9.3 Übersicht über virtuelle Labors

Die große Masse an Internet-basierten Lehr-/Lernumgebungen behandelt Themen der Informationstechnologie, Ingenieurwissenschaften, Sprachen usw. durch die Aufbereitung entsprechender Lerninhalte für das World Wide Web mit Hilfe von Java-Applets, *Flash*-Applikationen, *ActiveX*-Komponeten und weiteren Browser-Plug-Ins. Als Beispiele seien hier das kommerzielle *academy4me*-Netzwerk[1], das auch den Erwerb industriell anerkannter Qualifikationsgrade (z.B. MCSE-Abschlüsse) erlaubt, und das Verzeichnis kostenfreier Lehrangebote im WWW *Study Web*[2] genannt.

Spezielle Lehrangebote aus dem Bereich der Automatisierungstechnik und verwandter Disziplinen finden sich jedoch selten und beschränken sich meist auf reine Simulation. Typische Systeme aus diesem Bereich werden beispielsweise in [CS98, DW98] beschrieben. Das *VCLab*-Projekt[3] der Universität Bochum bietet ebenfalls eine ganze Reihe derartiger Experimente an, die hauptsächlich auf VRML-Simulationen beruhen und teilweise zusätzlich auf *Matlab/Simulink* aufbauen [Sch98a, Sch99]. Weiterhin wurde in jüngster Vergangenheit die Möglichkeit integriert, auch auf reale Geräte zuzugreifen [JS00]. Im Einzelnen werden zwei Internet-basierte Übungen aus dem Themenbereich Regelungstechnik angeboten, die beide die Installation entsprechender, speziell entwickelter Browser-Plug-Ins erfordern.

Frühe Systeme aus diesem Umfeld, die ebenfalls Zugriff auf reale Geräte unterstützen, sind beispielsweise in den Projekten *Onika*[4] [CCH$^+$95], *UCLA Commotion Laboratory*[5] [GSNK94] und *Second Best to Being There*[6] [ABCS96] entstanden.

Onika ist ein graphisches Programmierssystem, das das dynamische Zusammenstellen von Softwarekonfigurationen über Hyperlinks erlaubt. Im Gegensatz zu dem bereits vorgestellten Konzept der *XML*-basierten Software-Frameworks ist das Onika-System jedoch auf ein einzelnes Betriebssystem (in diesem Fall das realzeitfähige *Chimera*-Betriebssystem der Carnegie Mellon Universität) beschränkt. Das System erlaubt die X11R5/RPC-basierte dynamische Zusammenstellung und Parametrierung von Gerätesteuerungs-Software, die auf entfernten *Chimera*-Steuerungssystemen ausgeführt und überwacht werden kann.

Die Zielsetzung des *UCLA Commotion (Cooperative Motion) Laboratory* Pro-

[1] http://academy4me.msn.de
[2] http://www.studyweb.com
[3] http://www.esr.ruhr-uni-bochum.de/VCLab/
[4] http://www.cs.cmu.edu/afs/cs.cmu.edu/user/mwgertz/www/Onika.html
[5] http://muster.cs.ucla.edu
[6] http://www.ece.orst.edu/∼aktanb/distance-labs.html

jekts ist es, eine Gruppe mobiler Roboter als Internet-basierte Evaluationsplattform für neuartige Kooperationsalgorithmen der Robotik zu realisieren. Jeder Roboter ist mit einer Unix-Workstation ausgerüstet, auf der entsprechende Steuerungs-Software ausgeführt werden kann. Der Zugriff auf die Workstations und damit auf die Roboter erfolgt hauptsächlich über Telnet und FTP.

Das *Second Best to Being There* System ermöglicht den Internet-basierten Transfer von nativem C-Quellcode auf ein Unix-basiertes Steuerungssystem für einen Roboterarm, auf dem der Steuerungscode dynamisch übersetzt und ausgeführt werden kann. Die Ausführung des Steuerungsalgorithmus wird mit Kameras und Mikrophonen digital erfasst und in Form von entsprechenden Multimediadateien an die Benutzer übermittelt. Ein Benutzer kann dann die erhaltenen Ergebnisse analysieren und bei Bedarf die Steuerungslogik entsprechend anpassen.

Die Systeme der VVL-Projektpartner und die internationalen Projekte *Tele-Education in Aerospace and Mechatronics (TEAM)* und *Innovative Educational Concepts for Autonomous and Tele-Operated Systems (IECAT)* zur Integration von realen Geräten in Internet-basierte Lehr-/Lernumgebungen sind in [SRR01] beschrieben.

9.4 Lösungsansätze

9.4.1 Architektur

Abbildung 9.2 gibt einen Überblick über die entwickelten Lösungskomponenten ([GKNB99, BKGN00, GKBN01]). Das in der unteren Bildhälfte gezeigte reale Labor umfasst folgende Geräte bzw. Anlagen:

1. Werkstückvereinzelung (Klappen, Foto-Sensoren, Druckluftdüsen, E/A-Modul)

2. Leuchtschrift (E/A-Modul)

3. CANopen-E/A-Module (Selectron Dioc711, Beckhoff Bk5100)

4. Steuerbare Digitalkamera (Sony EVI D31)

Abgesehen von der Kamera kommunizieren alle Geräte über den CAN-Feldbus (siehe Abschnitt 10.1.2). Die Einbindung der Kamera wurde nicht im Rahmen dieser Arbeit durchgeführt und ist in [NBGK01] beschrieben.

Um eine universelle Integration von CAN-Geräten zu ermöglichen, wurde die Java CAN API (siehe Kapitel 10) entwickelt, die eine objektorientierte Programmierschnittstelle für CAN-Kommunikation in Java realisiert. Einzelne

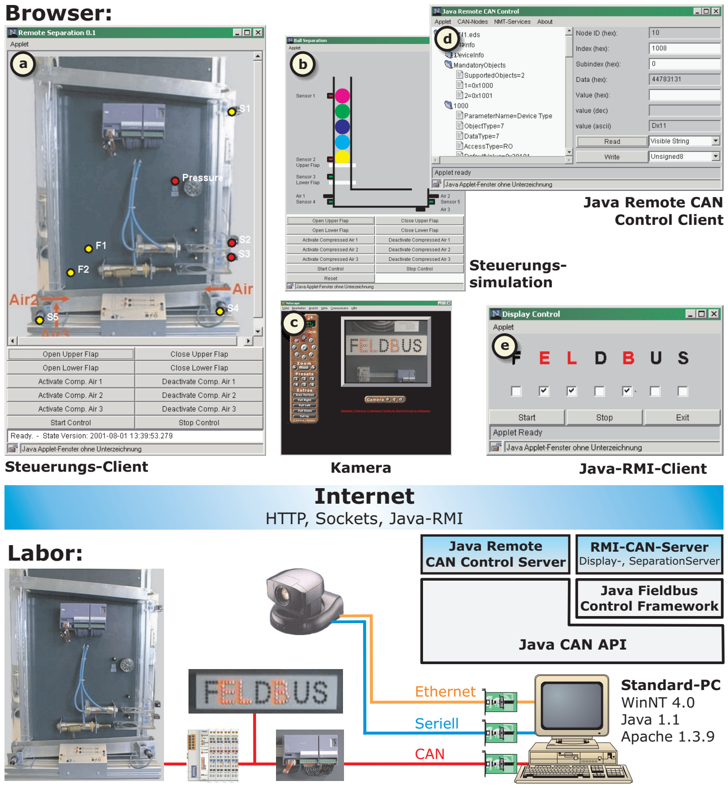

Abbildung 9.2: Integration von industriellen Geräten in das World Wide Web. a) Fernsteuerung einer Werkstückvereinzelung b) Simulation einer Werkstückvereinzelung c) Internet-Kamera d) Generische Integration von CANopen-Geräten e) Integration einer Leuchtschrift

Kommunikationsobjekte unterschiedlicher CAN-Protokollschichten können direkt instanziiert, parametriert und abgeschickt werden. Ebenso kann das asynchrone auftauchen bestimmter CAN-Kommunikationsobjekte auf einfache Weise überwacht werden.

Aufbauend auf dieser API wurde das *Java Fieldbus-based Control Framework* (*JFCF*) realisiert, das die objektorientierte Programmierung von nebenläufigen Steuerungsalgorithmen unterstützt (siehe Abschnitt 11). Es besteht aus einer Abstraktionshierarchie für unterschiedliche industrielle Geräte und diversen Konzepten zur Verwaltung und Synchronisation nebenläufiger ausführbarer Einheiten. Steuerungslogik kann über generische Schnittstellen weitgehend hardwareunabhängig entwickelt und einzelnen ausführbaren Einheiten dynamisch zugeordnet werden. Die implementierten Prototypen wurden für den CAN-Feldbus entwickelt und verwenden eine Busanbindung über die Java CAN API.

Aufbauend auf der Java CAN API und z.T. auf *JFCF* (siehe Abbildung 9.2) wurden drei Java-Server-Applikationen entwickelt, die den entfernten Zugriff auf die Geräte und Anlagen des Labors ermöglichen:

1. *Java Remote CAN Control Server* (*JRCC*-Server)
2. *DisplayServer*
3. *SeparationServer*

Das *JRCC*-System erlaubt den generischen, geräteprofilbasierten Zugriff auf beliebige CANopen-Geräte via Internet. Hierbei können sämtliche Zustandsinformationen der entfernten CANopen-Knoten individuell gelesen und soweit erlaubt auch geschrieben werden. Zusätzlich bietet das System Zugriff auf CANopen-spezifische Managementfunktionalität. Da dieser Ansatz keine Konzepte der *JFCF* erfordert, setzt der *JRCC*-Server direkt auf der Java CAN API auf. Der *DisplayServer* ermöglicht Java-RMI basierten Zugriff auf eine Leuchtschrift, die über ein digitales CANopen-E/A-Modul angesteuert wird. Dieser Server zusammen mit dem entsprechenden Java-Applet-Client (vgl. Abbildung 9.2e) stellen letztendlich eine Spezialisierung des generischen *JRCC*-Systems auf einen bestimmten Anwendungskontext dar. Dieser Server verwendet Teile der *JFCF*-Gerätehierarchie zur Kapselung des E/A-Moduls, wobei der Durchgriff auf die Hardware über die Java CAN API erfolgt.

Ein Nachbau der bereits in Teil 1 vorgestellten Werkstückvereinzelung (siehe Abschnitt 5.6) ist über den *SeparationServer* zugänglich. Ein Benutzer kann nicht nur interaktiv auf die einzelnen Anlagenkomponenten zugreifen (z.B. Öffnen/Schließen einzelner Klappen), sondern auch einen kompletten Vereinzelungsalgorithmus auswählen und ausführen lassen (vgl. Abbildung 9.2a). Zur Implementierung und Verwaltung der Steuerungsalgorithmen wird das *JFCF*-Framework eingesetzt.

Wie in Abbildung 9.2 angedeutet, wurden die Java-RMI-basierten Server-Komponenten (*DisplayServer* und *SeparationServer*) zu einem Server-Prozess (*RMI-CAN-Server*) zusammengefasst, der die einzelnen Implementierungen der von *java.rmi.Remote* abgeleiteten Schnittstellen bei der RMI-Registry anmeldet und so einem Internet-basierten Zugriff zugänglich macht. Beide Server nutzen das *ServerArbitration*-Framework (siehe Abschnitt 6.1.1) zur Realisierung der Benutzersynchronisation.

Der Server des *JRCC* Systems kommuniziert direkt über Java-Sockets und ist als eigenständiger Prozess realisiert.

9.4.2 Realisierte Übungen

Basierend auf den im vorigen Abschnitt vorgestellten Softwarelösungen wurden folgende drei Übungsaufgaben realisiert (siehe Kapitel 12):

1. CANopen-Geräteprofile
2. Java-RMI-basierter Zugriff auf eine Leuchtschrift
3. Steuerung einer Werkstückvereinzelungsanlage

Die erste Aufgabe nutzt das *JRCC*-System zum generischen Zugriff auf CANopen-Geräte via Internet. Die Studenten analysieren und verändern den Zustand von entfernten CANopen-Geräten und experimentieren mit dem CANopen-Knotenzustandsdiagramm. Geführt werden die Studenten über ein Online-*HTML*-Formular, in das die einzelnen Ergebnisse und Erkenntnisse eingetragen werden. Die Studenten erhalten hierbei Einblick in das Geräteprofilkonzept, das CANopen-Protokoll und das *Electronic Data Sheet* Format zur externen Beschreibung von CANopen-Geräteprofilen.

Im Rahmen der Zweiten Aufgabe implementieren die Studenten einen Java-RMI-Client gegen die Schnittstelle des *DisplayServer*s. Die entwickelten Lösungen unterstützen dann den Zugriff auf die entfernte Leuchtschrift und erlauben den Studenten jeweils einzelne Buchstaben an- oder auszuschalten. Das Ergebnis kann über die Web-Kamera beobachtet werden. Die Studenten erhalten hierbei Einblick in eine exemplarische, Java-basierte Architektur zur Realisierung von Teleservice- und Gerätefernsteuerungssystemen.

Aufgabe 3 vermittelt erste Erfahrungen im Bereich der Erstellung objektorientierter Gerätesteuerungen. Die Studenten entwickeln hierbei einen Steuerungsalgorithmus für die Werkstückvereinzelung. Dieser Steuerungsalgorithmus wird als Java-Objekt zunächst innerhalb eines entsprechenden Simulations-Applets (siehe Abbildung 9.2b) entwickelt und kann dann in einem zweiten Schritt auf der realen Anlage über den Steuerungs-Client der Werkstückvereinzelung (siehe Abbildung 9.2a) getestet werden.

Kapitel 10

Die Java CAN API

10.1 Einleitung

Die Java CAN API [BN00] realisiert eine objektorientierte Programmierschnittstelle (API) für den CAN-Feldbus in Java. Einzelne Kommunikationsobjekte unterschiedlicher CAN-Protokollschichten können direkt instanziiert, parametriert und abgeschickt werden. Ebenso kann das asynchrone Auftauchen bestimmter CAN-Nachrichten auf einfache Weise überwacht werden. Die Java CAN API unterstützt damit die schnelle Entwicklung von Java-Applikationen, die mit industriellen CAN-Geräte kommunizieren können.

Ein objektorientierter Ansatz eignet sich zur Kapselung von CAN-Kommunikation besonders gut, da das weit verbreitete Protokoll CANopen ein nach objektorientierten Ideen entwickeltes Schicht-7-Protokoll für den CAN-Feldbus darstellt.

10.1.1 Motivation

Die momentan für den CAN-Feldbus erhältlichen Programmierschnittstellen (z.B. der Firma *Vector-Informatik* (s.u.), *IXXAT* (www.ixxat.com) oder *Port* (www.port.de)) umfassen eine ganze Reihe an höheren Programmiersprachen (C/C++, *Borland Delfi* und *Microsoft Visual Basic*), wobei die Programmiersprache Java und objektorientierte Konzepte nicht unterstützt werden. Objektorientierte Software-Technologien und insbesondere Java werden jedoch zunehmend populärer im Bereich industrieller Steuerungen (vgl. [Lum99]).

In diesem Zusammenhang sind die folgenden Eigenschaften von Java besonders relevant:

1. Plattformunabhängigkeit
2. Automatisches Speichermanagement
3. Java-Bibliotheken

Plattformunabhängigkeit ist insbesondere im Bereich eingebetteter Systeme wertvoll. Der Portierungsbedarf für entwickelte Softwarelösungen ist in diesem Bereich oft erheblich, da eine Vielzahl unterschiedlicher Prozessoren mit unterschiedlichen Eigenschaften eingesetzt werden. Das Konzept der virtuellen Java-Maschine erlaubt hier eine weitaus einfachere Portierung von Software als dies mit C/C++ -basierten Ansätzen erreicht werden kann.

Da Software-Lösungen auf eingebetteten Systemen i.d.R. sehr lange (oft Monate und Jahre) in Betrieb und Speicherressourcen äußerst knapp bemessen sind, führen bereits kleinste Speicherlöcher (*Memory Leaks*) unweigerlich zu Systemabstürzen. Zusätzlich sind auf eingebetteten Systemen oft nur rudimentäre Debug- und Profiling-Werkzeuge vorhanden. Das automatische Speichermanagement von Java erleichtert und beschleunigt hier die Erstellung stabiler Software.

Die Vielzahl an kostenfrei verfügbaren Java-Bibliotheken, z.B. aus den Bereichen Netzwerkkommunikation, GUI-Programmierung oder Kryptographie erleichtert die Integration von erstellten Lösungen in die Informationsinfrastruktur eines Unternehmens.

Die im Kontext der Gerätesteuerungstechnik entscheidenden Nachteile von Java, kein Zugriff auf Hardware-Ressourcen und keine Unterstützung von Realzeit-Anforderungen, werden momentan durch unterschiedliche Ansätze beseitigt oder abgeschwächt. Insbesondere seien hier Implementierungen der virtuellen Maschine in Hardware (z.B. [Har01]) und die Realzeit-Spezifikation für Java und verwandte Ansätze genannt [BG00a, BHK00, KGLS97].

10.1.2 *Controller Area Network* (CAN)

Der CAN-Feldbus ist ein bis zu OSI-Schicht 2 spezifiziertes Protokoll [Rob91, Int93] zur Kommunikation über kurze Distanzen im Umfeld industrieller Steuerungen mit Realzeitanforderungen. Die wichtigsten Eigenschaften des CAN sind:

- Nachrichtenorientierung
- Jeder Knoten darf aktiv Nachrichten senden und empfangen.
- 8 Datenbytes und 11 bzw. 29 (für *Extended* CAN) Bit Nachrichten-ID
- Die ID einer Nachricht impliziert ihre Priorität

- CSMA/CA Buszuteilungsverfahren basierend auf Nachrichtenprioritäten

CAN ist im Vergleich zu anderen Feldbustechnologien, wie z.B. Profibus, meist kostengünstiger, aber auch beschränkter im Datendurchsatz, in der erlaubten Knotenanzahl und der maximal erlaubten räumlichen Ausdehnung des Busses. Laut der nicht kommerziell ausgerichteten *CAN in Automation* (CiA) Organisation[1], der mehr als 300 Firmen weltweit angehören, waren Ende 1999 bereits über 150 Mio. CAN-Knoten installiert. CAN ist vor allem im Bereich der KFZ-Technik, aber auch in der Medizintechnik und Automatisierung stark verbreitet.

10.1.2.1 Das CANopen-Protokoll

Es haben sich diverse Protokolle für Kommunikation auf OSI-Schicht 7 für den CAN etabliert, die alle direkt auf Schicht 2 aufsetzen. Die verbreitetsten Vertreter sind das *DeviceNet*-Protokoll der Firma Rockwell, das *CANKingdom*-Protokoll der Firma Kvaser und das offene CANopen-Protokoll [Can96b]. CANopen hat den Vorteil, dass es sich als offener Standard im CAN-Bereich stark durchgesetzt hat und die Spezifikationsdokumente frei zugänglich sind. Eine detaillierte Beschreibung des CANopen-Protokolls findet sich beispielsweise in [FB99].

Die wichtigsten Eigenschaften von CANopen sind:

- Implementierung eines objektorientierten Geräteprofils
- Die ID einer Nachricht impliziert zusätzlich die CANopen-Geräte-ID des Empfängers und den Nachrichtentyp
- Unterschiedliche Nachrichtentypen, die als Kommunikationsobjekte verstanden werden können:
  - Parametrierbare Nachrichten zum Zugriff auf beliebige Geräteparameter (*Service Data Objects*)
  - Vorkonfigurierte Nachrichten zum schnellen Zugriff auf bestimmte Geräteparameter (*Process Data Objects*)
  - Fehlernachrichten, Notfalltelegramme, Node-Guarding-Nachrichten
- Definition eines Knotenzustandsdiagramms
- Unterstützung diverser Datentypen (siehe [Can96c])
- Definition des *Electronic Data Sheet* Formats [Can99] zur externen, textbasierten Spezifikation des Geräteprofils (vgl. Abschnitt 4.7)

[1] http://www.can-cia.com

Das entscheidende Konzept des CANopen Protokolls ist, ähnlich wie bei vielen anderen Feldbussystemen auch, die Definition von parameterbasierten Geräteschnittstellen, die als Objektverzeichnis oder Geräteprofil bezeichnet werden. CANopen verwendet zweidimensionale Geräteprofile und adressiert einzelne Einträge/Parameter über sog. *Index*- und *Subindex*-Werte. Der CANopen-Parameter aus Abbildung 4.9 auf Seite 64 adressiert beispielsweise die Gerätetypinformation eines E/A-Moduls unter Index=1000 und Subindex=0. Alle verfügbaren Informationen eines Gerätes werden konzeptionell in ein derartiges Profil abgebildet, aus dem sie durch Angabe von Index und Subindex ausgelesen werden können. Ebenso kann durch Schreiben bestimmter Werte in das Profil spezifisches Verhalten des Gerätes hervorgerufen werden.

Zum Zugriff auf CANopen-Geräteprofile können zwei unterschiedliche Arten von Kommunikationsobjekten verwendet werden, die als *Service Data Object* und *Process Data Object* bezeichnet werden. Diese Kommunikationsobjekte werden jeweils durch ein oder mehrere CAN-Nachrichten auf Schicht 2 realisiert, und gehorchen entsprechenden CANopen-Teilprotokollen.

Service Data Objects Ein *Service Data Object* (SDO) ist auf logischer Ebene ein einzelnes Kommunikationsobjekt zum Lesen oder Schreiben eines bestimmten Eintrags eines CANopen-Geräteprofils. Das Objekt wird parametriert durch Angabe von *Index* und *Subindex* sowie bei schreibendem Zugriff mit den zu schreibenden Werten. Wie bereits erwähnt, ist die Kodierung diverser Datentypen in [Can96c] definiert. Da auch Datentypen großer und variabler Länge unterstützt werden und jede CAN-Nachricht maximal 8 Byte Daten tragen kann, ist die Anzahl an CAN-Nachrichten, die zum Lesen bzw. Schreiben eines Wertes nötig sind, ebenfalls variabel. Zur Abwicklung eines SDO-Datentransfers wurden deshalb diverse Nachrichtenformate definiert, die eine sichere, stückweise iterative Auslieferung eines Wertes erlauben. Hierbei wird jede gesendete Nachricht vom Empfänger bestätigt.

SDO-Nachrichten werden i.d.R. zur Konfiguration eines Gerätes in einer Initialisierungsphase verwendet und nicht zum Datenaustausch unter Realzeit-Anforderungen während des Betriebs. Die Prioritäten von SDO-Nachrichten sind dementsprechend relativ niedrig. Insbesondere können SDO-Nachrichten dazu verwendet werden, die PDO-Konfiguration (siehe nächster Paragraph) eines Gerätes festzulegen.

Process Data Objects Ein *Process Data Object* (PDO) ist auf logischer Ebene ein Daten-Container mit vorkonfiguriertem Ziel und Inhalt zum schnellen Transport bestimmter Parameterwerte. Das Ziel und der Inhalt einer PDO-Nachricht wird durch spezielle Einträge des Geräteprofils spezifiziert und ist nicht, wie bei SDO-Nachrichten, Teil der Nachricht selbst. Durch diesen Ansatz ist es möglich, die vollen 8 Datenbytes einer CAN-Nachricht für den Transport von Daten zu verwenden. Implementiert werden PDO-Nachrichten durch jeweils eine einzelne entsprechende CAN-Nachricht, die aufgrund ihrer Nachrichten-ID als PDO-Nachricht identifiziert wird. PDO-Nachrichten werden grundsätzlich vom Empfänger nicht bestätigt.

PDO-Nachrichten werden i.d.R. während des Betriebs eines Gerätes verwendet, um vorkonfigurierte Datenpakete mit hoher Priorität zu versenden. Typische PDO-Nachrichten sind beispielsweise zyklisch verschickte Positionsmeldungen von Schrittmotoren oder asynchrone Benachrichtigungen aufgrund von bestimmten Ereignissen. Innerhalb einer PDO-Konfiguration wird daher neben der Information, welche Parameterwerte jeweils in den Daten-Container gepackt werden sollen, auch festgehalten, ob das PDO zyklisch oder asynchron als Reaktion auf bestimmte Ereignisse ausgelöst werden soll.

Es werden zwei Arten von PDO-Nachrichten unterschieden: *Receive PDOs* werden eingesetzt, um das Objektverzeichnis eines Gerätes zu verändern. PDO-Nachrichten, die ein Gerät von sich aus verschickt, z.B. aufgrund eines bestimmten Ereignisses, werden hingegen als *Transmit PDOs* bezeichnet.

Weitere Nachrichtentypen Zusätzlich zu SDO- und PDO-Nachrichten definiert das CANopen-Protokoll die folgenden Kommunikationsobjekte:

- SYNC
- Time-Stamp
- Emergency
- Node-Guarding

Die SYNC-Nachricht ist eine CAN-Nachricht, die zyklisch durch einen SYNC-Master verschickt und zur Synchronisation der CAN-Kommunikation verwendet wird. Insbesondere können PDO-Nachrichten so konfiguriert werden, dass sie im Anschluss an jede n-te erhaltene SYNC-Nachricht gesendet werden.

Time-Stamp-Nachrichten können verwendet werden, um eine einheitliche Zeit-Definition innerhalb eines CAN zu realisieren. Emergency-Nachrichten sind asynchrone Nachrichten, die von einzelnen Konten im Fehlerfall gesendet werden können. Um die Verfügbarkeit einzelner Knoten überwachen zu können, kann der Status dieser Knoten durch spezielle Node-Guarding-Nachrichten zyklisch überwacht werden.

Network Management Services Das CANopen-Protokoll definiert ein Knotenzustandsdiagramm mit insgesamt 8 Knotenzuständen und diversen Zustandsübergängen. Nach erfolgtem Einschalten eines CANopen-Gerätes befindet sich das Gerät prinzipiell zu jedem Zeitpunkt in genau einem dieser 8 Zustände. In einer Boot-Up-Sequenz durchläuft ein Knoten zunächst diverse Zustände, in denen insbesondere die Konfiguration des Knotens (z.B. die PDO-Konfiguration) erfolgt. Nach erfolgtem Boot-Up kann sich ein Knoten in diversen Zuständen befinden, in denen er beispielsweise PDO-Nachrichten akzeptiert oder ignoriert.

Die einzelnen Übergänge in diesem Zustandsdiagramm werden durch entsprechende *Network Management Services* (NMT) Nachrichten zugänglich gemacht. Implementiert sind diese NMT-Nachrichten durch jeweils genau eine CAN-Nachricht, die den jeweiligen Zustandsübergang identifiziert. Eine NMT-Nachricht kann immer entweder an einen bestimmten Knoten oder an alle gleichzeitig gesendet werden.

Zusätzlich können über weitere CANopen-Services die Verteilung der Nachrichten-IDs und damit auch die Prioritäten der entsprechenden Nachrichten flexibel angepasst werden.

10.2 Überblick

Abbildung 10.1 gibt einen Überblick über die Schichtenarchitektur der Java CAN API. Die unterste Schicht besteht aus der nativen CAN-Treibersoftware, die normalerweise zusammen mit der CAN-Hardware ausgeliefert wird. Die realisierte Implementierung verwendet an dieser Stelle den universellen WinNT-Treiber *vcand32.DLL*[2] der Firma *Vector-Informatik* für unterschiedliche CAN-Schnittstellenkarten (z.B. CANcardX, CANpari und CAN-AC2). Dieser Treiber wird durch die entwickelte, native Bibliothek *JCan.DLL* gekapselt.

Die JCan.DLL unterstützt zum Einen einen feingranularen CAN-Nachrichtenfilter, zum Anderen exportiert sie eine abstrakte CAN-Kommunikationsschnittstelle über das Java Native Interface (JNI) (vgl. [Lia99]). Die Java-Schichten *CanProtocol* und *JCan* der Java CAN API setzen auf dieser Schnittstelle auf. Das *JCan*-Paket realisiert zusammen mit der JCan.DLL die Kommunikationsinfrastruktur und unterstützt sowohl CAN-Nachrichtenpolling als auch CAN-getriebene, asynchrone Benachrichtigung über einen Event-Mechanismus gemäß dem Observer-Pattern [GHJV95].

Das *CanProtocol*-Paket unterstützt objektorientierte CAN-Kommunikation sowohl auf Schicht 2 als auch über das CANopen-Protokoll auf Schicht 7. Es existieren

[2] Erhältlich über anonymes FTP unter ftp://ftp.vector-informatik.de/pub/support/canlib31.zip

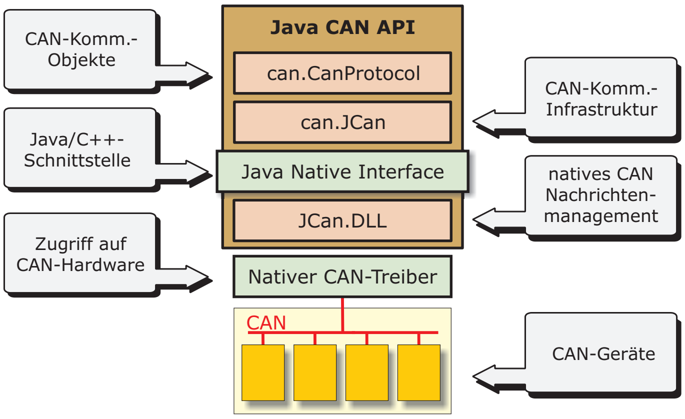

Abbildung 10.1: Schichtenarchitektur der Java CAN API

u.a. Java-Klassen und -Schnittstellen zur Kapselung von CAN-Nachrichten auf Schicht 2 sowie SDO- und PDO-Kommunikationsobjekten und der NMT-Services auf Schicht 7.

10.3 Architektur

In diesem Abschnitt werden die in Abbildung 10.1 aufgeführten Schichten der Java CAN API im Detail vorgestellt.

10.3.1 Die C++-Schicht

Um die Schnittstelle des nativen CAN-Treibers von Java aus ansprechen zu können, muss diese als dynamisch bindbare Bibliothek[3] auf Systemebene verfügbar sein und die Bezeichnung der exportierten Methoden bestimmten Restriktionen genügen. Diese Bibliothek wird dann während der Ausführung der virtuellen Java-Maschine durch einen sog. *Static Initializer* dynamisch zu dieser hinzugebunden. Nach erfolgter Bindung können dann Java Methodenaufrufe innerhalb der virtuellen Maschine über JNI auf die entsprechenden exportierten Funktionen der nativen Bibliothek abgebildet und so aufgerufen werden.

Eine Möglichkeit zur Integration der *vcand32*-Treiber-DLL in eine Java API wäre

[3]Auf Windows-Plattformen heißen diese Bibliotheken *Dynamic Link Libraries* (DLL), auf Unix-Plattformen *Shared Objects* (so).

gewesen, die einzelnen Funktionen dieser Bibliothek direkt über JNI zu exportieren. Dieser Ansatz hätte zur Folge, dass jeder Treiber-Aufruf über das JNI abgearbeitet werden müsste, was sehr zeitaufwendig sein kann. Dies ist insbesondere deshalb problematisch, weil die *vcand32.DLL* selbst keine asynchrone Benachrichtigung unterstützt. Um ein Nachrichten-Polling nicht über das JNI durchführen zu müssen, implementiert die JCan.DLL ein eigenständiges Nachrichtenmanagement, das die gezielte, asynchrone Benachrichtigung von Java-Objekten bei Eintreffen bestimmter CAN-Nachrichten erlaubt.

Die asynchrone Benachrichtigung kann in Verbindung mit speziellen Nachrichtenfiltern für jede Nachrichten-ID separat konfiguriert werden. Dieser Ansatz ermöglicht es, dass sich Java-Objekte nur vom Eintreffen bestimmter Nachrichten benachrichtigen lassen können und so beispielsweise hochfrequente aber uninteressante Nachrichten ausgeblendet werden. Da die Nachrichtenfilterung ebenfalls komplett innerhalb der JCan.DLL implementiert ist, kann diese sehr effizient ausgeführt werden.

Als Filterkriterium wird die ID der CAN-Nachrichten verwendet, die, wie in Abschnitt 10.1.2 bereits ausgeführt, sowohl Informationen über die Art der Nachricht (PDO, SDO, SYNC ...) als auch über den Empfänger der Nachricht enthalten. Dieses Verfahren ermöglicht es beispielsweise, speziell eine SDO-Antwort-Nachricht eines bestimmten Knotens aus dem sonstigen CAN-Nachrichtenverkehr herauszufiltern.

10.3.1.1 Implementierung

Die JCan.DLL wurde mit Hilfe des Microsoft Visual C++ Compilers und der Standard Template Library (STL) [MS96] entwickelt. Das Polling des CAN übernimmt indirekt der *MessageManager*-Thread (siehe Abbildung 10.2), der in einer Endlosschleife die Nachrichtenwarteschlange der Treiber-DLL, in der sämtliche CAN-Nachrichten abgelegt werden, ausliest. War der Zugriff auf die Treiberwarteschlange erfolgreich (d.h. seit dem letzten Zugriff sind neue CAN-Nachrichten auf dem Bus entdeckt worden), so legt der *MessageManager*-Thread die ausgelesene Nachricht in internen Warteschlangen ab. Für jede Nachrichten-ID kann eine separate Nachrichtenwarteschlange in Form eines STL-Queue-Objekts verwaltet werden, die dann nur CAN-Nachrichten mit dieser ID beinhaltet.

Zur Verwaltung dieser Warteschlangen können für jede der 2^{11} (bzw. 2^{29} für *Extended*-CAN) möglichen Nachrichten-IDs die Parameter *subscription*, *notification* und *qLength* angegeben werden (siehe Abbildung 10.2). Der *subscription*-Parameter gibt an, ob überhaupt eine Schlange für die entsprechende ID angelegt werden soll. Da in den meisten CAN-Anlagen überhaupt nur eine stark begrenzte Menge von Nachrichten-IDs auftreten, werden also nicht bis zu 2^{29} leere Queue-

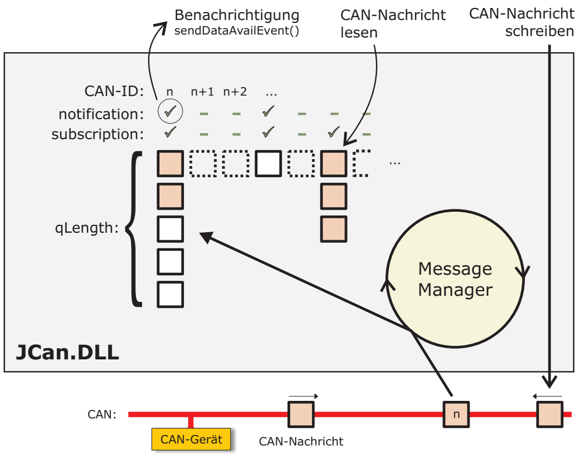

Abbildung 10.2: CAN-Nachrichtenmanagement in der JCan.DLL

Objekte angelegt, sondern nur die tatsächlich benötigte Anzahl. Verwaltet wird die Gesamtheit der Queue-Objekte in einem assoziativen *Map*-Objekt der STL, das einen effizienten Zugriff auf die einzelnen Warteschlangen erlaubt.

Der *notification*-Parameter gibt für eine Nachrichten-ID an, ob beim Eintreffen entsprechender Nachrichten eine asynchrone Benachrichtigung durchzuführen ist. Ist dieser Parameter für eine Nachrichten-ID gesetzt, so wird bei jedem Eintreffen dieser Nachricht die Java-Methode *sendDataAvailEvent()* (siehe Abschnitt 10.3.2.1) desjenigen Java-Objekts, das die JCan.DLL gebunden hat, über das JNI aufgerufen. Als Parameter wird jeweils die ID der eingetroffenen Nachricht übergeben. Das Java-Objekt ist dann selbst dafür verantwortlich, angemessen zu reagieren, z.B. durch Lesen aus der entsprechenden Warteschlange oder durch Benachrichtigung weiterer Java-Objekte.

Der *qLength*-Parameter gibt für jede Warteschlange die maximale Ausdehnung an, bevor die jeweils ältesten Nachrichten verworfen werden. Insbesondere kann die Länge 1 gewählt werden, um immer jeweils nur die aktuellste Nachricht einer bestimmten ID vorzuhalten.

Wie bereits erwähnt, werden die einzelnen Warteschlangen durch den *MessageManager*-Thread gefüllt. Geleert werden sie, außer bei einem Überlauf, durch Auslesen der Nachrichten über entsprechende Methoden (siehe Abschnitt 10.3.2.1). Die JCan.DLL bietet weiterhin eine Funktion zum Generieren bestimmter Nachrichten auf dem CAN, die im Wesentlichen einfach auf die entsprechende Treiberfunktion durchgreift.

Um eine korrekte Verwaltung der Nachrichten auch bei nebenläufigem Zugriff auf die einzelnen Warteschlangen zu gewährleisten, wird das Konzept der *Critical Sections* des Windows-Betriebssystem zur Synchronisation der einzelnen Zugriffe eingesetzt.

10.3.2 Die Java-Schicht

10.3.2.1 Das *JCan*-Paket

Das *JCan*-Paket realisiert eine objektorientierte CAN-Kommunikationsinfrastruktur. Die zentrale Klasse dieses Paketes und das Pendant zur JCan.DLL ist die Klasse *CanPort* (siehe Abbildung 10.3). Diese Klasse koordiniert sämtliche Zugriffe auf den CAN und propagiert eintreffende Nachrichten an registrierte *CanPortEventListener*-Objekte. Innerhalb einer Java CAN API-basierten Anwendung existiert i.d.R. pro CAN-Kanal genau eine *CanPort*-Instanz, deren Erzeugung über das Singleton-Pattern implementiert werden kann.

Die Methoden *subscribeID()* bzw. *unsubscribeID()* konfigurieren die Nachrichtenwarteschlangen der JCan.DLL wie im vorigen Abschnitt beschrieben. Die *write()*-Methode schreibt eine CAN-Nachricht auf den Bus, wobei die Nachrichten-ID als Integer-Wert und die Datenbytes als Array vom Typ *byte* übergeben werden. Die Methode *read()* liest eine Nachricht aus der durch den Parameter *id* identifizierten Nachrichtenwarteschlange der JCan.DLL. Die Datenbytes der gelesenen Nachricht werden in das Byte-Array *b* kopiert und aus der Warteschlange entfernt.

Um ein aktives Polling des CAN von Java-Seite aus zu vermeiden, unterstützt die Java CAN API asynchrone Benachrichtigung gemäß dem Observer-Pattern. Wie bereits erwähnt, können einzelne Nachrichtenwarteschlangen über die Methode *subscribeID()* und den Parameter *notify* (vgl. Abbildung 10.3) so konfiguriert werden, dass beim Eintreffen entsprechender CAN-Nachrichten die Callback-Methode *sendDataAvailEvent()* der *CanPort*-Klasse über JNI aufgerufen wird. Die Implementierung dieser Methode benachrichtigt daraufhin sämtliche Objekte, die sich über die Methode *addEventListener()* hierfür registriert haben. Die Benachrichtigung der Objekte erfolgt jeweils über die Methode *canEvent()* der *CanPortEventListener*-Schnittstelle (siehe Abbildung 10.3). Diese Schnittstelle muss daher von jedem Objekt implementiert werden, das sich asynchron über das Eintreffen bestimmter Nachrichten informieren lassen möchte. Die Referenz auf die jeweiligen Implementierungen dieser Schnittstelle wird der *CanPort*-Klasse als Parameter innerhalb der *addEventListener()*-Methode übergeben.

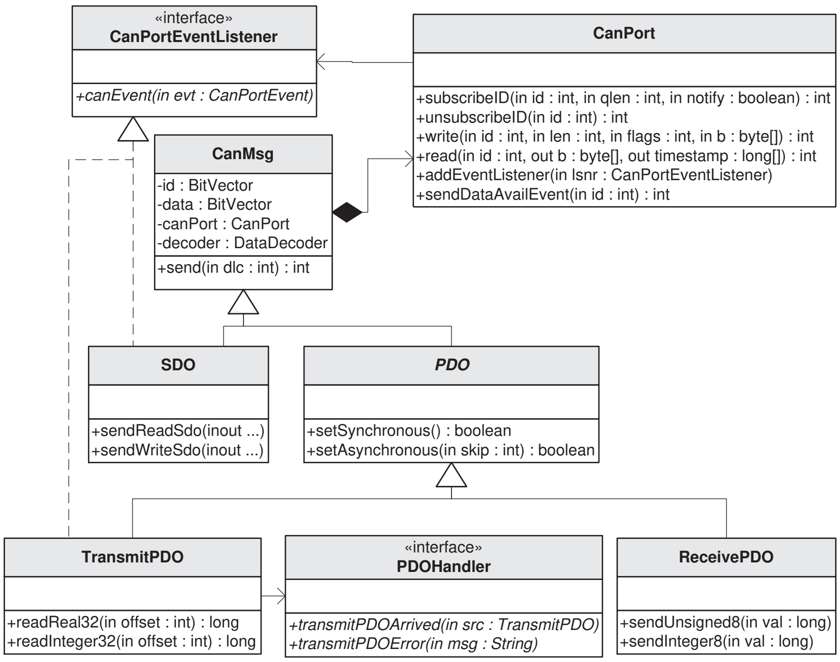

Abbildung 10.3: Ausschnitt aus der Klassenhierarchie der Java CAN API

10.3.2.2 Das *CanProtocol*-Paket

Dieses Paket kapselt sowohl generische CAN-Nachrichten auf Schicht 2 als auch CANopen-Kommunikationsobjekte. Die Philosophie dieses Paketes ist, dass die unterschiedlichen Kommunikationsobjekte die aktiven Elemente darstellen und als eigenständige Klassenhierarchie implementiert sind. CAN-Kommunikation innerhalb der Java CAN API läuft daher wie folgt ab:

- Schreibender Zugriff

  1. Instanziierung des entsprechenden Kommunikationsobjektes (falls nötig)
  2. Parametrierung des Kommunikationsobjektes
  3. Das Kommunikationsobjekt generiert und überwacht die entsprechenden CAN-Nachrichten auf dem Bus

- Lesender Zugriff
    1. Instanziierung des entsprechenden Kommunikationsobjektes (falls nötig)
    2. Das Kommunikationsobjekt generiert und überwacht die entsprechenden CAN-Nachrichten auf dem Bus, wobei die Informationen der Geräteantwortnachrichten analysiert und in entsprechenden Feldern des Kommunikationsobjekts abgelegt werden
    3. Zugriff auf die Informationen der Geräteantwort über objektspezifische Zugriffsmethoden

Dieser Ansatz unterscheidet die Java CAN API, abgesehen von der Implementierung in Java, grundlegend von den erhältlichen CAN/CANopen-Bibliotheken und erlaubt einen besonders bequemen Zugriff auf CANopen-Geräte. Abbildung 10.3 zeigt einen Ausschnitt aus der Klassenhierarchie der Kommunikationsobjekte, deren Basis die Klasse *CanMsg* darstellt. Ein *CanMsg*-Objekt kapselt eine CAN-Nachricht auf Schicht 2 und kann damit als Basis für alle spezialisierteren Schicht-7-Kommunikationsobjekte verwendet werden. Die Datenfelder *id* und *data* beinhalten die Nachrichten-ID bzw. die Datenbytes der Nachricht und sind jeweils über die Klasse *BitVector* implementiert. Entscheidend ist, dass jede *CanMsg*-Instanz eine Referenz auf ein *CanPort*-Objekt hat. Durch Aufruf der Methode *send()* schreibt sich ein *CanMsg*-Objekt auf den Bus, indem es die *write()*-Methode der assoziierten *CanPort*-Instanz aufruft.

Die Klassen *SDO* bzw. *PDO* sind direkt von *CanMsg* abgeleitet (vgl. Abbildung 10.3) und kapseln CANopen SDO- bzw. PDO-Kommunikationsobjekte.

SDO-Kommunikation Die Klasse *SDO* stellt mit den Methoden *sendReadSdo()* und *sendWriteSdo()* Methoden zum Lesen und Schreiben einzelner Einträge aus beliebigen CANopen-Geräteprofilen zur Verfügung. Parametriert werden diese Methoden insbesondere durch die Angabe der *Index*- und *Subindex*-Werte und den Datentyp des zu lesenden/schreibenden Geräteparameters Die Abarbeitung beispielsweise der *sendReadSdo()*-Methode besteht aus folgenden Schritten:

1. Parametrierung der initialen CAN-Nachricht zum Einleiten des SDO-Datentransfers

2. Registrierung für die zu erwartende Antwortnachricht bei der *CanPort*-Instanz

3. Aufruf der *send()*-Methode

4. Analyse der Geräteantwort in der Implementierung der *canEvent()* Methode, die durch die *CanPort*-Instanz aufgerufen wird. Diese Antwort enthält eine Größenangabe des zu transferierenden Wertes

5. Generierung weiterer Nachrichten zum stückweise iterativen Datentransfer nach obigem Schema, bis der Wert vollständig geladen ist

6. Dekodierung der Antwort gemäß [Can96c] mit Hilfe des *DataDecoder*-Objekts der *CanMsg*-Klasse

Die Abarbeitung der *sendWriteSdo()*-Methode läuft weitestgehend analog.

PDO-Kommunikation Die Klasse *PDO* ist die Basisklasse für die zwei unterschiedlichen PDO-Spezialisierungen: *TransmitPDO* zum Abhören von asynchron gesendeten PDOs bestimmter Geräte und *ReceivePDO* zum Senden von PDOs an bestimmte Geräte (vgl. Abbildung 10.3). Die Basisklasse *PDO* kapselt erwartungsgemäß die Eigenschaften, die für beide Arten relevant sind, wie z.B. die Auslösungskonfiguration (synchron, asynchron).

Die *ReceivePDO*-Klasse erlaubt das Verpacken bestimmter Werte in die PDO-Nachricht, wie sie auf Geräte-Seite durch entsprechende Einträge im Geräteprofil spezifiziert ist, und das Verschicken der Nachricht. Die CAN-spezifische Kodierung wird wieder durch die Klasse *DataDecoder* erledigt. Eine derartige PDO-Nachricht, gepackt mit einem Unsigned8 [Can96c] Wert, kann beispielsweise dazu verwendet werden, die Ausgangsbelegung eines achtkanaligen digitalen E/A-Moduls zu setzen, sofern das E/A-Modul entsprechend konfiguriert ist.

Die *TransmitPDO*-Klasse erlaubt das selbstständige Überwachen bestimmter *Transmit PDOs*. Eine Instanz dieser Klasse registriert sich bei der *CanPort*-Instanz für die Nachrichten mit der entsprechenden Nachrichten-ID und überlagert sich selbst bei jedem Callback mit der gelesenen PDO-Nachricht. Zusätzlich informiert das *TransmitPDO*-Objekt bei jeder Überlagerung eine entsprechende *PDOHandler*-Implementierung, die dann in angemessener Weise reagieren kann, z.B. indem sie bestimmte Daten aus der überlagerten Nachricht ausliest.

Typischerweise propagieren E/A-Module eine Änderung ihrer Eingangsbelegung durch asynchrones Versenden eines entsprechenden *Transmit PDOs*. Derartige Zustandsänderungen können bequem durch Erzeugung eines entsprechenden *TransmitPDO*-Objekts und einer entsprechenden *PDOHandler*-Implementierung überwacht werden.

NMT-Services Die einzelnen *Network Management Services* sind innerhalb der Klasse *NMT* als statische Methoden implementiert [Büh01].

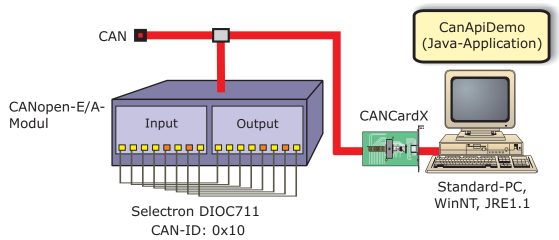

Abbildung 10.4: Hardware-Aufbau der CAN-Demo-Applikation

10.4 Anwendungsbeispiel

Die Java CAN API wurde in einer Reihe von Anwendungen erfolgreich eingesetzt (siehe Abschnitte 5.6, 6.4, 12.1, 12.2 und 12.3). In diesem Abschnitt wird ein einfaches Anwendungsszenario beschrieben, das den praktischen Einsatz der Java CAN API illustriert.

10.4.1 Hardware-Aufbau

Abbildung 10.4 zeigt den Hardware-Aufbau dieses Beispiels. Ein digitales CANopen-E/A-Modul der Firma *Selectron* ist über eine CAN-Schnittstellenkarte der Firma *Vector-Informatik* mit einem Standard-PC verbunden. Als Betriebssystem wird *Windows NT 4.0* und das *JRE 1.1.6* von *Sun microsystems* eingesetzt. Die einzelnen Ausgänge des E/A-Moduls sind über Feedback-Leitungen auf die entsprechenden Eingänge zurückgeführt. Dadurch kann die aktuelle Ausgangsbelegung auch an den Eingängen abgelesen werden. Weiterhin ist das E/A-Modul derart konfiguriert, dass bei jeder Änderung der Eingangsbelegung die neue Eingangsbelegung über ein *Transmit PDO* auf dem CAN publiziert wird.

10.4.2 Java CAN API-Programmierung

Die entwickelte Demo-Applikation *CanApiDemo* erfüllt folgende Aufgaben:

1. Es sollen alle entdeckten CAN-Nachrichten auf dem Bildschirm ausgegeben werden
2. Das E/A-Modul soll in den Zustand *operational* versetzt werden
3. Es soll der Gerätename mit Hilfe des SDO-Protokolls ausgelesen werden

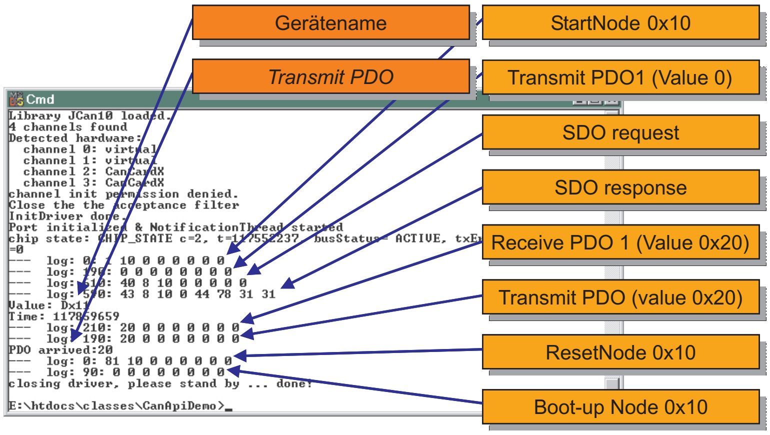

Abbildung 10.5: Bildschirmausgabe der *CanApiDemo*-Applikation

4. Alle *Transmit PDOs* des E/A-Moduls sollen erkannt und ausgegeben werden

5. Es soll die Belegung der Ausgänge verändert werden (dies provoziert ein *Transmit PDO* aufgrund der Feedback-Leitungen)

Abbildung 10.5 zeigt die Ausgaben von *CanApiDemo* auf der Konsole, Listing 10.1 den entsprechenden Java-Quellcode.

In Zeile 8 wird zunächst eine *CanPort*-Instanz angelegt. Die Parameter geben die Baudrate des CAN, den zu verwendenden CAN-Kanal und das Timer-Intervall, das in dieser Applikation keine Rolle spielt, an. Das in Zeile 9 erzeugte Objekt der Klasse *CanLog* (siehe Listing 10.2) übernimmt die Überwachung des CAN und gibt alle entdeckten Nachrichten auf der Konsole aus. Hierzu registriert sich das *CanLog*-Objekt zunächst als *CanPortEventListener* bei dem *CanPort*-Objekt und abonniert sämtliche Nachrichten mit der Methode *subscribeAllIDs()* (Listing 10.2, Zeile 12-14). Dies bewirkt, dass für jede CAN-Nachricht automatisch die Implementierung der *canEvent()*-Methode (Zeile 17-23) aufgerufen wird. Die *canEvent()*-Methode der *CanLog*-Klasse serialisiert die Nachricht in einen String und gibt diesen zusammen mit einem speziellen Präfix auf der Konsole aus (siehe Abbildung 10.5).

Innerhalb der *CanApiDemo*-Implementierung (siehe Lisiting 10.1) wird als nächstes der CANopen-Knotenzustand des E/A-Moduls auf *operational* gesetzt. Dies geschieht durch Aufruf der Methode *startRemoteNode()* der *NMT*-Klasse mit

```
   public class CanApiDemo {
6    public static void main(String[] args) {
       try {
8        CanPort port = new CanPort(125000, 2, 0);
         CanLog log = new CanLog(port);
10
         NMT.startRemoteNode(port, 0x10);
12
         StringBuffer value = new StringBuffer();
14       long[] result = new long[1];
         long[] timeStamp = new long[1];
16       int[] size = new int[1];

18       SDO sdo = new SDO(port, 0x10);
         sdo.sendReadSdo(0x1008, 0x0, result, value, size,
20         timeStamp, DataTypes.VISIBLE_STRING);
         System.out.print("\nValue:␣" + value);
22       System.out.print("\nTime:␣" + timeStamp[0]);
         sdo.cleanup();
24
         TransmitPDO tpdo = new TransmitPDO(port, 0x10, 1);
26       tpdo.addPDOHandler(new MyHandler());

28       ReceivePDO rpdo = new ReceivePDO(port, 0x10, 1);
         rpdo.sendUnsigned8(0x20);
30       //...
```

Listing 10.1: Die *CanApiDemo*-Klasse

der *CanPort*-Instanz und der CANopen-Geräte-ID des E/A-Moduls als Parameter (Zeile 11).

In den Zeilen 13-16 werden die einzelnen Parameter für den lesenden Zugriff auf einen Geräteprofileintrag erzeugt. Der Parameter *value* erhält eine String-Repräsentation des gelesenen Wertes, *timestamp[0]* den Zeitstempel der letzten CAN-Nachricht, die an dem CAN-Datentransfer beteiligt war, und *size[0]* die Größe des gelesenen Wertes in Bytes. Erzeugt und abgeschickt wird das entsprechende *SDO*-Objekt dann in Zeile 18 und 19. Die Werte $0x1008$ und $0x0$ sind die Index- und Subindex-Adresse des zu lesenden Gerätenamens. Der Datentyp *VISIBLE_STRING* referenziert den entsprechenden CANopen-Datentyp für das *DataDecoder*-Objekt der *SDO*-Instanz. In Zeile 21-23 wird der gelesene Wert auf die Konsole geschrieben (vgl. Abbildung 10.5) und verwendete Ressourcen freigegeben.

Im nächsten Schritt (Zeile 25 u. 26) wird ein *TransmitPDO*-Objekt erzeugt, das eintreffende *Transmit PDOs* des E/A-Moduls erkennt und ebenfalls auf der Konsole ausgibt. Der Parameterwert 1 identifiziert hierbei eine bestimmte PDO-

```java
public class CanLog implements CanPortEventListener {
  private CanPort port;
  private int msgId;
  private byte[] data;
  private long[] timeStamp;

  public CanLog(CanPort port) {
    this.port = port;
    data = new byte[8];
    timeStamp = new long[1];
    try {port.addEventListener(this);}
    catch (TooManyListenersException e) {e.printStackTrace();}
    port.subscribeAllIDs(1, true);
  }

  public void canEvent(CanPortEvent e) {
    msgId = e.getEventSrc();
    try {port.peekValue(msgId, data, timeStamp);}
    catch (CanPortException ex) {ex.printStackTrace();}
    System.out.print("\n---  log: " + Integer.toHexString(msgId) + ": ");
    CanMsg.showMsg(data);
  }
}
```

Listing 10.2: Die *CanLog*-Klasse

```java
public class MyHandler implements PDOHandler{
  public void transmitPDOError(String msg) {
    System.out.print("\nPDO error: " + msg);}

  public void transmitPDOArrived(TransmitPDO pdo) {
    try {System.out.print("\nPDO arrived:" +
      Long.toHexString(pdo.readUnsigned8(0)));}
    catch (CanDataTypeException e) {e.printStackTrace();}
  }
}
```

Listing 10.3: Die *MyHandler*-Klasse

Konfiguration[4]. Als zuständiges *PDOHandler*-Objekt wird ein neu erzeugtes Objekt der Klasse *MyHandler* (siehe Listing 10.3) registriert. Dieses Objekt wird damit von allen zukünftigen Änderungen der Eingangsbelegung des E/A-Moduls informiert.

Im nächsten Schritt wird mit Hilfe eines *Receive PDO* die Ausgangsbelegung und aufgrund der Feedback-Leitungen damit auch die Eingangsbelegung des E/A-Moduls verändert. In Zeile 28 u. 29 wird ein *ReceivePDO*-Objekt erzeugt, mit dem Wert $0x20$ bepackt und auf den Bus geschickt. Die resultierende Änderung der

[4] CANopen-Geräte können je nach Speicherkapazität unterschiedlich viele PDO-Konfigurationen unterstützen.

Eingangsbelegung veranlasst das E/A-Modul, die neue Eingangsbelegung über ein entsprechendes *Transmit PDO* zu publizieren. Diese CAN-Nachricht wird durch die *PDOHandler* Implementierung *MyHandler* (siehe Listing 10.3) abgefangen und innerhalb der Implementierung der *transmitPDOArrived()*-Methode auf der Konsole ausgegeben (vgl. Abbildung 10.5).

10.5 Zusammenfassung

Die Java CAN API realisiert eine objektorientierte Programmierschnittstelle (API) für den CAN-Feldbus in Java. Sie ermöglicht damit die schnelle Entwicklung von Java-Applikationen, die mit industriellen CAN-Geräten kommunizieren können. Zur Kommunikation mit der CAN-Hardware sind sowohl Konzepte auf Schicht 2 (CAN) als auch auf Schicht 7 (CANopen) vorhanden. Im Vordergrund stehen hierbei die Nachrichten bzw. Kommunikationsobjekte selbst, die objektspezifisch parametriert, analysiert und verschickt werden können.

Die Klassenhierarchie umfasst SDO-, PDO-, SYNC- und NMT-Abstraktionen sowie einen CANopen-Datentypübersetzer, die zusammen eine bequeme Verwaltung von CANopen-Geräteprofilen ermöglichen.

Die Klasse *CanPort* bildet die Schnittstelle zu der nativen JCan.DLL, die die Funktionalität der nativen CAN-Treiber-Bibliotheken zugänglich macht. Die JCan.DLL implementiert zusätzlich ein eigenes Nachrichten-ID-basiertes Nachrichtenmanagement, das sowohl native Nachrichtenfilterung als auch asynchrone Benachrichtigung aufgrund des Eintreffens bestimmter CAN-Nachrichten gemäß dem Observer-Pattern unterstützt.

Die Java CAN API wurde im Rahmen diverser prototypischer Anwendungen, z.B. aus den Bereichen Fernwartung und Integration von CAN-Geräten in Internet-basierte Lehr/Lernumgebungen, erfolgreich eingesetzt.

Kapitel 11

Das *Java Fieldbus Control Framework* (*JFCF*)

11.1 Einführung

Das *Java Fieldbus-based Control Framework* (*JFCF*) [BNKG01] ist ein objektorientiertes Java-Software-Framework zur einfachen Erstellung komplexer, nebenläufiger Steuerungslogik für industrielle Feldbusanlagen. Es besteht aus einer Abstraktionshierarchie für unterschiedliche industrielle Geräte und diversen Konzepten zur Verwaltung und Synchronisation nebenläufiger ausführbarer Einheiten. Steuerungslogik kann über generische Schnittstellen weitgehend hardwareunabhängig entwickelt und einzelnen ausführbaren Einheiten dynamisch zugeordnet werden.

11.1.1 Motivation

Im Bereich industrieller objektorientierter Steuerungen sind bereits einige Konzepte und Entwicklungssysteme in C++ realisiert worden. Die Realisierung eines derartigen Systems in Java ist nicht nur vor dem Hintergrund der zunehmenden Akzeptanz und Verbreitung von Java in diesem Umfeld interessant, sondern auch weil zusätzlich die Java-typischen Vorteile ausgenutzt werden können. Es wurde im Rahmen dieser Arbeit beispielsweise eine *JFCF*-basierte Umgebung geschaffen, die es erlaubt, Steuerungslogik über das Internet zu verschicken und ohne erneute Kompilierung auf dem laufenden System zur Ausführung zu bringen (siehe Abschnitt 11.4).

Weiterhin werden die bereits in Abschnitt 10.1.1 vorgestellten Vorteile der Programmiersprache Java, wie z.B. Plattformunabhängigkeit und automatisches Speichermanagement, auch im Umfeld industrieller Steuerungen nutzbar. Die Realzeit-Spezifikation für Java (RTSJ) [BG00a] wird es in Zukunft ermöglichen, derartige plattformunabhängige, wiederverwendbare Softwarelösungen auch realzeitfähig zu implementieren. Realzeitfähigkeit ist allerdings zunächst eine plattformabhängige Eigenschaft, die stark vom jeweiligen Geräteaufbau abhängt. Das bekannte WORA-Paradigma (*Write Once, Run Anywhere*) muss daher nach Greg Bollella[1] zu einem WOCRAC-Paradigma (*Write Once Carefully, Run Anywhere Conditionally*) dahingehend abgeschwächt werden, dass die Einhaltbarkeit der Deadlines u.U. auf jeder Hardware-Plattform neu überprüft werden muss.

JFCF in seiner jetzigen Form integriert direkt noch keine Konzepte der RTSJ, ist aber an den entscheidenden Stellen flexibel angelegt worden, um diese Konzepte in einem nächsten Schritt leicht in das Framework integrieren zu können.

11.1.2 Überblick

Abbildung 11.1 gibt einen Überblick über *JFCF*, das im Wesentlichen aus den beiden Schichten *control* und *devices* besteht. Die *devices*-Schicht besteht aus einer Anzahl abstrakter Geräteschnittstellen, einer Klassenhierarchie von Geräteabstraktionen und weiteren Klassen für Ereignis- und Fehlerbehandlung. Die *control*-Schicht liefert diverse Basisklassen und Schnittstellen zur Definition von nebenläufigen Steuerungs-Tasks, die über die abstrakten Geräteschnittstellen der *devices*-Schicht auf die Geräte-Klassenhierarchie und damit letztlich auf die Hardware zugreifen. Zum Zugriff auf die realen Geräte setzt *JFCF* im Moment direkt auf der Java CAN API auf, wobei der Großteil der in *JFCF* realisierten Konzepte unabhängig vom verwendeten Kommunikationsprotokoll bzw. der Feldbus-API sind.

Die Schlüsselkonzepte von *JFCF* sind:

- Unterscheidung von kommunikationsfähigen und logischen Geräten
- Trennung von Geräteschnittstellendefinition und deren Implementierung
- Trennung von ausführbarer Einheit und Steuerungslogik
- Definition von Gerätekomplexen als gesteuerte Einheiten

Die Unterscheidung von kommunikationsfähigen Geräten, die direkt an Netzwerkkommunikation teilnehmen können, und logischen Geräten, die jeweils von

[1] Vortrag auf der Konferenz ISORC 2001, Magdeburg.

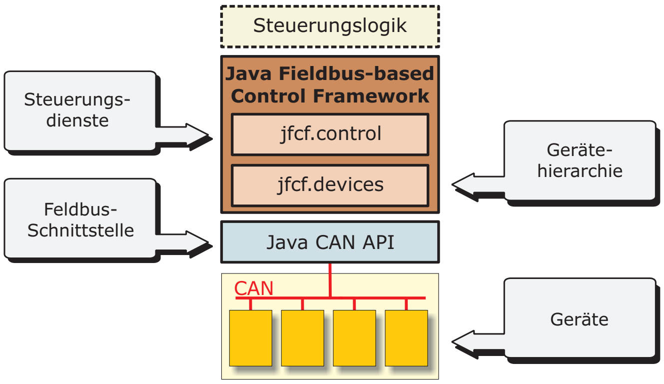

Abbildung 11.1: *JFCF*-Schichtenarchitektur

kommunikationsfähigen Geräten abhängig sind und eine abstraktere Funktion zur Verfügung stellen (z.B. ein Lichtsensor an einem E/A-Modul), spiegelt sich in der realisierten Geräte-Klassenhierarchie wider. Dabei wurde Wert darauf gelegt, die Schnittstellendefinition von Geräten streng von der Implementierung zu trennen, um die Steuerungslogik unabhängig von den verwendeten Kommunikationsparadigmen entwickeln zu können. Eine Steuerungslogik nutzt jeweils nur die generische Schnittstelle eines Gerätetyps (z.B. *Drive*), die innerhalb des Frameworks gerätespezifisch (z.B. für einen CANopen-Schrittmotor) implementiert ist.

Die Trennung von ausführbarer Einheit (z.B. ein gewöhnlicher Java-Thread) und Steuerungslogik hat den Vorteil, dass die Steuerungslogik zum Einen dynamisch über Internet geladen werden und auf einem laufenden System direkt ausgeführt werden kann (siehe Abschnitt 11.4). Zum Anderen kann die Steuerungslogik je nach Realzeit-Anforderungen in Zukunft zu unterschiedlichen ausführbaren Einheiten, wie z.B. den Thread-Konzepten *RealtimeThread* und *NoHeapRealtime-Thread* der RTSJ, dynamisch assoziiert werden.

Das *DeviceComplex*-Konzept erlaubt die Aggregation mehrerer abstrakter Geräteschnittstellen, einer Steuerungslogik und einer ausführbaren Einheit zu einem eigenständigen Steuerungs-Task.

Als Fallstudien wurden *JFCF*-Steuerungen für die bereits in Abschnitt 5.6 vorgestellte Automatisierungsanlage und eine Java-RMI-basierte Fernsteuerung der ebenfalls in Abschnitt 5.6 vorgestellten Werkstückvereinzelungsanlage realisiert. Die Steuerungen unterstützen jeweils ein GUI zum interaktiven Zugriff auf die Anlage sowie die selbstständige Steuerung der Anlage auf Standardkomponenten

(PC, *Windows NT*, *JRE 1.1.6*).

11.1.3 Stand der Forschung

Auf dem Gebiet der Steuerung industrieller Anlagen mit objektorientierten Konzepten wurden mehrere C++-basierte Ansätze erforscht. Ein mächtiges System, das im Rahmen des ADOORTA-Projekts[2] entwickelt wurde, unterstützt beispielsweise sämtliche Entwicklungsstufen bei der Entwicklung objektorientierter Realzeitsysteme [BGNP99]. Es integriert Komponenten zur Modellierung, Simulation, Code-Generierung und Überwachung von objektorientierten Programmabläufen sowie entsprechende graphische Benutzerschnittstellen. Der Code-Generator erstellt lauffähigen Code für RT-Unix-Systeme (z.B. QNX), basierend auf den erstellten Modellen der Modellierungskomponente. Als Zwischensprache wird die C++-Erweiterung AO/C++ [Per94] verwendet.

Die im Rahmen des HIGHROBOT-Projekts [KGLS97, KGSL97] entwickelten Konzepte unterstützen die Entwicklung von objektorientierten, nebenläufigen Realzeit-Steuerungen auf POSIX.4 Workstations in C++. Auf diesen Ergebnissen aufbauend wurden Konzepte zur komponentenbasierten Steuerung [Spe00] und zur Anbindung von Geräten als *Virtual Java Devices* an des World Wide Web entwickelt [LGK98, Lum99]. Die virtuellen Java-Geräte sind hierbei Stellvertreter realer Gerätekomplexe, die über eine CORBA-Architektur vordefinierte Funktionalität auf den Geräten anstoßen können. Die Ergebnisse und Erfahrungen, die innerhalb dieses Projekts gesammelt wurden, flossen teilweise auch in die Entwicklung von *JFCF* mit ein, das jedoch konsequent die Programmiersprache Java einsetzt und daher auch die Java-spezifischen Features (z.B. dynamisches Laden von Klassen) voll nutzen kann.

11.2 Architektur

11.2.1 Die Geräteschicht

Diese Schicht wird durch das Java-Paket *devices* (siehe Abbildung 11.2) implementiert. Sie besteht im Wesentlichen aus einer Anzahl abstrakter Geräteschnittstellen, einer Klassenhierarchie von Geräteabstraktionen, die alle von einer gemeinsamen Basisklasse abgeleitet sind, und weiteren Klassen für Ereignis- und Fehlerbehandlung. Sie realisiert ein Framework, das durch Ableitung auf mehreren abstrakten Ebenen schrittweise an konkrete Geräte angepasst werden kann.

[2] http://automation.delet.ufrgs.br/Adoorata/

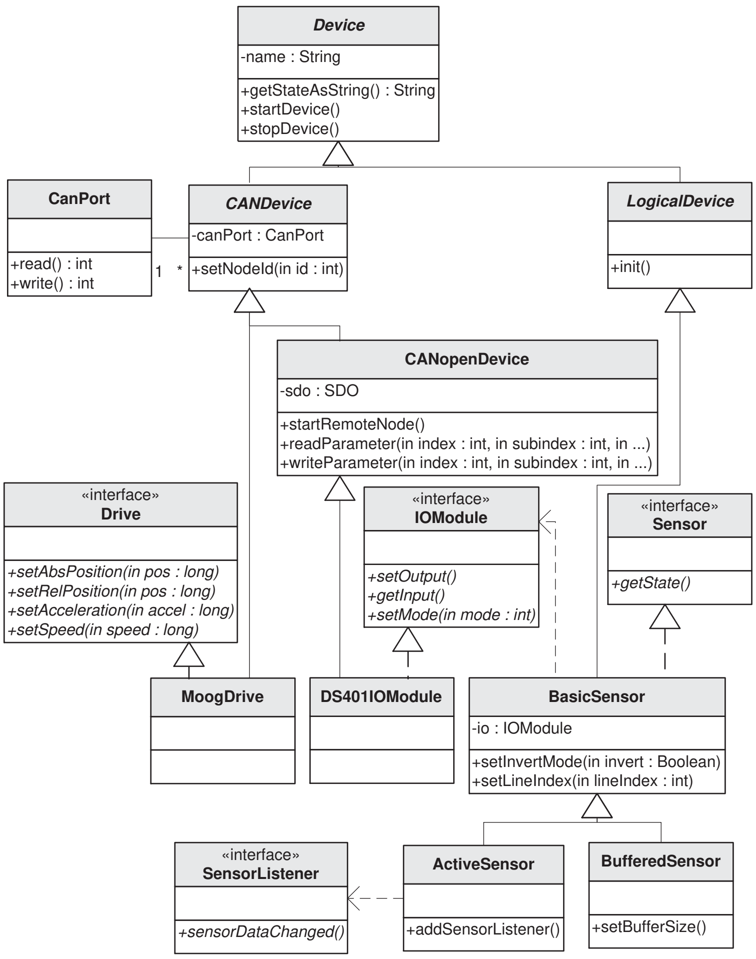

Abbildung 11.2: Ausschnitt aus der Klassenhierarchie des *devices*-Pakets

Die Klasse *Device* ist die Basisklasse für alle Geräteimplementierungen innerhalb von *JFCF*. Die Methode *getStateAsString()* dieser Klasse liefert für jedes Gerät eine textuelle Beschreibung des aktuellen Zustands, z.B. in Form von *XML*-Fragmenten. Diese Methode erlaubt das einfache Generieren einer textuellen Zustandsrepräsentation einer Anlage, indem diese Methode auf allen involvierten

Geräteobjekten aufgerufen wird und die Ergebnisse konkateniert werden.

Die von *Device* abgeleitete *CANDevice*-Klasse kapselt ein CAN-Gerät. Die Objekte dieser Klasse verwenden zum Zugriff auf das repräsentierte reale CAN-Gerät jeweils eine Referenz auf eine Instanz der *CanPort*-Klasse, die als Teil der Java CAN API implementiert ist. Die Klasse *CANopenDevice* erweitert die *CANDevice*-Klasse um CANopen-spezifische Funktionalität. Sie bietet Funktionen zum Setzen des CANopen-Knotenzustandes via NMT-Services (vgl. Abschnitt 10.3.2.2) und Zugriff auf sämtliche Geräteprofileinträge über die Methoden *readParameter()* und *writeParameter()* [Büh01]. Implementiert sind diese beiden Methoden durch die Klasse *SDO* der Java CAN API.

Abbildung 11.2 zeigt die momentan definierten kommunikationsprotokollunabhängigen Geräteschnittstellen *IOModule*, *Sensor* und *Drive*. Diese Schnittstellen verbergen die tatsächliche Implementierung der jeweiligen Geräte. Die Klasse *DS401IOModule* implementiert die *IOModule*-Schnittstelle beispielsweise durch eine spezialisierte *CANopenDevice*-Klasse, die ein CANopen-Profil gemäß dem Standard DS-401 [Can96a] auf dem E/A-Modul voraussetzt. Die Klasse *MoogDrive* implementiert andererseits die *Drive*-Schnittstelle als spezialisierte *CANDevice*-Klasse, da diese Motoren kein CANopen, sondern ein ein proprietäres Schicht-7-Protokoll verwenden (vgl. Abschnitt 5.6).

Bei der Realisierung einer Steuerungslogik (siehe nächster Abschnitt) werden die einzelnen Geräte immer über die generischen Schnittstellen und nie über die Implementierung direkt angesprochen. Für eine Steuerungslogik ist es beispielsweise nicht relevant zu wissen, wie ein bestimmter Ausgang eines E/A-Moduls feldbustechnisch gesetzt wird, sondern er muss nur gesetzt werden können. Dieser Ansatz ermöglicht einen hohen Grad an Wiederverwendbarkeit von Steuerungslösungen unabhängig von den verwendeten Kommunikationsprotokollen (CAN, CANopen, proprietäre Protokolle usw.).

Die Klasse *LogicalDevice* (siehe Abbildung 11.2) ist die Basisklasse für alle logischen Geräte, die nicht direkt über eine Netzwerkverbindung angesprochen werden können. Logische Geräte sind abhängig von kommunikationsfähigen Geräten, die durch das Netzwerk erreicht werden können. Sie implementieren i.d.R. eine Vor- oder Nachbearbeitung der Funktionalität der kommunikationsfähigen Geräte. Beispiele für logische Geräte sind Lichtsensoren als Vorverarbeitung optischer Information für E/A-Module oder Druckluftdüsen, die von E/A-Modulen kontrolliert werden.

Die Klasse *LogicalDevice* ist ebenfalls von *Device* abgeleitet, so dass logische Geräte über die *getStateAsString()*-Methode auf die gleiche Weise in einen Monitoring-Kontext eingebunden werden können wie kommunikationsfähige Geräte. In Abbildung 11.2 sind als Beispiele für logische Geräte diverse Implementierungen der *Sensor*-Schnittstelle gezeigt.

Die Klasse *BasicSensor* hält eine Referenz auf das assoziierte E/A-Modul als kommunikationsfähiges Gerät und verwaltet die Konfigurationsdaten eines Sensors. Als spezialisierte Sensoren sind in Abbildung 11.2 die Klassen *ActiveSensor* und *BufferedSensor* dargestellt. Ein *ActiveSensor*-Objekt publiziert Zustandsänderungen selbstständig unter allen registrierten *SensorListener*-Objekten durch Aufruf der entsprechenden *sensorDataChanged()*-Methodenimplementierungen gemäß dem Observer-Pattern. Die Klasse *BufferedSensor* speichert alle erfassten Zustandsänderungen in einer Queue, bis sie tatsächlich ausgelesen werden. Dieser Ansatz garantiert, dass selbst bei unregelmäßiger Kontrolle keine Zustandsänderungen übersehen werden.

11.2.2 Die Steuerungsschicht

Die Steuerungsschicht von *JFCF* wird durch das Paket *control* (siehe Abbildung 11.3) implementiert. Diese Schicht liefert diverse Basisklassen und Schnittstellen zur Definition von nebenläufigen Steuerungs-Tasks.

Die zentrale Klasse ist hier die *DeviceComplex*-Klasse, die eine Anzahl Geräte zu einer steuerbaren Einheit zusammenfasst. Die einzelnen Geräteobjekte können über die Methode *addDevice()* bei einem Gerätekomplex angemeldet werden und nehmen so automatisch an der Auswertung der *getStateAsString()*-Methode teil, die eine textuelle Repräsentation des momentanen Zustands eines Gerätekomplexes liefert. Über die Methoden *setLogic()* und *setControlThread()* werden einem Gerätekomplex die Steuerungslogik und die ausführbare Einheit zugeordnet, die innerhalb von *JFCF* getrennt verwaltet werden. Die Steuerung für einen derart definierten Gerätekomplex kann dann durch die Methode *startControl()* gestartet werden.

Die Basisklasse für alle Steuerungsalgorithmen ist die *AbstractLogic*-Klasse. Die abstrakte Methode *control()* beinhaltet dabei die eigentliche Steuerungslogik, die über die abstrakten Geräteschnittstellen des *devices*-Pakets auf die realen Geräte zugreift. Aufgerufen wird die *control*-Methode durch das Framework bzw. eine Implementierung der abstrakten *ControlThread*-Schnittstelle, die eine uniforme Verwaltung aller ausführbaren Einheiten innerhalb von *JFCF* erlaubt. Die *DefaultControlThread*-Klasse ist direkt von *java.lang.Thread* abgeleitet und realisiert eine Defaultimplementierung dieser Schnittstelle. Im Zuge einer zukünftigen Integration der RTSJ können hier durch Verwendung der *RealtimeThread*- und *NoHeapRealtimeThread*-Klassen realzeitfähige Implementierungen der *ControlThread* Schnittstelle realisiert werden.

I.d.R. müssen innerhalb der *control*-Methode der *AbstractLogic*-Klasse die

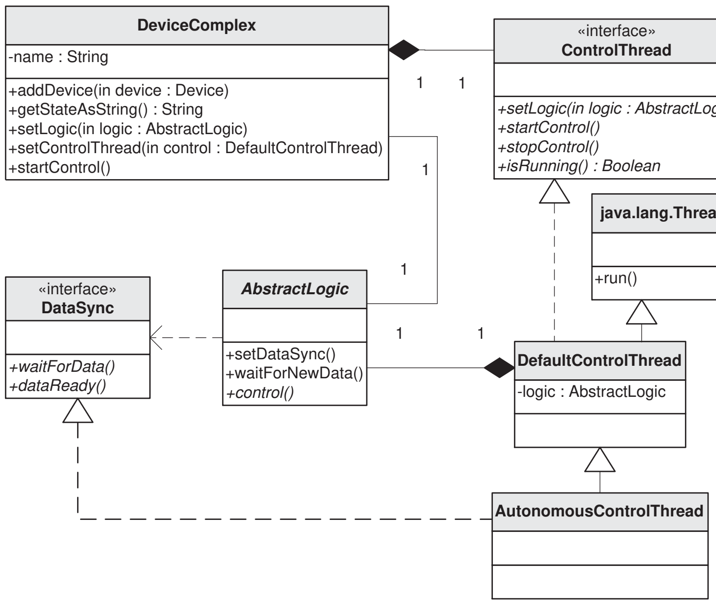

Abbildung 11.3: Ausschnitt aus der Klassenhierarchie des *control*-Pakets

einzelnen Ausführungsschritte des Steuerungsalgorithmus mit der Verfügbarkeit von Prozessdaten und/oder Buszyklen synchronisiert werden. Typischerweise zerfällt die Ausführung von Steuerungsalgorithmen in drei Phasen, die ständig iteriert werden. In der ersten Phase werden die aktuellen Prozessdaten gelesen. In der zweiten Phase analysiert die Steuerungslogik diese Daten und berechnet den neuen gewünschten Zustand, bzw. die Befehle, die zum Erreichen dieses Zustands ausgeführt werden müssen. Die Ausführung dieser Befehle erfolgt dann in der dritten Phase. Zur Synchronisation dieser Phasen verwendet die *AbstractLogic*-Klasse eine Referenz auf die *DataSync*-Schnittstelle.

Die Implementierung der *waitForData()*-Methode der *DataSync*-Schnittstelle wartet, bis auf dem Kommunikationsmedium neue Daten verfügbar werden, z.B. im Falle von CAN beim Eintreffen des nächsten SYNC-Telegramms. Entscheidend ist, dass das *DataSync*-Interface eine uniforme Schnittstelle zu Bussynchronisationsverfahren bildet, die protokollspezifisch implementiert werden kann. Diese Implementierungen können sehr unterschiedlich ausfallen. Die Steuerung einer Anlage über eine Java-RMI-Verbindung (siehe Abschnitt 11.4) erforderte beispielsweise eine Implementierung dieser Schnittstelle für unvorhersehbare Zykluszeiten, abhängig von der jeweils verfügbaren Internet-Bandbreite.

Die Klasse *AutonomousControlThread* implementiert dieses Interface selbstständig, für den Fall, dass keine externe Synchronisation notwendig ist.

11.3 Fallstudie 5: *JFCF*-basierte Steuerung einer Automatisierungsanlage

Mit Hilfe von *JFCF* wurde eine objektorientierte Steuerung für die in Abschnitt 5.6 vorgestellte Automatisierungsanlage komplett in Java realisiert. Da bei der Steuerung dieser Anlage keine Realzeitanforderungen zu beachten sind, konnte die Steuerung auf der Standard-Java-Laufzeitumgebung (*JDK 1.2.2*) und einem normalen PC unter *Windows NT* ausgeführt werden. Die Steuerung hat deswegen keine Realzeitanforderungen, weil die zeitkritische Ansteuerung der Robotermotoren in entsprechenden Hardware-Controllern gekapselt ist, die direkt die Angabe von Absolutpositionen erlauben. Auf alle anderen Zustände der Anlage kann jeweils sicher gewartet werden, so dass keine unerlaubten Systemzustände aufgrund von unvorhergesehen Laufzeitsystemverzögerungen auftreten können.

Wie bereits in Abschnitt 5.6 erwähnt, zerfällt die Automatisierungsanlage in vier Gerätekomplexe:

1. Werkstückvereinzelung (Klappen, Foto-Sensoren, Druckluftdüsen, E/A-Module)
2. Transfersystem (Motor, Foto-Sensoren, Positionierungseinheiten, E/A-Module)
3. Bosch SCARA60 Roboter (4 Motoren/Achsen)
4. Lift (Klappen, Foto-Sensoren, Druckluftdüsen, E/A-Module)

Wie in Abbildung 11.4 dargestellt, werden die einzelnen Gerätekomplexe jeweils durch entsprechende *DeviceComplex*- und *AbstractLogic*-Spezialisierungen modelliert.

Die Hauptklasse dieser Java-Applikation ist die Klasse *CellControl*. In einer Initialisierungsphase werden zunächst die notwendigen Geräteklassen instanziiert, die dann zu entsprechenden Gerätekomplexen zusammengefasst werden. Anschließend werden die zugehörigen Steuerungsklassen instanziiert und den einzelnen Gerätekomplexen zugeordnet. Weiterhin erstellt die *CellControl*-Klasse

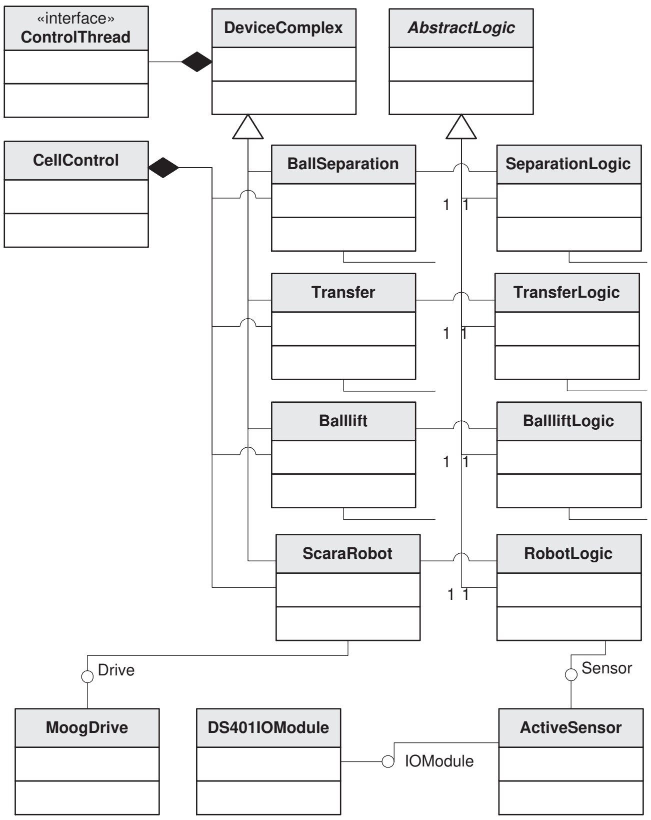

Abbildung 11.4: Ausschnitt aus der Architektur der Anlagensteuerung

ein GUI (siehe Abbildung 11.5), das den interaktiven Zugriff auf die einzelnen Anlagenkomponenten erlaubt.

Wenn die Steuerung gestartet wird, rufen die vier *ControlThread*-Implementierungen die Implementierung der abstrakten *control()*-Methode der vier *DeviceComplex*-Spezialisierungen auf, die dann quasi-parallel ausgeführt werden.

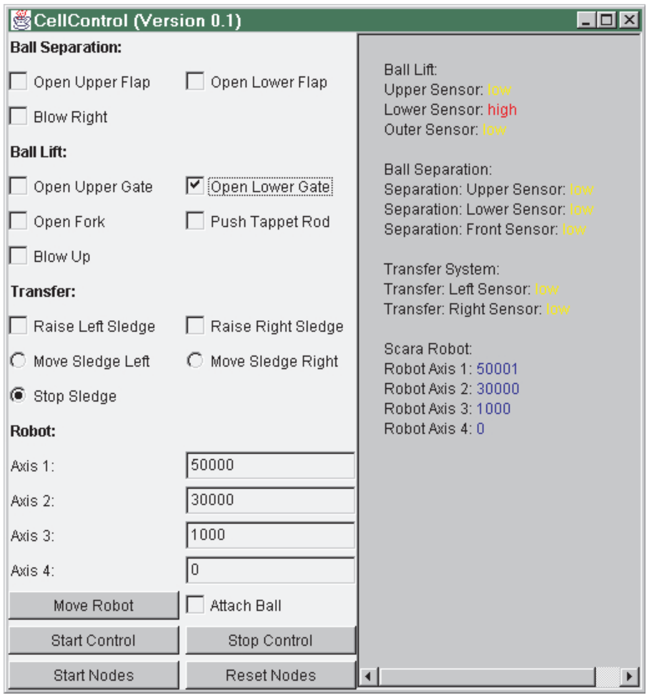

Abbildung 11.5: Die Benutzerschnittstelle der Anlagensteuerung

11.4 Fallstudie 6: *JFCF*-basierte Fernsteuerung einer Werkstückvereinzelung mit Java-RMI

11.4.1 Überblick

Der Hardware-Aufbau dieses Systems entspricht dem Werkstückvereinzelungskomplex der Automatisierungsanlage aus dem vorherigen Abschnitt. Da dieser Gerätekomplex ferngesteuert wird, wurde er als autarkes System realisiert, das ohne menschliche Interaktion auch bei unsachgemäßer Bedienung (nahezu) immer in einem legalen Zustand bleibt. Konkret bedeutet dies, dass Werkstücke, die korrekt vereinzelt wurden, über ein Röhren-System direkt wieder in den Vorratsbehälter überführt werden und Werkstücke nicht an Stellen liegen bleiben, von denen sie nicht mehr wegbewegt werden können.

Abbildung 9.2a (S. 154) zeigt ein statisches Foto des Hardware-Aufbaus im

Fernsteuerungs-Applet dieser Anlage. Der Benutzer kann durch Drücken der entsprechenden Tasten die einzelnen Klappen und Druckluftdüsen öffnen und schließen. Die Information, wo sich die einzelnen Werkstücke zu einem bestimmten Zeitpunkt befinden, ob die einzelnen Klappen geöffnet oder geschlossen sind usw., wird indirekt durch Visualisierung entsprechender Sensorsignale (in Abbildung 9.2a mit S1-S5 bezeichnet) dargestellt. Zusätzlich kann die Internet-Kamera (vgl. Abbildung 9.2c) benutzt werden, um ein Live-Bild der Anlage zu übertragen. Mit dieser Funktionalität ist es für den Benutzer möglich, die Anlage interaktiv fernzusteuern und kennenzulernen.

Das Steuerungs-Applet kann die Anlage auch selbstständig durch eine entsprechende Steuerungslogik steuern. Durch Drücken der *Start Control* -Taste kann ein Benutzer eine Default-Steuerungslogik verwenden oder eine Steuerungslogik dynamisch durch Angabe des Klassennamens und einer URL über ein Netzwerk laden, instanziieren und ausführen.

11.4.2 Architektur

Client/Server-Kommunikation Der Steuerungs-Client dieses Systems kommuniziert mit dem bereits erwähnten *SeparationServer*, der eine Teilkomponente des *RMI-CAN-Servers* darstellt (vgl. Abbildung 9.2), über Java-RMI. Auf Server-Seite ist die zentrale Schaltstelle die *RSServerImpl*-Klasse. Das *RSServerImpl*-Objekt ist eine Implementierung der von *java.rmi.Remote* abgeleiteten Schnittstelle *Remote-Separation* [Büh01], die die komplette, via Internet zugängliche Funktionalität des Server-Objekts und damit der Anlage definiert. Sie enthält Methoden zur Steuerung der Anlage (z.B. *openUpperFlap()*) und zum dynamischen Laden externer Steuerungslogik.

Die Implementierungen der Zugriffsmethoden werden auf Methoden entsprechender Geräteabstraktionen des *JFCF*-Frameworks abgebildet. Die Synchronisation mehrerer Steuerungs-Clients, die gleichzeitig auf das System zugreifen wollen, übernimmt das *ServerArbitration*-Framework, das bereits in Abschnitt 6.1.1 vorgestellt wurde.

Implementierung der Steuerungslogik Die Implementierung der Steuerungslogik ist als Spezialisierung der *JFCF*-Klasse *AbstractLogic* realisiert. Entscheidend für die Realisierung der assoziierten *DataSync*-Schnittstellenimplementierung ist die Tatsache, dass die Steuerungslogik nicht auf der virtuellen Maschine des Server-PC, sondern innerhalb des Web-Browsers des entfernten Benutzers ausgeführt wird. Sämtliche Steuerungsbefehle der Steuerungslogik werden (für den Programmierer transparent) auf entsprechende Java-RMI-Aufrufe abgebildet, die, je nach verfügbarer Netzwerkbandbreite, unvorhersehbare Ausführungszeiten aufweisen. Die verwendete Implementierung der *DataSync*-Schnittstelle implementiert daher ein Synchronisationsverfahren, das diesen Gegebenheiten Rechnung trägt. Die Implementierung der *waitForData()*-Methode wartet beispielsweise jeweils, bis die entsprechende Zustandsinformation mit Hilfe eines entsprechenden Java-RMI-Aufrufs vollständig über das Netzwerk transferiert wurde.

Bevor die Steuerungslogik gestartet wird, wird sie zunächst einer ausführbaren Einheit in Form einer *ControlThread*-Implementierung zugeordnet. Hierbei spielt es keine Rolle, ob die Steuerungslogik zur Compile-Zeit auf dem System bekannt war oder nicht, da diese innerhalb des Frameworks lediglich als *AbstractLogic*-Klasse aufgefasst wird. Falls eine fremde Steuerungslogik verwendet werden soll, transferiert der Client zunächst die Identifikation (Klassennamen und URL) der Steuerungslogik zum Server. Der Server lädt dann den spezifizierten Java-Byte-Code über eine HTTP-Verbindung und instanziiert ein entsprechendes Objekt über Java-Reflection. Das Objekt wird auf die Klasse *AbstractVirtualSeparationControl*, die direkt von *AbstractLogic* abgeleitet wurde, gecastet und zum Client transferiert. Dieser Umweg über den Server ist nötig, da ein Java-Applet aufgrund des Sicherheitskonzepts der Java-Programmiersprache keine HTTP-Anfragen an fremde Rechner richten darf.

Nach erfolgter Assoziation von ausführbarer Einheit und Steuerungsobjekt im Client ruft dann die ausführbare Einheit die *control()*-Methode der *AbstractLogic*-Spezialisierung auf und die Steuerungslogik wird ausgeführt.

Die Implementierung der Steuerungslogik kann hierbei unabhängig von der realen Anlage erfolgen. Die zwei einzigen Anforderungen an die Implementierung einer neuen Steuerungsklasse für die Werkstückvereinzelung sind, dass die neue Klasse von der bereits erwähnten *AbstractVirtualSeparationControl*-Klasse abgeleitet wird, und dass zum Zugriff auf die Anlage die in dieser Klasse vorhandene Referenz auf die Schnittstelle *VirtualSeparation* verwendet wird. Diese Schnittstelle ist der bereits erwähnten *RemoteSeparation*-Schnittstelle als weitere Indirektionsstufe vorgeschaltet und erlaubt alternativ zu Java-RMI die Integration weiterer Middleware-Lösungen[3]. Die Abbildung auf die gewünschte Middleware-Lösung, also im Falle von Java-RMI auf die *RemoteSeparation*-Schnittstelle, wird von der Klasse *VirtualSeparationHandler* durchgeführt.

Die Implementierung der *VirtualSeparation*-Schnittstelle muss hierbei nicht zwingend auf die realen Geräte durchgreifen, sondern kann diese auch nur simulieren. Genau diese Eigenschaft wurde dazu genutzt, eine Laborübung zu realisieren, in der Studenten die Steuerungslogik der vorgestellten Anlage zunächst

[3]Konkret wurde zusätzlich eine HTTP-basierte Kommunikation über ein entsprechendes *Separation*-Servlet realisiert, um diese Übung auch über Firewalls hinweg durchführen zu können.

innerhalb einer Simulation entwickeln und die entwickelte Steuerungsklasse im Anschluss daran ohne weitere Modifikationen direkt innerhalb des vorgestellten Fernsteuerungs-Client ausführen können.

11.5 Zusammenfassung

Das *Java Fieldbus-based Control Framework* ist ein schlankes Java-Software-Framework zur einfachen Erstellung von Steuerungslogiken für industrielle Feldbusanlagen. Um das Framework möglichst unabhängig von den verwendeten Feldbusprotokollen zu halten, wurde auf eine konsequente Trennung zwischen Schnittstellen und Implementierung geachtet. So wurden beispielsweise generische Schnittstellen für E/A-Module, Sensoren, Motoren und zur Kommunikationssynchronisation entworfen, die dann protokollspezifisch implementiert werden können. Dieser Ansatz erlaubt die Realisierung portabler Steuerungsklassen gemäß dem WOCRAC-Paradigma.

Die durch *JFCF* realisierte Trennung von ausführbarer Einheit und Steuerungslogik erlaubt zum Einen die einfache Integration der RTSJ-Konzepte *RealtimeThread* und *NoHeapRealtimeThread* über die generische *ControlThread*-Schnittstelle, zum Anderen den dynamischen Download und die Instanziierung fremder Steuerungslogik zur Laufzeit.

Zur Validierung des Frameworks wurden zwei Fallstudien durchgeführt, die beide die Java CAN API als Feldbuskommunikationsschnittstelle verwenden. Die erste Fallstudie realisiert eine komplexe Steuerung einer Automatisierungsanlage, die aus vier nebenläufigen Tasks für die Gerätekomplexe Werkstückvereinzelung, Transfersystem, Roboter und Liftsystem aufgebaut wurde.

In der zweiten Fallstudie wurde ein Java-RMI-basiertes Client/Server-System realisiert, das die Fernsteuerung einer Werkstückvereinzelungsanlage erlaubt. Die Steuerungslogik kann hierbei als Konzept der *JFCF* dynamisch via Internet geladen, instanziiert und innerhalb des Clients ausgeführt werden. Dieses System wird weiterhin eingesetzt, um eine Internet-basierte Laborübung zur Gerätesteuerung zu realisieren.

Kapitel 12

Internet-basierte Laborübungen

In diesem Kapitel werden die im Kontext des VVL-Projekts entwickelten Laborübungen [GKNB99, BKGN00, GKBN01] vorgestellt. Sämtliche Übungen sind rund um die Uhr verfügbar[1] und können über Standard-Web-Browser bedient werden. Tabelle 12.1 gibt eine Übersicht über die entwickelten Übungen und ihre jeweiligen Lernziele.

Die einzelnen Übungen sind in die Abschnitte *Motivation*, *Zielgruppe*, *Lernziele*, *Voraussetzungen*, *Umfeld*, die eigentliche Übung und *Feedback* gegliedert. Alle Übungen sind gemäß dieser Gliederung über die Homepage des Projekts zugänglich (siehe Abbildung 12.1). Diese Homepage realisiert uniforme Zugangswege zu den einzelnen Übungen und Demonstrationen sowie zu weiteren Online-Ressourcen des Projekts. Die Navigationskomponente dieser Seite, die als integriertes Java-Applet realisiert wurde (siehe linke Bildhälfte), unterstützt den direkten Zugriff auf alle relevanten Seiten innerhalb der Homepage und ermöglicht so eine intuitive Navigation. Diese Navigationskomponente ist über Konfigurationsdateien flexibel anpassbar und wurde ohne spezielle Modifikationen auch für die Homepage des MFB-Projekts (siehe Abschnitt 13.1) verwendet.

An jede Übung schließt sich ein als *HTML*-Formular realisiertes Feedback-Formular an, das eine Rückkopplung der Benutzer mit den Entwicklern der Übungen ermöglicht. Diese Daten werden zusätzlich automatisch über ein entsprechendes CGI-Skript verbundprojektweit zentral erfasst und ausgewertet.

Alle drei Übungen wurden bereits mehrere Male in entsprechenden Vorlesungen

[1] siehe http://www-sr.informatik.uni-tuebingen.de/vvl/

Titel	Lernziele
CANopen-Geräteprofile	1. Verstehen des CANopen-Geräteprofilkonzepts 2. Verstehen des konkreten Geräteprofils eines E/A-Moduls 3. Sammeln von Erfahrungen in der Konfiguration von CANopen-Geräten 4. Verstehen der SDO- und PDO-Kommunikationsobjekte. 5. Verstehen des CANopen-Zustandsdiagramms und der entsprechenden Zustandsübergänge.
Java-RMI-basierter Zugriff auf eine Leuchtschrift	1. Vermittlung von Client/Server-Grundlagen 2. Verstehen von Teleservice-Architekturen 3. Vermittlung von Konzepten objektorientierter Middleware
Steuerung einer Werkstück-vereinzelungsanlage	1. Vermittlung von Grundlagen objektorientierter Gerätesteuerung 2. Vermittlung von Grundlagen nebenläufiger Programmierung 3. Aufzeigen der Grenzen von Gerätesimulationen im Vergleich mit realen Gerätesteuerungen.

Tabelle 12.1: Realisierte Internet-basierte Laborübungen

und Praktika an der Universität Tübingen und der Fachhochschule Reutlingen erfolgreich durchgeführt.

12.1 Übung 1: CANopen-Geräteprofile

Diese Übung vermittelt interaktive Erfahrungen im Umgang mit CANopen-Geräten. Das *Java Remote CAN Control* (*JRCC*) System, das für diese Übung implementiert wurde, erlaubt den Internet-basierten lesenden und schreibenden Zugriff auf CANopen-Geräteprofile. Es können beliebige CANopen-Geräte (z.B. Motoren, E/A-Module usw.) direkt am System angemeldet werden, ohne dass eine aufwändige Konfiguration des Systems notwendig wird. Zusätzlich unterstützt

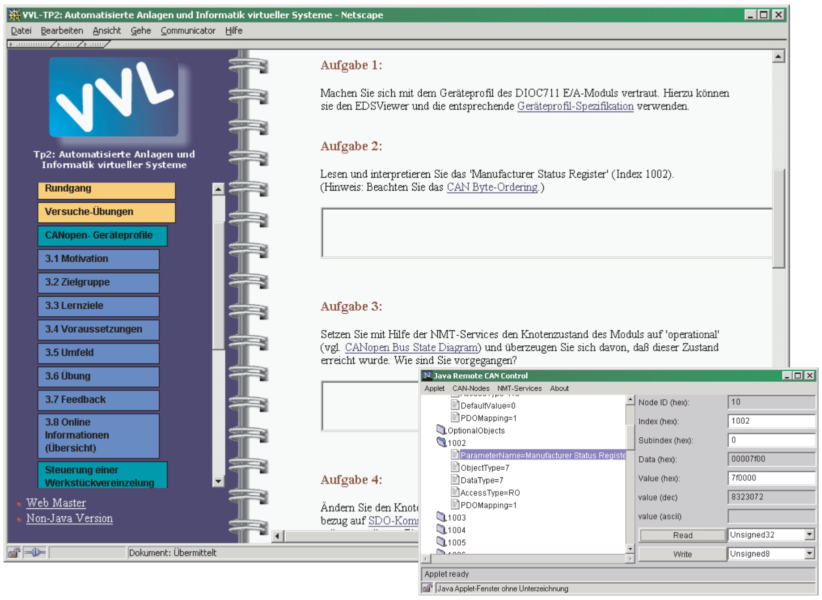

Abbildung 12.1: Präsentation der Übungen im World Wide Web

das System den Zugriff auf CANopen-NMT-Services, wodurch Zustandsübergänge einzelner Geräte innerhalb des CANopen-Knotenzustandsdiagramms angestoßen werden können.

Außerhalb dieses Anwendungsfeldes kann das *JRCC*-System als einfaches Fernwartungssystem für CANopen-Anlagen aufgefasst werden. Prinzipiell kann das System beliebige CANopen-Parameter beliebiger CANopen-Geräte setzen und lesen und so vollständigen Zugriff auf ein Gerät vermitteln.

12.1.1 Das *Java Remote CAN Control* (JRCC) System

JRCC [BNGK99] ist ein Client/Server-System bestehend aus dem *JRCC*-Server und dem *JRCC*-Client (siehe Abbildung 9.2). Client und Server kommunizieren gemäß einem speziell entwickelten Applikationsprotokoll über Java-Sockets. Der Server wiederum kommuniziert mit den CANopen-Geräten mit Hilfe der Java CAN API. Der Client ist als Java-Applet implementiert und kann direkt über entsprechende *HTML*-Seiten referenziert und gestartet werden.

Die folgende Funktionalität wird durch das *JRCC*-System unterstützt:

- Lesender/schreibender Zugriff auf beliebige Geräteprofileinträge
- Decodierung der CANopen-spezifischen Datentypen
- Zugriff auf NMT-Services
- Visualisierung von EDS-Dateien

Ein Benutzer kann die Identifikation eines bestimmten Parameters auf einem bestimmten Gerät durch entsprechende GUI-Komponenten des *JRCC*-Clients angeben (siehe Abbildung 9.2d) und über die Tasten *Read*/*Write* lesend bzw. schreibend auf das ausgewählte Gerät zugreifen. Der Client generiert hierfür entsprechende Anfragen an den *JRCC*-Server, der den spezifizierten Parameter über den CAN-Bus liest bzw. schreibt und das Ergebnis an den Client zurückgibt. Falls zusätzlich der CANopen-Datentyp des Parameters angegeben wird, werden die jeweiligen Parameterwerte entsprechend [Can96c] (de-)kodiert.

Zum Zugriff auf die Geräteprofileinträge werden entsprechende CANopen *Service Data Objects* (siehe Abschnitt 10.1.2.1) verwendet, die durch die Klasse *SDO* der Java CAN API implementiert sind. Die Kodierung der CANopen-spezifischen Datentypen wird an die bereits erwähnte Klasse *DataDecoder* delegiert.

Der Zugriff auf einzelne NMT-Services wird über entsprechende Dialoge des *JRCC*-Clients ermöglicht. Der *JRCC*-Server reicht derartige Anfragen an die ebenfalls bereits vorgestellte Klasse *NMT* der Java CAN API weiter.

Die Konfiguration eines *JRCC*-Systems wird über Applet-Parameter innerhalb der *HTML*-Seiten festgelegt, die den *JRCC*-Client starten. Die zwingend erforderliche Konfiguration besteht hierbei lediglich aus der Angabe der CANopen-Geräte-IDs, die über das System zugänglich sein sollen, und den Dateinamen der zugehörigen EDS-Dateien. Diese Dateien werden beim Start des Applets über die Socket-Verbindung vom *JRCC*-Server geladen und in einer entsprechenden GUI-Komponente visualisiert (siehe Abbildung 9.2d links). Die Visualisierung der EDS-Information erlaubt das Durchsuchen des Geräteprofils eines Knotens nach bestimmten Parametern. Die Identifikation eines dort beschriebenen Parameters kann dann per Doppelklick in die entsprechenden *Index*- und *Subindex*-GUI-Komponenten übertragen und zum Zugriff auf den Parameter verwendet werden.

Zusätzlich können innerhalb der *JRCC*-Konfiguration beliebige Parametereinträge gesperrt werden, um diese vor unsachgemäßem Zugriff zu schützen. Ebenso können auch weiterreichende Privilegien verteilt werden (z.B. kann die beschriebene Anmeldepflicht für Geräte ausgesetzt und der Zugriff auf beliebige Geräte am Bus erlaubt werden).

Die Tatsache, dass die Konfiguration innerhalb der *HTML*-Seite erfolgt, die den *JRCC*-Client referenziert, hat den Vorteil, dass mit ein und derselben

Server-Konfiguration viele unterschiedliche Client-Konfigurationen bedient werden können, wobei die unterschiedlichen Client-Konfigurationen über unterschiedliche *HTML*-Seiten definiert sind. Die Zugangsberechtigung zu einer bestimmten Client-Konfiguration kann daher über entsprechende Sicherheitskonzepte des Web-Servers (z.B. über *.htaccess*-Dateien beim Web-Server Apache) geregelt werden.

12.1.2 Aufgabenstellung

Abbildung 12.1 zeigt einen Ausschnitt aus der Aufgabenstellung dieser Laborübung. Sie besteht aus mehreren Teilaufgaben, die den Benutzer Schritt für Schritt mit dem CANopen-Geräteprofilkonzept vertraut machen. Die wesentlichen Teilaufgaben sind hierbei, bestimmte Geräteprofileinträge zu lesen und zu interpretieren, die aktuelle PDO-Konfiguration zu analysieren sowie den CANopen-Knotenzustand des Gerätes zu verändern und das dadurch veränderte Verhalten zu verstehen. Zum entfernten Zugriff auf die Geräte wird das *JRCC*-System verwendet, wobei der *JRCC*-Client direkt aus der Aufgabenstellung heraus gestartet werden kann.

Die Ergebnisse und Beobachtungen werden von den Benutzern in entsprechende Textfelder der Aufgabenstellung eingetragen. Durch Drücken einer entsprechenden *HTML*-Formulartaste werden diese Daten zusammen mit der Identifikation des Benutzers über ein CGI-Skript auf den Server übertragen. Dort können die Ergebnisse von entsprechenden Experten korrigiert und bewertet werden.

Unabhängig von der Aufgabenstellung haben die Benutzer durch das *JRCC*-System die Möglichkeit, mit den Geräten in einer selbstständigen Weise zu experimentieren. Zusätzlich wird das *JRCC*-System während Vorlesungsveranstaltungen in Verbindung mit einem Videobeamer verwendet, um bestimmte CANopen-Phänomene live zu demonstrieren.

12.2 Übung 2: Java-RMI-basierter Zugriff auf eine Leuchtschrift

Das System zum Zugriff auf die Leuchtschrift ist ein Java-RMI-basiertes Client/Server-System, wobei die Server-Komponente in den RMI-CAN-Server (siehe Abbildung 9.2) integriert wurde. Der Hardware-Aufbau besteht aus einem digitalen CANopen-E/A-Modul, dessen Ausgänge mit den einzelnen Buchstaben der Leuchtschrift verbunden sind, so dass sie einzeln an- bzw. ausgeschaltet werden können.

Das im RMI-CAN-Server angemeldete *DisplayImpl*-Objekt implementiert und exportiert die von *java.rmi.Remote* abgeleitete Schnittstelle *DisplayInterface* [Büh01], die die via Java-RMI erreichbare Schnittstelle dieser Anlage definiert. Das entsprechende *DisplayClient*-Applet (siehe Abbildung 9.2e) erlaubt das An- und Ausschalten einzelner Buchstaben und das Starten einer gespeicherten Abfolge von derartigen Befehlen. Der Zustand der Leuchtschrift kann sowohl über das Live-Bild der Internet-Kamera als auch direkt über das Client-Applet, das die Farben der GUI-Komponenten dem tatsächlichen Zustand der entfernten Anlage anpasst (vgl. Abbildung 9.2e), überwacht werden.

Das E/A-Modul ist über entsprechende Konzepte des *Java Fieldbus-based Control Frameworks* gekapselt, wobei der Zugriff auf die Hardware wieder über die Java CAN API erfolgt. Die Synchronisation von konkurrierenden Client-Anfragen wird durch das *ServerArbitration*-Framework (siehe Abschnitt 6.1.1) geregelt.

Dieses System kann in weiten Teilen als Spezialisierung des universellen *JRCC*-Systems auf einen bestimmten Anwendungskontext, hier das Bedienen einer Leuchtschrift, betrachtet werden. Die Leuchtschrift kann zwar ebenfalls mit dem *JRCC*-System gesteuert werden, indem entsprechende Werte an entsprechende Positionen des Geräteprofils geschrieben werden, wobei diese Bedienung jedoch Kenntnisse über CANopen-Geräteprofile voraussetzt. Das komfortablere GUI des *DisplayClient*-Applet verliert hingegen an Universalität gegenüber dem generischen *JRCC*-System.

12.2.1 Aufgabenstellung

Die Aufgabenstellung besteht darin, ein Applet zu implementieren, das die von der *DisplayImpl*-Klasse implementierte *DisplayInterface*-Schnittstelle via Java-RMI referenziert und zum Zugriff auf die entfernte CAN-Anlage nutzt. Ein entsprechendes Rahmenprogramm für dieses Applet kann zusammen mit dem zugehörigen Dokumentationsmaterial über die Web-Seiten geladen werden. Die Aufgabenstellung führt dann Schritt für Schritt durch die Ergänzung dieses Rahmenprogramms, bis die volle Funktionalität des *DisplayClient*-Applets erreicht ist. Es müssen hierfür zum Einen die für einen RMI-Client typischen Schritte, wie z.B. Kommunikation mit der entfernten RMI-Registry, durchgeführt, zum Anderen die Besonderheiten der Geräteintegration beachtet werden.

Die Implementierung der Client-Lösungen erfolgt hierbei unabhängig von dem *DisplayImpl*-Objekt, das auf dem entfernten Server-PC ausgeführt wird. Die korrekten Applet-Lösungen erlauben schließlich den Zugriff auf die entfernte Leuchtschrift, was von den Studenten als besonders motivierend empfunden wurde.

12.3 Übung 3: Steuerung einer Werkstückvereinzelungsanlage

In Rahmen dieser Aufgabe entwickeln die Studenten eine objektorientierte Steuerungslogik für die bereits vorgestellte Werkstückvereinzelungsanlage (siehe Abschnitt 5.6 u. 11.4).

12.3.1 Aufgabenstellung

Die Steuerungslogik wird zunächst innerhalb eines Java-Rahmenprogramms implementiert, das eine Simulation der realen Anlage darstellt (siehe Abbildung 9.2b). Das Rahmenprogramm kann wieder von den Studenten über entsprechende Web-Seiten via Internet geladen und ergänzt werden.

Die Steuerungsklasse wird, wie bereits in Abschnitt 11.4.2 beschrieben, innerhalb des *JFCF*-Frameworks als Spezialisierung der *AbstractLogic*-Klasse implementiert. Sie benützt zur Kommunikation mit der in diesem Falle simulierten Anlage ausschließlich die Schnittstelle *VirtualSeparation* [Büh01].

Die Hauptaufgabe für die Studenten besteht in der Implementierung der entsprechenden *JFCF-control()* Methode, die eine korrekte und effiziente Vereinzelung der Werkstücke durchführt. Hierfür werden Methoden wie z.B. *openUpperFlap()* der *VirtualSeparation*-Schnittstelle aufgerufen und anschließend solange gewartet, bis die verfügbare Sensor-Information das erfolgreiche Öffnen der Klappe anzeigt. Schreibende und lesende Zugriffe werden hierbei, wie innerhalb von *JFCF* üblich, über eine Implementierung der *DataSync*-Schnittstelle synchronisiert.

Wenn die erstellte Steuerungsklasse innerhalb der Simulation zufriedenstellend läuft, können die Studenten diese Klasse auf der realen Anlage testen. Hierzu müssen sie lediglich die Java-*Class*-Datei der Steuerungsklasse via Internet verfügbar machen, z.B. indem sie sie in das dem universitären Web-Server zugängliche Unterverzeichnis ihres *Home-Directories* ablegen. Als nächstes öffnen Sie den Steuerungs-Client der realen Anlage (siehe Abbildung 9.2a) über einen *HTML*-Link innerhalb der Aufgabenstellung. Innerhalb dieses Applets können sie den Klassennamen und die URL der abgelegten Java-Class-Datei angeben. Der assoziierte Server-Prozess des Steuerungs-Client lädt und instanziiert (vgl. Abschnitt 11.4) daraufhin die von den Studenten erstellte und referenzierte Steuerungsklasse und gibt sie als Ergebnis eines entsprechenden RMI-Aufrufs an den Client zurück. Dieser assoziiert das Steuerungsobjekt zu einer *ControlThread*-Implementierung und ruft über deren *startControl()*-Methode die *control()*-Methodenimplementierung des Steuerungsobjekts auf. Innerhalb der *control()*-Methode kann nun über die *VirtualSeparationHandler*-Klasse und die

RemoteSeparation Schnittstelle mit der realen Anlage kommuniziert werden.

I.d.R. arbeiten die auf der Simulation erstellten Steuerungsalgorithmen auf der realen Anlage nicht zufriedenstellend, da sich die reale Anlage in einigen Punkten komplexer verhält als die Simulation. Die Vermittlung dieses typischen Sachverhalts ist ein wichtiges Lernziel dieser Übung. Als Zusatzaufgabe können die Studenten die Steuerungsklasse weiter verfeinern, bis sie auch auf der realen Anlage zufriedenstellende Ergebnisse liefert.

12.4 Zusammenfassung

Im Rahmen des VVL-Projekts wurden drei Internet-basierte Laborübungen entwickelt, die rund um die Uhr zugänglich sind und die ohne menschliche Interaktion auf der Server-Seite von entfernten Benutzern selbstständig durchgeführt werden können. Die Übungen sind in eine konsistent navigierbare Homepage eingebettet, die die einzelnen Übungen in gegliederter Form zur Verfügung stellt.

Die drei Laborübungen unterscheiden sich bzgl. ihrer Lernziele und damit auch bzgl. der intendierten Zielgruppe. Die erste Übung erlaubt den interaktiven Zugriff auf industrielle Feldbusgeräte gemäß dem CANopen-Protokoll. Sie vermittelt damit Wissen und Erfahrung in der Analyse, Konfiguration und Handhabung von CANopen-Feldbusgeräten.

Die zweite Übung behandelt Teilaspekte der Technologie selbst, die zur Realisierung der Internet-basierten Laborübungen notwendig sind. Es werden anhand einer exemplarischen, objektorientierten Client/Server-Architektur mögliche Konzepte zur Integration realer Geräte in Internet-basierte Informationssysteme aufgezeigt. Konkret entwickeln die Studenten einen Java-RMI-Client zum Zugriff auf eine entfernte CANopen-Anlage, ohne jedoch über die entfernte Anlage mehr wissen zu müssen, als ihre RMI-Schnittstelle zu kennen.

Die Erstellung objektorientierter Gerätesteuerungen und das Aufzeigen von Grenzen der Gerätesimulation ist Thema der dritten Übung. Es wird zunächst ein objektorientierter Steuerungsalgorithmus als Teil eines Simulations-Applets entwickelt, der nach erfolgreichem Test auch auf der realen, entfernten Anlage ausgeführt werden kann.

Alle Übungen können innerhalb von Standard-Web-Browsern ohne zusätzliche Plug-Ins durchgeführt werden, wobei zur Durchführung von Aufgabe 2 und 3 zusätzlich der Zugriff auf eine Java-Entwicklungsumgebung[2] vorausgesetzt wird. Die Übungen wurden bereits mehrere Male in entsprechenden Vorlesungen an

[2] Z.B. kostenlos erhältlich unter http://java.sun.com.

der Universität Tübingen und der Fachhochschule Reutlingen erfolgreich durchgeführt.

Kapitel 13

Integration von Computeralgebrasystemen in Internet-basierte Lehr-/Lernumgebungen

13.1 Überblick

Die Ergebnisse, die in diesem Kapitel vorgestellt werden, sind im Rahmen des Projektes *Mathematik für BioInform@tiker*[1] (MFB) entstanden. Dieses Projekt war ein Bestandteil des Verbundprojekts *BioInform@tik*[2] an der Universität Tübingen, das durch die Multimedia-Gemeinschaftsinitiative Baden-Württemberg/Deutsche Telekom AG gefördert wurde.

Ziel des Projekts war die Erstellung einer Internet-basierten Lehr-/Lernumgebung zur Unterstützung einer einführenden Mathematikvorlesung an der Universität Tübingen. Das Vorlesungsskript wurde mit Hilfe von LaTeX2HTML und weiteren Perl-Skripten in ein *HTML*-basiertes Kursbuch übersetzt und in das bereits in Kapitel 12 vorgestellte Navigationskonzept integriert. Die so entstandene MFB-Homepage (siehe Abbildung 13.1) stellt zusätzlich neben einer Volltextsuche auch ein schwarzes Brett und eine Chat-Möglichkeit zur Verfügung, so dass sich Studenten am schwarzen Brett verabreden können, um bestimmte Aufgabenstellun-

[1] http://mfb.informatik.uni-tuebingen.de

[2] http://www-ra.informatik.uni-tuebingen.de/bioinformatik/

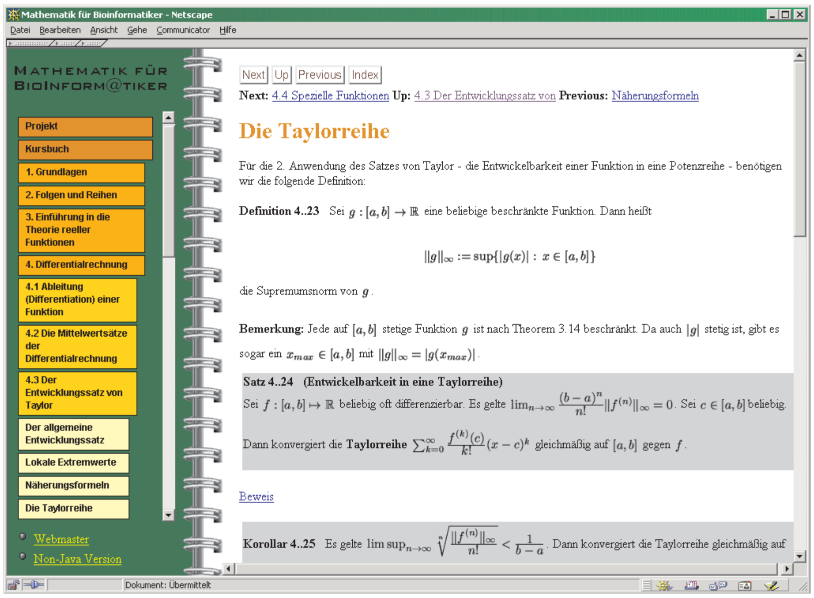

Abbildung 13.1: Der MFB-Internet-Auftritt

gen über den Chat zu diskutieren.

Zusätzlich wurde eine Werkzeugsammlung (*Mathe-Tools*) entwickelt und in die MFB-Homepage integriert, die die innerhalb des Kursbuchs beschriebenen mathematischen Konzepte visualisiert und interaktiv erfahrbar macht. Realisiert sind die einzelnen Komponenten dieser Werkzeugsammlung als Java-Applets, die zum Einen direkt aus dem Kursbuch an den relevanten Stellen referenziert werden, zum Anderen über separate Navigationseinträge zugänglich sind.

Die realisierten Applets (siehe Tabelle 13.1) lassen sich gemäß ihrer Visualisierungsverfahren in drei unterschiedliche Kategorien einteilen:

1. Visualisierung in Java
2. Integration externer Visualisierungskomponenten
3. Mischformen aus Kategorie 1 und 2 (hybride Visualisierung)

Als externe Visualisierungskomponenten werden momentan die Computeralgebrasysteme Maple und Mathematica eingesetzt. Wichtig ist hierbei, dass die Unterscheidung der Visualisierungskategorien für einen Benutzer transparent bleibt und

Mathematisches Konzept	Abb.	Kategorie
Folgen und Reihen im $\mathbb{R}^2$	-	1
Funktionen im $\mathbb{R}^2$	13.3	1
Funktionen und Singularitäten im $\mathbb{R}^2$	13.5	3 (Maple-Anbindung)
Partiell definierte Funktionen im $\mathbb{R}^2$	-	1
Komplexe Zahlen	-	1
Komplexe Folgen und Reihen	13.2e	1
Animierte Differentiation im $\mathbb{R}^2$	13.2h	1
Symbolische Integration/Differentiation	-	2 (Maple-Anbindung)
Matrizenrechner	13.2c	1
Vektoren im $\mathbb{R}^2$	-	1
Koordinatentransformation im $\mathbb{R}^2$	-	1
Geraden und Ebenen im $\mathbb{R}^3$	13.2a	2 (Maple-Anbindung)
Geraden und Ebenen im $\mathbb{R}^3$	13.2g	1 (Java 1.0 3D-Bibliothek [Han])
Parameterfläche im $\mathbb{R}^3$	13.2b	2 (Maple-Anbindung)
Affine Transformation im $\mathbb{R}^3$	13.2f	1 (Java 1.0 3D-Bibliothek)
Integration im $\mathbb{R}^3$	13.2d	2 (Mathematica-Anbindung)

Tabelle 13.1: Übersicht über die *Mathe-Tools*-Werkzeugsammlung

die entwickelten Applets einen einheitlichen, intuitiven Zugang zu den unterstützten Konzepten ermöglichen. Insbesondere verbergen die Applets der Kategorie 2 und 3 die spezifische Syntax der integrierten externen Visualisierungskomponenten hinter intuitiven GUI-Komponenten. Die externen Visualisierungskomponenten werden über entsprechende Client/Server-Architekturen transparent eingebunden und müssen nur Server-seitig installiert sein. Sämtliche Applets benötigen daher nur die virtuelle Java-Maschine der Standard-Web-Browser, um ausgeführt werden zu können. Abbildung 13.2 zeigt einige Screen-Shots der realisierten Applets.

13.2 Die *Mathe-Tools*-Werkzeugsammlung

Die Applets der *Mathe-Tools*-Werkzeugsammlung (siehe Tabelle 13.1) sind über die MFB-Homepage unter *http://mfb.informatik.uni-tuebingen.de* frei zugänglich. Sie sind direkt in den Standard-Web-Browsern lauffähig und setzen keinerlei zusätzliche Software oder Browser-Plug-Ins voraus. Für jedes Applet sind eine detaillierte Bedienungsanleitung und meist auch konkrete Anwendungsbeispiele online verfügbar. Die mathematischen Beispiele starten i.d.R. die zugehörigen Applets in einem entsprechend vorkonfiguriertem Zustand, so dass die Beispiele direkt visualisiert werden. Das in Abbildung 13.3 gezeigte Beispiel definiert bei-

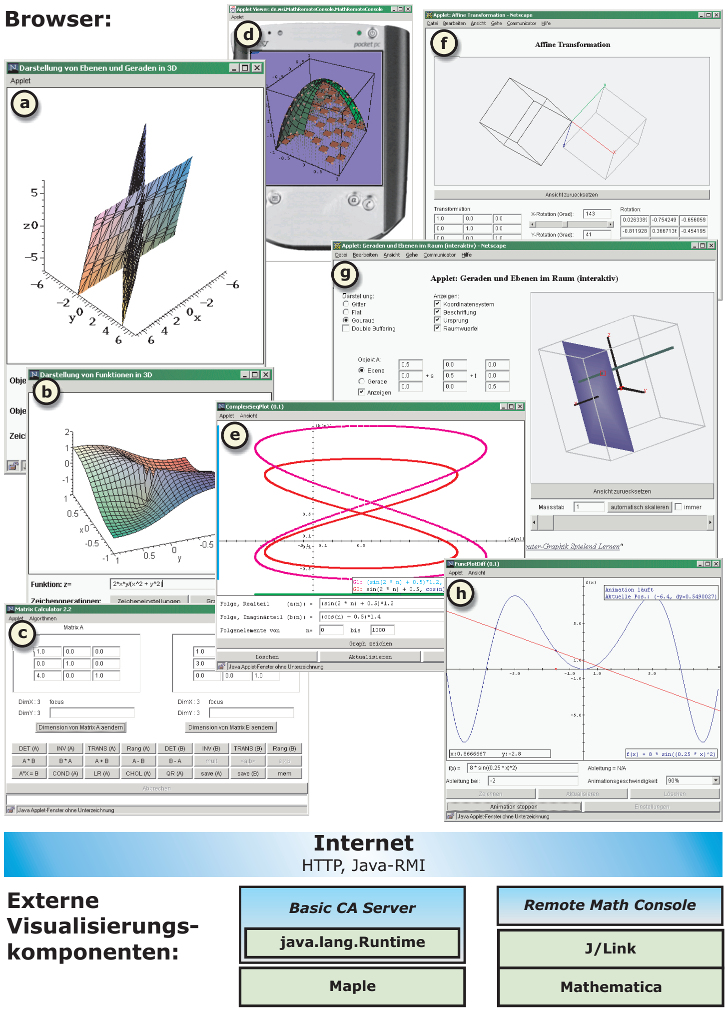

Abbildung 13.2: Einige Beispiele der *Mathe-Tools*-Werkzeugsammlung

spielsweise mehrere Ableitungen der reellen Funktion $x^3 - x^2 + x - 1$, die durch Drücken des Computer-Symbols direkt durch das entsprechende Applet visualisiert werden. Die Konfigurierung der Applets erfolgt jeweils durch entsprechende Parameter des *Applet*-Tags innerhalb der zugehörigen *HTML*-Seite.

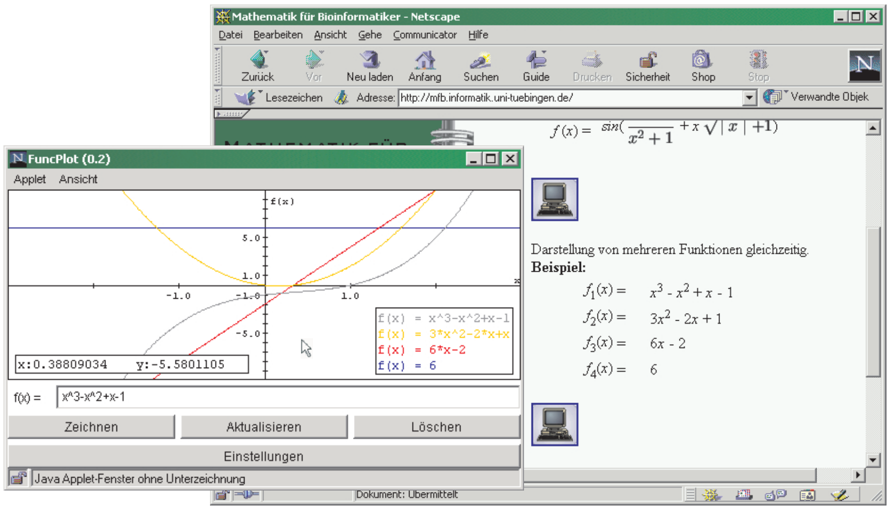

Abbildung 13.3: Starten eines *Mathe-Tools*-Applets in einer vordefinierten Konfiguration

Der Vorteil gegenüber der Hinterlegung entsprechender statischer Bilder liegt darin, dass die Benutzer die Beispiele innerhalb der geöffneten Applets verändern und das entsprechende Verhalten der mathematischen Konzepte beobachten können.

Wie bereits erwähnt, lassen sich die *Mathe-Tools*-Applets in drei Kategorien einteilen, die in den folgenden Abschnitten vorgestellt und miteinander verglichen werden.

13.2.1 Visualisierung in Java

Die Applets dieser Kategorie implementieren selbst die erforderlichen Berechnungen und Visualisierungsalgorithmen und haben daher keine externe Abhängigkeiten (vgl. Tabelle 13.1). Das in Abbildung 13.3 gezeigte *FuncPlot*-Applet zur Visualisierung von reellen Funktionen ist ein typisches Beispiel aus dieser Kategorie. Es visualisiert beliebige reelle Funktionen in einer Unbekannten über einem frei wählbaren Darstellungsbereich. Hierzu erfüllt das Applet die folgenden Aufgaben:

1. Parsen der Funktionsdefinition

2. Berechnung der relevanten Funktionswerte

3. Darstellung des entsprechenden Graphen

Die Applets der Kategorie 2 delegieren diese drei Aufgaben hingegen an eine externe Visualisierungskomponente und importieren dann die generierten Bilder in einem binären Format.

13.2.1.1 Implementierung

Die ersten beiden Aufgaben wurden durch das Java-Paket *MathExpression* und die dritte Aufgabe durch das Paket *Plot2d* bzw. *FuncPlot* und *SeqPlot* implementiert [BGV$^+$00]. Das *MathExpression*-Paket übernimmt das Parsen und numerische Auswerten der mathematischen Konzepte für die Applets der Kategorie 1 und definiert damit auch die intuitive und einheitliche Syntax zur Spezifikation von Funktionsausdrücken für alle *Mathe-Tools*-Applets. Zur numerischen Auswertung werden ausschließlich Methoden und Konstanten der Klasse *java.lang.Math*[3] verwendet. Eine Sonderstellung in dieser Beziehung nimmt die Implementierung des Matrizenrechners (siehe Abbildung 13.2c) ein, der anspruchsvollere numerische Verfahren (u.a. LR-Zerlegung, QR-Zerlegung, Laplace-Entwicklung) außerhalb des *MathExpression*-Pakets implementiert.

Das *Plot2d*-Paket fasst alle generelle Funktionalität zum Plotten in einer zweidimensionalen Ebene zusammen. Es wurde bewusst darauf verzichtet, die mächtige Java 2D API[4] des Java 2 Standards zu verwenden, da diese in den meisten Browsern (z.B. Internet Explorer 5.x und Netscape 4.x) nur nach vorheriger Installation des Java-Plug-Ins lauffähig wäre. Die Pakete *FuncPlot* bzw. *SeqPlot* erweitern die Visualisierungsfunktionalität des *Plot2d*-Pakets hinsichtlich Graphen von Funktionen bzw. Folgen und Reihen.

Die Applets der Kategorie 1, die mathematische Konzepte innerhalb des $\mathbb{R}^3$ visualisieren (vgl. Tabelle 13.1), wurden mit Hilfe einer Java-3D-Bibliothek realisiert, die im Rahmen des *Computer Graphik spielend Lernen*-Projekts am Lehrstuhl für Graphisch-Interaktive Systeme der Universität Tübingen entwickelt wurde [Han]. Diese Bibliothek wurde unter Java 1.0 realisiert und ist somit in den gängigen Web-Browsern direkt lauffähig.

13.2.1.2 Eigenschaften

Bei Applets der Kategorie 1 sind die darzustellenden mathematischen Werte dem Applet explizit verfügbar. Dies erlaubt die Realisierung der folgenden Funktiona-

[3] $+, -, \cdot, /$, pow(), cos(), acos(), sin(), asin(), tan(), atan(), exp(), ln(), sqrt(), abs(), floor(), round(), π, e

[4] http://java.sun.com/products/java-media/2D/index.html

litäten in Realzeit[5]:

1. Modifikation der Parameterwerte
2. Skalierung
3. Freie Wahl des Darstellungsbereichs (Zoom)
4. Möglichkeit der schrittweisen Konstruktion einer Szene und Animation
5. Werteabtastung des Darstellungsbereichs mit der Maus

Modifikation der Parameterwerte Die Parameterwerte eines mathematischen Konzepts können interaktiv angepasst werden. Innerhalb des *FuncPlot*-Applets kann beispielsweise die bereits erwähnte Funktion $x^3 - x^2 + x - 1$ leicht zu $\sin(x^3 - x^2 + x - 1)$ verändert werden, wobei der neue Graph auf Wunsch zusätzlich zu dem bestehenden Graphen in die Szene mit aufgenommen wird. So können leicht Phänomene wie Dehnung, Spiegelung usw. erfahren werden. Eine zusätzliche Art der interaktiven Parametrierung wird durch das Applet *Affine Transformation* (siehe Abbildung 13.2g) unterstützt, das beispielsweise die interaktive Drehung von Objekten im Raum durch entsprechende Schiebebalken unterstützt und gleichzeitig die entsprechenden Transformationsmatrizen berechnet und anzeigt.

Skalierung Die Skalierung des Darstellungsbereichs erfolgt i.d.R. einfach durch Vergrösserung/Vekleinerung des Applet-Fensters, woraufhin die Abbildung von realen Koordinaten auf Applet-Koordinaten entsprechend angepasst wird.

Freie Wahl des Darstellungsbereichs Der Darstellungsbereich definiert ein Sichtfenster auf die Visualisierung eines mathematischen Konzepts. Er kann über entsprechende GUI-Komponenten oder über Mausbewegungen innerhalb einer Maximal- bzw. Minimalausdehnung frei festgelegt werden. Die Minimalausdehnung beschränkt hierbei die Zoomauflösung einer Visualisierung.

Animation Das *SeqPlot*-Applet zur Visualisierung von Folgen und Reihen sowie das Applet zur Differentiation (siehe Abbildung 13.2h) unterstützen eine schrittweise Konstruktion bzw. Animation einer Szene. Innerhalb des *SeqPlot*-Applets kann die Berechnung einzelner Folgenglieder und insbesondere von entsprechenden Teilfolgen schrittweise dargestellt und so eine Szene langsam aufgebaut werden. Das Differentiations-Applet animiert die beiderseitige Berechnung

[5] Unter „Realzeit“ wird in diesem Kapitel „ohne merkliche Verzögerung für den Benutzer“ verstanden.

des Grenzwertes des Differenzenquotienten und zeigt jeweils die entsprechenden Sekanten, die sich im Laufe der Animation u.U. zur gemeinsamen Ableitung vereinigen.

Werteabtastung Aufgrund der durch die aktuelle Skalierung festgelegten Beziehung zwischen realen Koordinaten und Applet-Koordinaten kann leicht eine von der aktuellen Mausposition abhängige Übersetzung von Applet-Koordinaten in reale Koordinaten erfolgen und in die Szene eingeblendet werden (siehe Abbildung 13.3 unten links). Dadurch können beispielsweise die Funktionswerte des Graphen einer Funktion abgetastet werden, wobei jeweils die realen Funktionswerte angezeigt werden. Ebenso können die Koordinaten von Schnittpunkten und Ähnlichem einfach durch Anfahren mit der Maus schnell ermittelt werden.

Entscheidend ist bei all diesen Eigenschaften, dass sie in Realzeit verfügbar sind und ein interaktiver Zugang zu der entsprechenden Funktionalität erreicht wird. Dies ermöglicht den spielerischen Umgang mit unterschiedlichen Konfigurationen und Parameterwerten und die Erfahrung der entsprechenden Auswirkungen.

Falls eine Szene durch eine externe, über Internet eingebundene Visualisierungskomponente erfolgt, sind die Funktionalitäten 1-3 nur mit erheblichem Netzwerkverkehr möglich, da für jede Parameteränderung die Szene als Binärdatei nicht nur neu erstellt, sondern auch komplett über das Netzwerk angefragt werden muss. Die Werteabtastung mit der Maus könnte, wenn überhaupt, nur mit erheblichem Aufwand realisiert werden.

Der Nachteil dieses Ansatzes liegt im Aufwand für die Implementierung der numerischen Verfahren und Visualisierungen, der trotz des entwickelten flexiblen objektorientierte Designs besteht. Hinzu kommt, dass diese Verfahren i.d.R. nicht mit denen spezialisierter Visualisierungssysteme (z.B. [Fat92, ABK95]) oder kommerzieller Computeralgebrasysteme (z.B. Maple oder Mathematica) konkurrieren können.

13.2.2 Integration externer Visualisierungskomponenten

In diesem Kapitel wird die Integration der Computeralgebrasysteme (CA-Systeme) Maple und Mathematica in die MFB-Homepage beschrieben. Zielsetzung dieser Integration ist es, eine intuitive, homogene, Internet-basierte Zugangsschicht zu realisieren, die bestimmte Funktionalitäten dieser Systeme online verfügbar macht. Die Parametrierung der jeweiligen Teilfunktionalitäten soll über speziell angepasste GUIs auch unerfahrenen Benutzern intuitiv möglich sein. Insbesondere soll die Abbildung von Funktionalität auf entfernte Maple- oder

Mathematica-Systeme für den Benutzer transparent geschehen und keine lokale Maple- oder Mathematica-Installation voraussetzen.

13.2.2.1 Implementierung

Wie durch die Abbildungen 9.2 (S. 154) und 13.2 angedeutet, unterscheidet sich die Integration von CA-Systemen in Internet-basierte Lehr-/Lernumgebungen nicht wesentlich von der entsprechenden Integration industrieller Geräte (vgl. Abbildung 9.2). Die CA-Systeme werden ebenfalls durch eine Java-RMI-Middleware-Schicht gekapselt, die bestimmte Funktionalität der CA-Systeme nach außen weiterreicht. Zur Kommunikation mit den CA-Systemen werden jedoch anstatt von Feldbusprotokollen CA-spezifische Applikationsprotokolle verwendet.

Integration des Mathematica-Kernels Das Mathematica-System enthält die spezielle Werkzeugsammlung *J/Link*, die direkt den Zugriff auf den Mathematica-Kernel durch Java-Applikationen unterstützt. Ein entsprechender Java-RMI-Server kann über *J/Link* bequem auf die Funktionalität des Mathematica-Kernels zugreifen und diese über entsprechende RMI-Schnittstellen Clients zur Verfügung stellen.

Im Falle des Applets zur Integration im $\mathbb{R}^3$ (siehe Abbildung 13.2d) wurde ein entsprechendes Mathematica-Skript auf dem Server hinterlegt. Wenn eine Anfrage zur Integration einer bestimmten Fläche an den Server gestellt wird, wird dieses Skript durch den Server entsprechend parametriert und auf dem Mathematica-Kernel ausgeführt. Als Resultat wird die berechnete GIF-Datei in den Dokumentenbaum des Web-Servers geschrieben, von wo sie durch den Client über einen HTTP-Request geladen und anschließend dargestellt wird.

Integration des Maple-Kernels Die Einbindung des Maple-Kernels geschieht letztendlich über die *java.lang.Runtime*-Klasse, die das Erzeugen neuer nativer Prozesse über entsprechende Betriebssystemaufrufe unterstützt. Parametriert wird der Maple-Aufruf über Kommandozeilenparameter, die durch den implementierten Maple-RMI-Server in Abhängigkeit von den jeweiligen Client-Anfragen generiert werden. Die generierten Bilddateien im PCX-Format werden ebenfalls zunächst von Maple im lokalen Dateisystem abgelegt und dann durch den RMI-Server als Resultat des RMI-Aufrufs an den Client zurückgegeben.

Da die Berechnung komplexer 3D-Szenen erhebliche Zeit in Anspruch nehmen kann, lohnt es sich, vor dem Maple-Aufruf zunächst Rechenzeit in Verteilungskonzepte und Load-Balancing zu investieren. Die implementierte Maple-Anbindung nutzt daher ein Java-basiertes Komponentensystem zur intelligenten Lastverteilung auf einem Pool von Workstations [WKE98, EKW00].

13.2.2.2 Eigenschaften

Bei Applets der Kategorie 2 sind die darzustellenden mathematischen Werte dem Applet nicht explizit verfügbar. Stattdessen stehen über eine Netzwerkverbindung mächtige, externe CA-Systeme zur Verfügung, die auf Anfrage binär kodierte Bilddaten für die mathematischen Werte liefern. Dieser Ansatz zeichnet sich durch die folgenden Eigenschaften aus:

1. Hohe Qualität der Visualisierungen
2. Einfache Erweiterbarkeit

Die graphischen Resultate von kommerziellen CA-Systemen und speziellen Visualisierungssystemen sind i.d.R. komplexer und von höherer Qualität als die durch Java-Applets erzeugten Visualisierungen. Zusätzlich stehen bei CA-Systemen neben numerischen auch symbolische Evaluationsmethoden zur Verfügung, was die korrekte Visualisierung von mathematischen Konzepten verbessern oder überhaupt erst ermöglichen kann (z.B. symbolische Integration/Differentiation). Prinzipiell liegt der Vorteil dieses Ansatzes darin, dass Teilfunktionalitäten der vollen Mächtigkeit derartiger Systeme in ein Internet-basiertes System eingebunden werden können.

Durch die im vorherigen Abschnitt beschriebenen Client/Server-Architekturen kann die zur Verfügung gestellte Funktionalität leicht erweitert werden, solange sie durch die integrierten CA-Systeme unterstützt wird. Es müssen lediglich entsprechende Skripte und/oder Schnittstellen an den Servern angemeldet und die zugehörigen graphischen Benutzerschnittstellen entworfen werden, die eine intuitive Parametrierung der neuen Funktionalität erlauben.

Der Hauptnachteil dieses Ansatzes ist der Verlust der Parametrierung der Szene in Realzeit. Jede Veränderung der Parameterwerte resultiert in der Generierung einer entsprechenden Bilddatei auf der Server-Seite und deren Transport über ein Netzwerk. Dieser Nachteil ist gerade in einer Lehr-/Lernumgebung nicht zu unterschätzen, da Systeme mit langen Reaktionszeiten kaum von den Studenten genutzt werden. Die anspruchsvolle Anbindung des Maple-Systems auf mehreren parallelen Workstations kann dieses Problem nicht beseitigen, da sie zwar die Zeit, die für die Generierung einer neuen Szene benötigt wird, reduziert, nicht aber die Zeit für den Netzwerktransfer.

Interaktive Elemente, wie die im vorigen Abschnitt beschriebene Werteabtastung mit dem Mauszeiger, lassen sich mit diesem Ansatz zunächst nicht realisieren.

Ebenso sind freie Animationen, wie die beschriebene Animation der Ableitung einer reellen Funktion, nicht realisierbar.

13.2.3 Hybride Visualisierung

Die vorangegangen Abschnitte haben gezeigt, dass die Applets der Kategorie 1 einfache Visualisierungen mit einem hohen Grad an Interaktionsmöglichkeit und die Applets der Kategorie 2 komplexe Visualisierungen mit wenig Interaktionsmöglichkeit realisieren. Der im Rahmen dieser Arbeit entwickelte Ansatz der *hybriden Visualisierung* [BCK00] kombiniert diese beiden Konzepte und ermöglicht so interaktive, hochqualitative Visualisierungen.

Die grundlegende Idee ist hierbei, bestimmte Funktionalität von Server-seitigen CA-Systemen in Client-seitige Visualisierungen zu integrieren. Die CA-Systeme liefern keine komplett gerenderten Szenen, sondern zusätzliche Informationen, die während der Darstellung einer Szene auf Client-Seite verwendet werden, um die Qualität einer Szene zu verbessern.

Als konkretes Beispiel wurde die Visualisierung von reellen Funktionen im $\mathbb{R}^2$ samt evtl. vorhandener Singularitäten gewählt. Dieses Applet basiert auf dem bereits vorgestellten *FuncPlot*-Applet (siehe Abschnitt 13.2.1) und integriert zusätzlich eine symbolische Singularitätenberechnung, die durch eine Maple-Server Anbindung realisiert wurde.

Die Berechnung von Singularitäten einer reellen Funktion ist eine komplexe Aufgabe, die von den meisten kommerziellen CA-Systemen zwar prinzipiell unterstützt wird, aber nicht in die Plot-Funktionalität integriert ist. Abbildung 13.4 zeigt beispielsweise das Ergebnis der Maple- bzw. Mathematica-*plot*-Funktion für zwei Funktionen, die Singularitäten aufweisen. Die links dargestellte Funktion $\sin(x + 2)/(x + 2)$ weist eine Singularität an der Stelle $x = -2$ auf, die nicht in der Visualisierung zu erkennen ist. Gravierender ist die Situation in der rechts dargestellten Funktion $1/\sin(1/x)$, die einen Häufungspunkt von Singularitäten um $x = 0$ aufweist, der ebenfalls nicht in der Visualisierung erkennbar ist.

Im Gegensatz dazu werden die vorhandenen Singularitäten dieser Funktionen durch das entsprechende *Mathe-Tools*-Applet (siehe Abbildung 13.5) korrekt dargestellt. Zum Einen werden die einzelnen Singularitäten durch entsprechend schraffierte Bereiche im Plottergebnis hervorgehoben, um den Betrachter darauf hinzuweisen, dass die Darstellung des Graphen in diesem Bereich nicht verlässlich ist. Je nach dem, ob einzelne Singularitäten oder ein Häufungspunkt von Singularitäten vorliegt, werden hierfür unterschiedliche Farben verwendet. Zum Anderen werden die berechneten Singularitäten in einem Textfeld explizit angegeben.

Die Hervorhebung von Singularitäten im Graphen einer Funktion ist insbesondere

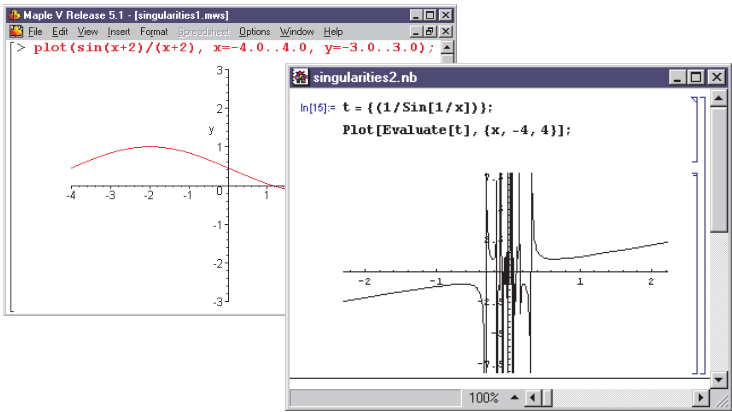

Abbildung 13.4: Plotten von Funktionen mit Singularitäten in Maple und Mathematica

im Kontext einer einführenden Mathematikvorlesung wichtig, da dort die angesprochene Zielgruppe i.d.R. nicht über das nötige Wissen verfügt, um diese Anormalitäten selbst zu erkennen und richtig einzuschätzen.

13.2.3.1 Implementierung

Die Implementierung basiert auf der bereits in Abschnitt 13.2.2.1 vorgestellten Maple-Anbindung. In diesem Fall werden jedoch keine Bilddateien angefragt, sondern eine Berechnung der Singularitäten der aktuellen Funktion angestoßen. Die Berechnung der Singularitäten ist durch ein speziell entwickeltes Maple-Skript definiert, das auf den Maple-Funktionen *discont()* und *fdiscont()* basiert (vgl. [BCK00]). Das Resultat wird jeweils in ein entsprechendes Java-Objekt verpackt und als Antwort auf eine Java-RMI-Anfrage an den Client zurückgegeben. Der Client bezieht diese Informationen dann in der beschriebenen Art und Weise in die Darstellung des Graphen der Funktion mit ein.

13.2.3.2 Eigenschaften

Die Funktionalität des *Mathe-Tools*-Applets zur Visualisierung von reellen Funktionen mit Singularitäten stellt eine Verbesserung bestehender Funktionenplotter dar, die Singularitäten meist völlig ignorieren. Zusätzlich ermöglicht das Applet eine Werteabtastung mit dem Mauszeiger sowie freie Skalierung wie

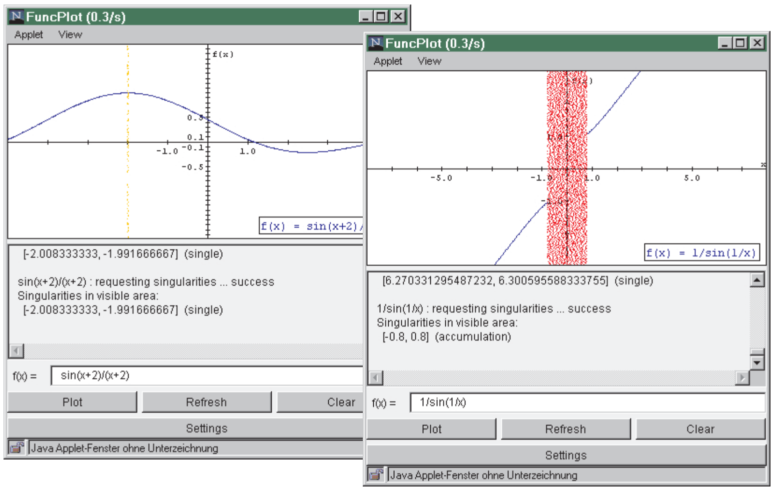

Abbildung 13.5: Plotten von Funktionen mit Singularitäten durch eine Erweiterung des *FuncPlot*-Applets

bei den Applets der Kategorie 1. Die Reparametrierung ist i.d.R. schneller als bei Applets der Kategorie 2, da nur ein kleines, wenige Byte großes Java-Objekt über das Netzwerk übertragen werden muss. Weiterhin bleiben die allen *Mathe-Tools*-Applets gemeinsamen Eigenschaften erhalten, so dass der Funktionenplotter insbesondere in einem Standard-Web-Browser ohne weitere Software-Anforderungen, abgesehen von einem Internet-Zugang, ausgeführt werden kann.

13.3 Zusammenfassung

Im Rahmen des Projekts *Mathematik für BioInform@tiker* wurde ein interaktives, Internet-basiertes Kursbuch mit Anbindung einer Volltextsuche, eines schwarzen Brettes, einer Chat-Möglichkeit und der *Mathe-Tools*-Werkzeugsammlung realisiert. Die *Mathe-Tools*-Sammlung besteht aus insgesamt 17 Java-Applets zur Visualisierung mathematischer Konzepte. Diese können beispielsweise direkt aus dem Kursbuch heraus gestartet werden, um die dort beschriebenen Konzepte zu visualisieren. Die Visualisierungen können dann jeweils interaktiv parametriert werden, um das Verhalten eines bestimmten Konzeptes näher zu erforschen.

Es werden drei Kategorien von Applets unterschieden. Applets der ersten Kategorie berechnen die jeweilige Visualisierung innerhalb des Clients direkt auf den zu visualisierenden Werten. Funktionalitäten wie die Werteabtastung mit dem Mauszeiger, die auf diese Werte angewiesen sind, lassen sich leicht durch wiederverwendbare Software-Pakete realisieren und in die Clients integrieren. Die Visualisierungen sind i.d.R. relativ einfach gehalten (es werden z.B. keine Beleuchtungsmodelle für Objekte im Raum berechnet) und können daher in Realzeit, z.T. sogar über entsprechende Schiebebalken, verändert werden.

Die Applets der zweiten Kategorie integrieren externe CA-Systeme über eine Java-RMI-basierte Client/Server-Architektur in die *Mathe-Tools*-Sammlung. Die einzelnen Visualisierungen werden hierbei jeweils als entsprechende Graphikdateien angefordert und im Client dargestellt. Durch diesen Ansatz wird die vollständige Visualisierungsfunktionalität dieser Systeme online verfügbar und es können komplexe Phänomene ansprechend visualisiert werden. Die Reaktionszeit dieses Ansatzes ist jedoch recht hoch, da für jede Reparametrierung nicht nur eine neue Visualisierung berechnet, sondern die entsprechende Bilddatei auch über ein Netzwerk transferiert werden muss.

Um die Vorteile dieser beiden Ansätze zu vereinen, wurde das Konzept der *hybriden Visualisierung* entwickelt, das eine Client-seitige Berechnung der Visualisierung unter Einbeziehung von entfernten CA-Systemen realisiert. Dieser Ansatz erhält die Vorteile der Kategorie 1, z.B. die Werteabtastung mit dem Mauszeiger, und reduziert den Netzwerkverkehr zwischen Client und CA-System, da nur wenige integrale Daten anstatt von binären Graphikdateien transferiert werden müssen.

Als beispielhafte Anwendung wurde ein Funktionenplotter implementiert, der reelle Funktionen über einer Veränderlichen samt ihrer evtl. vorhandenen Singularitäten visualisiert. Hierbei werden nur die Singularitäten durch einen entfernten Maple-Server berechnet und in die Darstellung des Graphen im Client mit einbezogen.

Sämtliche Applets der entwickelten *Mathe-Tools*-Sammlung können direkt in den Standard-Web-Browsern ausgeführt werden und benötigen keine weiteren Software-Ressourcen, wie z.B. Plug-Ins oder lokale Verfügbarkeit von CA-Systemen.

Kapitel 14

Zusammenfassung

Im Rahmen des zweiten Teils dieser Arbeit wurden mehrere Konzepte zur Integration externer Komponenten in Internet-basierte Informationssysteme entwickelt und entsprechende Systeme im Umfeld der Internet-basierten Lehre implementiert. Unter externen Komponenten werden hierbei insbesondere industrielle Feldbus-Geräte, aber auch externe Softwarepakete verstanden.

Im Einzelnen wurden zunächst mit der Realisierung der Java CAN API, des *Java Fieldbus-based Control Frameworks* und der Systeme zur Einbindung der Computeralgebrasysteme offene Architekturen geschaffen, die dann für die Erstellung diverser Applikationen im Kontext der virtuellen Lehre verwendet wurden (𝔹9).

Die Java CAN API wurde als objektorientierte Schnittstelle zu CAN-Feldbussystemen entwickelt (𝔹1). Diese Programmierschnittstelle erlaubt die einfache Kommunikation mit beliebigen CAN- und insbesondere mit CANopen-Geräten durch Unterstützung der entsprechenden Protokolle. Die API repräsentiert die einzelnen CAN-Kommunikationsobjekte durch eine entsprechende Klassenhierarchie, angefangen bei einer einfachen Basisklasse für CAN-Kommunikation auf OSI-Schicht 2 bis hin zu komplexen Kommunikationsobjekten zum Datentransfer variabel großer Datentypen über das CANopen-Protokoll auf Schicht 7. Einzelne Kommunikationsobjekte unterschiedlicher CAN-Protokollschichten können direkt instanziiert, parametriert und abgeschickt werden. Ebenso kann das asynchrone Auftauchen bestimmter CAN-Nachrichten auf einfache Weise überwacht werden. Ferner unterstützt die API den Zugriff auf CANopen-spezifische Netzwerkmanagementfunktionalität.

Diese API stellt die Basis für einen großen Teil der im Rahmen dieser Arbeit entwickelten Prototypen dar, z.B. die Integration einer CAN-basierten Automatisierungsanlage in ein Internet-basiertes Fernwartungssystem (siehe Abschnitte 5.6

und 7.11) oder die Integration einer CAN-basierten Werkstückvereinzelung in eine Internet-basierte Lehr-/Lernumgebung (siehe Abschnitt 11.4).

Das *Java Fieldbus-based Control Framework* (*JFCF*) realisiert ein objektorientiertes Software-Framework zur einfachen Realisierung von feldbusbasierten Gerätesteuerungen in Java ($\mathbb{B}3$). Um das Framework möglichst unabhängig von den verwendeten Feldbusprotokollen zu halten, wurde auf eine konsequente Trennung zwischen Schnittstellen und Implementierung geachtet. Dieser Ansatz erlaubt die Realisierung portabler Steuerungsklassen gemäß dem WOCRAC-Paradigma und bereitet zusätzlich den Weg für eine zukünftige Integration von Konzepten der *Real-Time Specification for Java*. Aufgrund der konsequenten Realisierung als Java-Framework konnten die für Java typischen Funktionalitäten, wie beispielsweise dynamisches Laden von Klassen, in das Framework integriert werden, was eine Ausführung dynamisch geladener und instanziierter Steuerungsklassen erlaubt.

Um die Einsetzbarkeit von *JFCF* zu evaluieren, wurden zwei Fallstudien durchgeführt, die beide die Java CAN API als Feldbuskommunikationsschnittstelle verwenden. Die erste Fallstudie realisiert eine komplexe Steuerung einer Automatisierungsanlage, die aus vier nebenläufigen Tasks für die Gerätekomplexe Werkstückvereinzelung, Transfersystem, Roboter und Liftsystem aufgebaut wurde. In der zweiten Fallstudie wurde ein Java-RMI-basiertes Client/Server-System realisiert, das die Fernsteuerung einer Werkstückvereinzelungsanlage erlaubt. Die Steuerungslogik kann hierbei als Konzept der *JFCF* zunächst innerhalb einer Simulation entwickelt, dann dynamisch via Internet geladen, instanziiert und schließlich innerhalb des Clients zur Ausführung gebracht werden.

Basierend auf *JFCF* und der Java CAN API wurden drei Übungsaufgaben realisiert und in eine Internet-basierte Lehr-/Lernumgebung eingebunden ($\mathbb{B}4$, $\mathbb{B}5$).

Die erste Übung behandelt das CANopen-Geräteprofilkonzept und erlaubt den interaktiven Zugriff auf beliebige entfernte CANopen-Geräte über deren Geräteprofil. Realisiert wird diese Funktionalität durch das *Java Remote CAN Control* (*JRCC*) System, das mit Hilfe der Java CAN API implementiert wurde. Es erlaubt den direkten lesenden und schreibenden Zugriff auf beliebige Geräteprofileinträge von beliebigen CANopen-Geräten und den Zugriff auf CANopen-Managementfunktionalität ($\mathbb{B}2$).

Die zweite Übung stellt eine exemplarische Client/Server-Teleservicearchitektur vor. Die Studenten implementieren ein Applet-Client gegen eine Java-RMI-Schnittstelle eines entfernten Servers, der den Zugriff auf eine Leuchtschrift unterstützt. Zustandsänderungen der Leuchtschrift können jeweils über eine steuerbare Internet-Kamera oder direktes Feedback des Servers überwacht werden. Die Studenten lernen im Rahmen dieser Aufgabe nicht nur Grundlagen objektorientierter Middleware, sondern auch die Besonderheiten im Zugriff auf reale Geräte kennen.

Im Rahmen der dritten Aufgabe entwickeln die Studenten zunächst eine objektorientierte Steuerungsklasse für eine Werkstückvereinzelungsanlage innerhalb eines Simulations-Applets. Die entwickelte Steuerungsklasse kann dann in einem zweiten Schritt ohne Modifikationen über ein weiteres Applet auf der realen Werkstückvereinzelung getestet werden. Diese Funktionalität ist möglich, da die Steuerungsklassen jeweils als *JFCF*-Konzept entwickelt und ausgeführt werden.

Alle drei Übungen sind rund um die Uhr zugänglich und erfordern keine menschliche Interaktion auf der Server-Seite ($\mathbb{B}$6, $\mathbb{B}$7). Zum Zugriff auf die entfernten Anlagen ist jeweils nur ein Standard-Web-Browser notwendig. Insbesondere werden keine weiteren Plug-Ins oder Software-Erweiterungen vorausgesetzt ($\mathbb{B}$8).

Im Rahmen des Projekts *Mathematik für BioInform@tiker* wurde ein interaktives, Internet-basiertes Kursbuch mit Anbindung einer Volltextsuche, eines schwarzen Brettes, einer Chat-Möglichkeit und der *Mathe-Tools*-Werkzeugsammlung realisiert ($\mathbb{C}$1). Die *Mathe-Tools*-Sammlung besteht aus insgesamt 17 Java-Applets zur Visualisierung mathematischer Konzepte ($\mathbb{C}$3), die teilweise externe Computeralgebrasysteme über zwei Client/Server-Architekturen mit einbeziehen ($\mathbb{C}$1). Diese können beispielsweise direkt aus dem Kursbuch heraus gestartet werden, um die dort beschriebenen Konzepte zu visualisieren. Die Visualisierungen können dann jeweils interaktiv parametriert werden, um das Verhalten eines bestimmten Konzeptes näher zu erforschen.

Es wurden drei Kategorien von Applets unterschieden und bewertet ($\mathbb{C}$2): Visualisierung in Java, Import von Bildern aus Computeralgebrasystemen und Visualisierung in Java unter Einbeziehung von Computeralgebrasystemen (hybride Visualisierung).

Insbesondere der entwickelte Ansatz der hybriden Visualisierung erlaubt die Kombination der Vorteile der anderen beiden Ansätze. Als Anwendung dieses Konzeptes wurde ein Funktionenplotter implementiert, der reelle Funktionen über einer Veränderlichen samt ihrer evtl. vorhandenen Singularitäten visualisiert. Hierbei werden nur die Singularitäten durch einen entfernten Maple-Server berechnet und in die Darstellung des Graphen im Client mit einbezogen.

Sämtliche Applets der entwickelten *Mathe-Tools*-Sammlung können direkt in den Standard-Web-Browsern ausgeführt werden und benötigen keine weiteren Software-Ressourcen, wie z.B. Plug-Ins oder lokale Verfügbarkeit von CA-Systemen.

Teil III

Zusammenfassung der Ergebnisse

Kapitel 15

Ergebnisse

Im Rahmen dieser Dissertation wurden innovative Konzepte zur Integration von heterogenen Informationsquellen und -senken in Internet-basierte Informationssysteme entwickelt und entsprechende Prototypen implementiert. Als Informationsquellen werden hierbei neben Datenbanken und Dateisystemen insbesondere Geräte und Softwarepakete verstanden. Die entwickelten Prototypen entstammen den Gebieten Fernwartung von Automatisierungsanlagen, Integration von industriellen Geräten in die Internet-basierte Lehre und Integration von Computeralgebrasystemen in einen Online-Mathematikkurs.

Der innovative Kern dieser Arbeit liegt in dem Konzept der Verwaltung von heterogenen Daten als serialisierte Zeichenketten in Form von *XML*-Fragmenten mit assoziierten Java-Klassen und der für diesen Ansatz entwickelten universellen Infrastruktur des INSIGHT-Systems. Für die Infrastruktur treten sämtliche Daten unabhängig von den jeweiligen Datentypen immer nur als *XML*-Knoten oder direkt als Unicode-Strings auf, die alle auf dieselbe Weise verwaltet werden können. Die jeweiligen datentypspezifischen Funktionen und Informationen sind jeweils über die durch *XJM*-Referenzen identifizierten Java-Klassen eindeutig bestimmt.

Entscheidend ist hierbei, dass die *XJM*-Referenz auch die Internet-Adresse der referenzierten Klasse enthält und diese dadurch global eindeutig identifiziert. Ein so spezifiziertes Paar aus *XML*-Fragment und Java-Klasse stellt daher eine wohldefinierte Repräsentation des serialisierten Wertes dar. Die in der *XJM*-Referenz enthaltene Internet-Adresse erlaubt weiterhin, dass die referenzierte Klasse zur Laufzeit eines Systems geladen werden kann und insbesondere nicht zur Compile-Zeit des jeweiligen Kern-Systemes lokal vorhanden sein muss.

Dieser Ansatz ermöglicht es, externe *XML*-Beschreibungen, z.B. in Form von *DeMML*- oder *XJML*-Dokumenten, für Systemkonfigurationen hinsichtlich des

Zusammenspiels heterogener Daten sinnvoll einzusetzen, da sämtliche semantische Informationen über die verwendeten Datentypen dynamisch über eindeutig referenzierte Java-Klassen verfügbar sind.

Die durch das INSIGHT-System entwickelte universelle, Internet-basierte Infrastruktur, über die sämtliche derartig erfasste Daten direkt verwaltet werden können, spiegelt das Potential dieses Ansatzes wider. Im Einzelnen kann auf folgende Funktionalitäten zurückgegriffen werden:

- Datenbankanbindung inklusive einer *XML*-basierten Query-Schnittstelle via Internet
- Visualisierung über *XSL*-Dokumente und GUI-Komponenten via Internet
- Bewertung und Vergleich der heterogenen Daten durch das *XJM_Eval*-System via Internet

Ein Resultat dieses Ansatzes kann, wie in Teil 1 dieser Arbeit gezeigt, ein universelles Fernwartungssystem für industrielle Anlagen sein. Die Anlagen werden hierbei jeweils durch entsprechende externe *DeMML*-Dokumente beschrieben, wobei insbesondere die vorhandenen Gerätedaten und ihre assoziierten Java-Klassen festgelegt werden. Erfasste Anlagenzustände können in Form von *XML*-Dokumenten über die Internet-basierte Datenbankanbindung zentral verwaltet und ausgewertet werden. Insbesondere können behobene Fehlerzustände innerhalb einer weltweit zentralen Fehlerdatenbank dazu verwendet werden, eine Funktion zur Unterstützung von Fehlerdiagnosen zu realisieren, indem eine Ähnlichkeitssuche für einen unbekannten Fehlerzustand in diesem Datenbestand durchgeführt wird. Das gefundene ähnlichste Dokument kann im besten Fall direkt mit dem unbekannten Fehler „identisch“ sein und ihn so eindeutig identifizieren oder aber zumindest Hinweise auf mögliche Ursachen geben.

Für diese Ähnlichkeitssuche muss jeweils auf die tatsächlichen Daten entsprechend ihrem Datentyp zugegriffen werden können, was über die assoziierten Klassen ermöglicht wird. Konkret erlaubt das *XJM_Eval*-System die flexible Zuordnung dieser Klassen zu den einzelnen serialisierten Daten und deren flexible Kombination mit entsprechenden Vergleichsmethoden. Die so realisierte Funktionalität ermöglicht eine Ähnlichkeitsbewertung für beliebige *XML*-Dokumente. Ferner sind *XJML*-basierte Auswertungen des Inhalts einzelner Dokumente leicht zu realisieren.

Die unterschiedlichen Visualisierungsmöglichkeiten der Infrastruktur erlauben eine flexible und interaktive Generierung von dedizierten Sichten auf den jeweiligen Zustand einer entfernten Anlage, indem beispielsweise bestimmte Informationen aggregiert und andere verbogen werden. Diese Funktionalität ist essentiell, um einem menschlichen Experten eine flexible Einsicht in einen entfernten Anlagenzustand zu gewähren.

Die Komponente, die die Serialisierung von hardwarespezifisch kodierten Gerätedaten in das universelle *XML*-Format implementiert, wurde durch das *XML*-basierte Software-Framework *DeviceInvestigator* realisiert. Konfiguriert wird *DeviceInvestigator* durch ein externes *DeMML*-Dokument, das sämtliche zu aggregierenden Gerätedaten identifiziert und die entsprechenden Hardwarezugriffsklassen über *XJM*-Referenzen assoziiert. *DeviceInvestigator* unterstützt nach einer Initialisierungsphase, in der die unterschiedlichen Hardwarezugriffsklassen dynamisch über Internet geladen, instanziiert und in entsprechenden Index-Strukturen abgelegt werden, den lesenden Zugriff auf den momentanen Anlagenzustand durch Generierung eines entsprechenden *DeMML*-Dokuments, das sämtliche aktuelle Gerätedaten in serialisierter Form enthält.

Zusätzlich vermittelt *DeviceInvestigator* den schreibenden Zugriff auf einzelne Gerätedaten, sofern die assoziierten Hardwarezugriffsklassen dies unterstützen. *DeviceInvestigator* kann weiterhin als eigenständiges Monitoring-System betrieben werden, das die generierten Anlagenzustandsdokumente direkt in einer entfernten *XML*-Datenbank oder dem lokalen Dateisystem ablegt.

Alle Teile der INSIGHT-Infrastruktur sind konsequent in Java entwickelt worden, wobei auf Server-Seite eine entsprechende virtuelle Maschine in der Version 1.1 genügt. Sie erreicht dadurch einen hohen Grad an Portabilität, so dass beispielsweise die *DeviceInvestigator*-Komponente auch auf einem kostengünstigen Embedded-System ausgeführt werden könnte. Diese Option ist insbesondere auch deshalb interessant, da durch die integrierte Datenbankanbindung via HTTP die erfassten Daten nicht auf dem ressourcenschwachen Embedded-System gehalten werden müssen.

Die Wahl der Programmiersprache Java war neben der Plattformunabhängigkeit und der Vielzahl von verfügbaren Bibliotheken insbesondere auch deshalb angezeigt, weil das Java Reflection Konzept die Realisierung der dynamischen Zuordnung von serialisierten Daten zu Java-Klassen überhaupt erst als sinnvoll erscheinen lässt.

Die Qualität der entwickelten Konzepte und Implementierungen wurde anhand von mehreren Fallstudien bestätigt, die die Integration von feldbusbasierten Anlagen und von Magnetresonanztomographen der Firma Siemens in Internet-basierte Informationssysteme umfasst.

Die im zweiten Teil dieser Arbeit beschriebenen grundlegenden Architekturen der Java CAN API, des *Java Fieldbus-based Control Frameworks* und des *Java Remote CAN Control* Systems stellen hierbei Bausteine und Spin-Offs des CAN-spezifischen Vorgängers CANINSIGHT des INSIGHT-Systems dar, das durch die

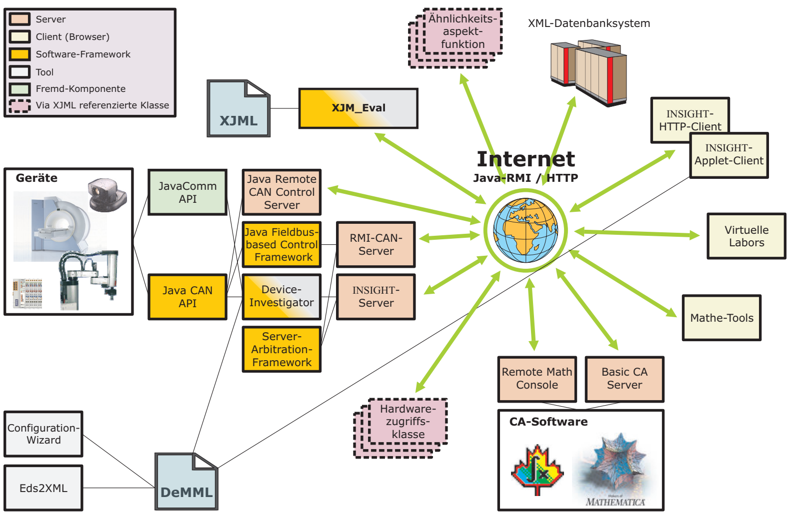

Abbildung 15.1: Übersicht über die realisierten Architektur-Komponenten

Integration des *XJM*-Konzepts zu einem universell für beliebige Datentypen und Datenquellen einsetzbaren System verallgemeinert wurde. Abbildung 15.1 gibt noch einmal einen Überblick über das Zusammenspiel der im Rahmen dieser Arbeit entwickelten Komponenten und Konzepte.

Die entwickelten Internet-basierten Laborübungen und Visualisierungen mathematischer Konzepte werden nachhaltig an der Fachhochschule Reutlingen und Universität Tübingen eingesetzt und erlauben den entfernten Zugriff auf industrielle Geräte bzw. Computeralgebrasysteme aus Internet-basierten Lehr-/Lernumgebungen.

Im Rahmen dieser Dissertation erschienen die folgenden wissenschaftlichen Veröffentlichungen (in chronologischer Reihenfolge):

1. GRUHLER, GERHARD, WOLFGANG KÜCHLIN, GERD NUSSER und DIETER BÜHLER: *Internet-basiertes Labor für Automatisierungstechnik und Informatik*. In: SCHMID, DIETMAR (Herausgeber): *Virtuelles Labor: Ihr Draht in die Zukunft*, Seiten 27–36, Ellwangen, Deutschland, Oktober 1999. R. Wimmer Verlag.

2. BÜHLER, DIETER, GERD NUSSER, GERHARD GRUHLER und WOLFGANG KÜCHLIN: *A Java Client/Server System for Accessing*

Arbitrary CANopen Fieldbus Devices via the Internet. South African Computer Journal, (24):239–243, November 1999.

3. GRUHLER, GERHARD, GERD NUSSER, DIETER BÜHLER und WOLFGANG KÜCHLIN: *Teleservice of CAN Systems via Internet*. In: *Proc. of the 6th International CAN Conference (ICC 99)*, Torino, Italy, November 1999. CAN in Automation (www.can-cia.com).

4. BÜHLER, DIETER, WOLFGANG KÜCHLIN, GERHARD GRUHLER und GERD NUSSER: *The Virtual Automation Lab - Web Based Teaching of Automation Engineering Concepts*. In: *Proc. of the 7th IEEE International Conference on the Engineering of Computer Based Systems (ECBS 2000)*, Edinburgh, Schottland, April 2000. IEEE Computer Society Press.

5. SCHIMKAT, RALF-DIETER, GERD NUSSER und DIETER BÜHLER: *Scalability and Interoperability in Service-Centric Architectures for the Web*. In: *Proc. of the 11th International Workshop on Database and Expert Systems Applications (DEXA 2000)*, London, UK, September 2000. IEEE Computer Society Press.

6. BÜHLER, DIETER und GERD NUSSER: *The Java CAN API – A Java Gateway to Fieldbus Communication*. In: *Proc. of the 2000 IEEE International Workshop on Factory Communication Systems (WFCS 2000)*, Porto, Portugal, September 2000. IEEE.

7. BÜHLER, DIETER und GERHARD GRUHLER: *XML-based Representation and Monitoring of CAN Devices*. In: *Proc. of the 7th International CAN Conference (ICC 2000)*, Amsterdam, The Netherlands, Oktober 2000. CAN in Automation (www.can-cia.com).

8. BÜHLER, DIETER, CORINNE CHAUVIN und WOLFGANG KÜCHLIN: *Plotting Functions and Singularities with Maple and Java on a Component-based Web Architecture*. In: GANZHA, V. G., E. W. MAYR und E.V. VOROZHTSOV (Herausgeber): *Computer Algebra in Scientific Computing – CASC 2000*, Samarkand, Usbekistan, Oktober 2000. Springer-Verlag.

9. BÜHLER, DIETER: *The CANopen Markup Language – Representing Fieldbus Data with XML*. In: *Proc. of the 26th Annual Conference of the IEEE Industrial Electronics Society (IECON 2000)*, Nagoya, Japan, Oktober 2000. IEEE.

10. BÜHLER, DIETER und WOLFGANG KÜCHLIN: *Remote Fieldbus System Management with Java and XML*. In: *Proc. of the IEEE International Sym-*

posium on Industrial Electronics (ISIE 2000), Puebla, Mexiko, Dezember 2000. IEEE.

11. BÜHLER, DIETER, GERD NUSSER, WOLFGANG KÜCHLIN und GERHARD GRUHLER: *The Java Fieldbus Control Framework - Object-oriented Control of Fieldbus Devices*. In: *Proc. of the 4th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 01)*, Magdeburg, Deutschland, Mai 2001.

12. GRUHLER, GERHARD, WOLFGANG KÜCHLIN, DIETER BÜHLER und GERD NUSSER: *Internet-based Lab Assignments in Automation Engineering and Computer Science*. In: SCHILLING, K., H. ROTH und O. ROESCH (Herausgeber): *Proc. of the International Worshop on Tele-Education in Mechatronics Based on Virtual Laboratories*, Ellwangen, Deutschland, Juli 2001. R. Wimmer Verlag.

13. BÜHLER, DIETER und WOLFGANG KÜCHLIN: *Flexible Similarity Assessment for XML Documents Based on XQL and Java Reflection*. In: MONOSTRI, LÁZLÓ, VÁNCZA JÓSEF und ALI MOONIS (Herausgeber): *Engineering of Intelligent Systems – Proc. of the 14th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEA/AIE 2001)*, Budapest, Ungarn, Juni 2001. Springer-Verlag (LNCS/LNAI).

14. NUSSER, GERD, DIETER BÜHLER, GERHARD GRUHLER und WOLFGANG KÜCHLIN: *Reality-driven Visualization of Automation Systems via the Internet based on Java and XML*. In: *Proc. of the 1st IFAC Conference on Telematics Applications in Automation and Robotics*, Weingarten, Deutschland, Juli 2001.

Kapitel 16

Ausblick

Die entwickelten Konzepte und Systeme können in viele Richtungen weiterentwickelt und vervollständigt werden. Wichtige Weiterentwicklungen wären beispielsweise:

- Integration eines Sicherheitskonzepts in die INSIGHT-Infrastruktur
- Personalisierung der INSIGHT-Infrastruktur
- Weitere Evaluation des *XJM_Eval*-Systems

Sicherheit In der momentanen Form unterstützt die INSIGHT-Infrastruktur keine Verschlüsselungs- oder Authentifizierungskonzepte. Dies ist für die industrielle Einsetzbarkeit des Systems jedoch eine Grundvoraussetzung. Ein Vorteil des INSIGHT-Systems ist, z.B. im Gegensatz zu SNMP, dass zur Integration der Daten in das Internet direkt die weitverbreiteten Protokolle HTTP und TCP/IP verwendet werden, für die entsprechende Software-Lösungen in großer Zahl verfügbar und unmittelbar einsetzbar sind.

Die Java-Programmiersprache unterstützt beispielsweise die Verwendung von *Secure Socket Layer* (SSL) Sockets für verschlüsselte RMI-Kommunikation. Ebenso sind für den Apache Web-Server kostenfrei SSL-Erweiterungen verfügbar, um eine Kommunikation über HTTPS zu realisieren.

Eine entsprechende Erweiterung der INSIGHT-Infrastruktur durch SSL-Verschlüsselung, Verwaltung von Benutzerprofilen und Authentifizierungskonzepten wurde in einer Vorab-Studie [Bey01] bereits untersucht und kann mit wenig Aufwand in INSIGHT integriert werden.

Personalisierung Um die INSIGHT-Infrastruktur zu einem Internet-Portal zu erweitern, müsste zusammen mit der Integration eines Sicherheitskonzeptes auch eine Verwaltung von Benutzerprofilen realisiert werden. Diese Benutzerprofile können dann nicht nur zur Authentifizierung eines Benutzers verwendet werden, sondern auch zur Verwaltung benutzerbezogener Sichten und Konfigurationen, z.B. bzgl. der zu verwendenden *XML*-Datenbankanbindung. Auf Server-Seite kann dies einfach dadurch geschehen, dass nicht mehr pro Anlage ein *DeMML*-Konfigurationsdokument verwaltet wird, sondern dass jedem Benutzer einer Anlage ein eigenes Konfigurationsdokument zugeordnet wird. Dieses Dokument könnte in einer bereits angebundenen *XML*-Datenbank abgelegt und auf Anfrage dynamisch geladen und zur Rekonfiguration der *DeviceInvestigator*-Komponente verwendet werden.

Für die Client-Seite müssten die entsprechenden *XSL*-Dokumente und/oder die GUI-Komponenten des Applet-Clients benutzerbezogen verwaltet werden.

Weitere Evaluation des *XJM_Eval*-Systems Das entwickelte Konzept zur Bewertung der strukturellen und insbesondere der inhaltsbezogenen Ähnlichkeit beliebiger *XML*-Dokumente sollte bzgl. Mächtigkeit und Effizienz weiter evaluiert werden.

Im Kontext der in Teil 1 beschriebenen Automatisierungsanlage wäre eine Integration von Pattern-Matching Algorithmen aus Computergraphiksystemen interessant, um auch den Grad der Übereinstimmung des Kamera-Bildes in die Ähnlichkeitsbewertung mit einzubeziehen. Hierzu müssten lediglich die entsprechenden *XJM*-Wrapper-Klassen erstellt und die jeweiligen Graphik-Bibliotheken angesprochen werden.

Die zusätzliche Integration von Audiosignalen könnten die Beschreibung eines Anlagenzustandes weiter vervollkommnen. Typische Phänomene sind hier z.B. zischende undichte Druckluftleitungen oder knarrende verschlissene Kugellager. Zur Bewertung der Ähnlichkeit von erfassten Audio-Frequenzmustern könnte beispielsweise mit minimalem Aufwand die vorgestellte Anbindung des Maple-Computeralgebrasystems über Internet verwendet werden.

Im Hinblick auf eine Steigerung der Effizienz des *XJM_Eval*-Systems erscheinen Parallelisierungskonzepte und insbesondere eine Implementierung über ein System von mobilen Agenten als attraktiv. Die Agenten-Lounge könnte hierbei beispielsweise die Ausführungsumgebung des *XJM_Eval*-Systems zur Verfügung stellen, während die Agenten über entsprechende *XJML*-Dokumente parametriert wären. Dieser Ansatz wäre dann besonders sinnvoll, wenn der zu durchsuchende Datenbestand auf Datenbanken an mehreren Standorten verteilt ist. Die einzelnen Agenten könnten sich selbstständig auf die involvierten Standorte verteilen und dort die Teilergebnisse lokal, ohne Netzwerkverkehr berechnen. Nach erfolgter Berechnung können sich einzelne Agenten wieder treffen, um die jeweiligen Teilergebnisse schrittweise zu einem Gesamtergebnis zu konsolidieren.

Anhang A

XML-Applikationen und Beispieldokumente

A.1 Die *Device Management Markup Language* DTD

```
<!-- Device Management Markup Language (DeMML) -->
<!-- (cf. http://www-sr.informatik.uni-tuebingen.de/Insight/DeMML.dtd) -->
<!-- Version 0.41-->
<!-- Date 01/14/2001 -->
<!-- Dieter Buehler (buehler@informatik.uni-tuebingen.de -->

<!-- Extension DTDs (optional) -->
<!ENTITY % DeviceML.dtd SYSTEM
  "http://www-sr.informatik.uni-tuebingen.de/Insight/DeviceML.dtd">
%DeviceML.dtd;

<!ENTITY % CANopenDeviceExtML.dtd SYSTEM
  "http://www-sr.informatik.uni-tuebingen.de/Insight/CANopenDeviceExt\
ML.dtd">
%CANopenDeviceExtML.dtd;

<!ENTITY % ImageDeviceExtML.dtd SYSTEM
  "http://www-sr.informatik.uni-tuebingen.de/Insight/ImageDeviceExtML\
.dtd">
%ImageDeviceExtML.dtd;

<!-- Element Declarations -->
<!ELEMENT DeMML (SystemConfiguration | Snapshot)>

<!-- SystemConfiguration -->
<!ELEMENT SystemConfiguration (FileInfo, StateInfo?, SystemInfo,
    (ImageDevice | CANopenDevice | Device)*)>
<!ELEMENT StateInfo (Description?)>
<!ATTLIST StateInfo StartOfQuery CDATA #IMPLIED
   EndOfQuery CDATA #IMPLIED>
```

```
<!ELEMENT SystemInfo (Description?, Location, (Contact)+)>
<!ELEMENT Location EMPTY>
<!ATTLIST Location Id CDATA #REQUIRED
   Country CDATA #REQUIRED
   Address CDATA #REQUIRED
   ServerIP CDATA #REQUIRED
   ServerName CDATA #IMPLIED>
<!ELEMENT Contact EMPTY>
<!ATTLIST Contact Id CDATA #REQUIRED
   FirstName CDATA #REQUIRED
   Surname CDATA #REQUIRED
   Email CDATA #IMPLIED
   Phone CDATA #IMPLIED
   Fax CDATA #IMPLIED
   Homepage CDATA #IMPLIED>

<!-- StateInformation -->
<!ELEMENT Snapshot (Description?, (ScannedDevice)*)>
<!ATTLIST Snapshot StartOfQuery CDATA #REQUIRED
   EndOfQuery CDATA #REQUIRED
   LocationId CDATA #REQUIRED
   SystemConfigurationCreationTime CDATA #REQUIRED>
<!ELEMENT ScannedDevice (LogEntry)*>
<!ATTLIST ScannedDevice DeviceId CDATA #REQUIRED>
<!ELEMENT LogEntry EMPTY>
<!ATTLIST LogEntry ParameterId CDATA #REQUIRED
   TimeStamp CDATA #IMPLIED
   Value CDATA #IMPLIED
   Valid (0|1) "1">
```

Listing A.1: *DeMML*-DTD

A.2 Die *Device Markup Language* DTD

```
<!-- Device Markup Language DeviceML -->
<!-- (cf. http://www-sr.informatik.uni-tuebingen.de/Insight/DeviceML.dtd) -->
<!-- Version 0.1 -->
<!-- Date 01/1/2001 -->

<!-- External DTDs -->
<!ENTITY % XJML.dtd SYSTEM
   "http://www-sr.informatik.uni-tuebingen.de/Insight/XJML.dtd">
%XJML.dtd;

<!-- Entity Declarations -->
<!ENTITY % parameterAtts "Name␣CDATA␣#IMPLIED
   DataType CDATA #IMPLIED
   LowLimit CDATA #IMPLIED
   HighLimit CDATA #IMPLIED
   Unit CDATA #IMPLIED
   AccessType (ro|wo|rw|rwr|rww|const) 'rw'
   Mute (0|1) '0'">

<!-- Element Type Declarations -->
<!ELEMENT Device (Description?, FileInfo?, DeviceInfo?,
```

```
    Parameters, DataAccess?, DeviceExt?)>
<!ATTLIST Device DeviceId CDATA #REQUIRED>
<!ELEMENT FileInfo (FileInfoExt?)>
<!ATTLIST FileInfo FileName CDATA #IMPLIED
    FileVersion CDATA #IMPLIED
    FileRevision CDATA #IMPLIED
    CreationTime CDATA #IMPLIED
    CreationDate CDATA #IMPLIED
    CreatedBy CDATA #IMPLIED
    Description CDATA #IMPLIED>
<!ELEMENT DeviceInfo (Vendor?, Product?, DeviceInfoExt?)>
<!ELEMENT Vendor (VendorExt?)>
<!ATTLIST Vendor Name CDATA #IMPLIED VendorId CDATA #IMPLIED>
<!ELEMENT Product (ProductExt?)>
<!ATTLIST Product Name CDATA #IMPLIED
    ProductId CDATA #IMPLIED
    Version CDATA #IMPLIED
    Revision CDATA #IMPLIED
    OrderCode CDATA #IMPLIED>
<!ELEMENT Description ANY>
<!ELEMENT Parameters (SupportedDataTypes?, ParameterCategory+,
    ParametersExt?)>
<!ELEMENT SupportedDataTypes (SupportedDataType)*>
<!ELEMENT SupportedDataType (Description?, SupportedDataTypeExt?)>
<!ATTLIST SupportedDataType DataTypeId CDATA #REQUIRED>
<!ELEMENT ParameterCategory (Description?, (Parameter | ParameterGroup)*)>
<!ATTLIST ParameterCategory Name CDATA #IMPLIED>
<!ELEMENT Parameter (Description?, DefaultValue?, ParameterValue?,
    DataAccess?, ParameterExt?)>
<!ATTLIST Parameter %parameterAtts;
    ParameterId CDATA #REQUIRED
    Monitoring CDATA #IMPLIED>
<!ELEMENT ParameterGroup (Description?, (Parameter*))>
<!ATTLIST ParameterGroup %parameterAtts;>
<!ELEMENT DefaultValue (#PCDATA)>
<!ELEMENT ParameterValue (#PCDATA)>
<!ATTLIST ParameterValue Valid CDATA #IMPLIED>
<!ELEMENT DataAccess (PortClass?, ParameterClass?)>
<!ELEMENT PortClass (JavaRef, (PortMethod)*)>
<!ATTLIST PortClass InitArgs CDATA #IMPLIED>
<!ELEMENT PortMethod (#PCDATA)>
<!ELEMENT ParameterClass (JavaRef)>
<!ATTLIST ParameterClass InitArgs CDATA #IMPLIED>
<!ELEMENT FileInfoExt EMPTY>
<!ELEMENT DeviceExt EMPTY>
<!ELEMENT VendorExt EMPTY>
<!ELEMENT ParametersExt EMPTY>
<!ELEMENT ParameterExt EMPTY>
<!ELEMENT DeviceInfoExt EMPTY>
<!ELEMENT ProductExt EMPTY>
<!ELEMENT SupportedDataTypeExt EMPTY>
```

Listing A.2: *DeviceML*-DTD

A.3 Die *CANopen Device Markup Language* DTD

```
<!-- CANopen Device Markup Language -->
<!-- (cf. www-sr.informatik.uni-tuebingen.de/Insight/CANopenDeviceML.dtd) -->
<!-- Version 0.1 -->
<!-- Date 01/02/2001 -->
<!-- Dieter Buehler (buehler@informatik.uni-tuebingen.de -->

<!-- External DTDs -->
<!ENTITY % DeviceML.dtd SYSTEM
  "http://www-sr.informatik.uni-tuebingen.de/Insight/DeviceML.dtd">
%DeviceML.dtd;

<!ENTITY % CANopenDeviceExtML.dtd SYSTEM
 "http://www-sr.informatik.uni-tuebingen.de/Insight/CANopenDeviceExtML.dtd">
%CANopenDeviceExtML.dtd;
```

Listing A.3: *CANopenDeviceML*-DTD

A.4 Die *CANopen Device Markup Language* Extension-DTD

```
<!-- CANopenDevice Markup Language Extension DTD -->
<!-- (cf. www-sr.informatik.uni-tuebingen.de/Insight/CANopenDeviceExtML.dtd) -->
<!-- Version 0.1 -->
<!-- Date 01/07/2001 -->
<!-- Dieter Buehler (buehler@informatik.uni-tuebingen.de -->

<!ELEMENT CANopenDevice (Device, CANopenDeviceExt)>
<!ELEMENT CANopenDeviceExt (LMT?, BaudRate?, BootMode?, DummyUsage?,
  ObjectLinks?, Comments?, (AdditionalNameValuePair)*)>
<!ATTLIST FileInfoExt ModificationTime CDATA #IMPLIED
  ModificationDate CDATA #IMPLIED
  ModifiedBy CDATA #IMPLIED>
<!ATTLIST DeviceInfoExt Granularity CDATA #IMPLIED
  DynamicChannelsSupported CDATA #IMPLIED
  GroupMessaging (0|1) "0"
  NrOfRXPDO CDATA #IMPLIED
  NrOfTXPDO CDATA #IMPLIED
  LSS_Supported (0|1) "0">
<!ELEMENT LMT EMPTY>
<!ATTLIST LMT ManufacturerName CDATA #IMPLIED
  ProductName CDATA #IMPLIED>
<!ELEMENT BaudRate EMPTY>
<!ATTLIST BaudRate rate10 (0|1) "0"
  rate20 (0|1) "0"
  rate50 (0|1) "0"
  rate100 (0|1) "0"
  rate125 (0|1) "0"
  rate250 (0|1) "0"
  rate500 (0|1) "0"
  rate800 (0|1) "0"
  rate1000 (0|1) "0">
```

```
<!ELEMENT BootMode EMPTY>
<!ATTLIST BootMode SimpleBootUpMaster (0|1) "0"
  SimpleBootUpSlave (0|1) "0"
  ExtendedBootUpMaster (0|1) "0"
  ExtendedBootUpSlave (0|1) "0"
  SimpleBootUp (0|1) "0"
  ExtendedBootUp (0|1) "0">
<!ELEMENT DummyUsage (DummySupport)*>
<!ELEMENT DummySupport EMPTY>
<!ATTLIST DummySupport DataTypeIndex CDATA #REQUIRED>
<!ATTLIST ParameterExt Index CDATA #REQUIRED
  Subindex CDATA #REQUIRED
  PDOMapping (0|1) '0'
  ObjectType (0|2|5|6|7|8|9) #IMPLIED>
<!ELEMENT ObjectLinks (ObjectLink)*>
<!ELEMENT ObjectLink (Link)*>
<!ATTLIST ObjectLink Index CDATA #REQUIRED>
<!ELEMENT Link EMPTY>
<!ATTLIST Link Target CDATA #REQUIRED>
<!ELEMENT Comments (Lines)*>
<!ELEMENT Lines (#PCDATA)>
<!ELEMENT AdditionalNameValuePair (AdditionalName, AdditionalValue)>
<!ELEMENT AdditionalName (#PCDATA)>
<!ELEMENT AdditionalValue (#PCDATA)>
```

Listing A.4: *CANopenDeviceML* Extension-DTD

A.5 Die *XML to Java Mapping Language* DTD

```
<!-- XML to Java Mapping Language -->
<!-- Version 0.1 -->
<!-- Date 11/12/2000 -->
<!-- Dieter Buehler (buehler@informatik.uni-tuebingen.de -->

<!-- Elements -->
<!ELEMENT XJML (Statement+)>
<!ELEMENT Statement (Purpose?, XMLDataRepository, (Mapping*), Evaluation)>
<!ATTLIST Statement Problem CDATA #IMPLIED>
<!ELEMENT Purpose ANY>
<!ELEMENT XMLDataRepository ( FileSystem | URL | Database )>
<!ATTLIST XMLDataRepository RootElement CDATA #IMPLIED>
<!ELEMENT FileSystem EMPTY>
<!ATTLIST FileSystem Directory CDATA #REQUIRED
   Suffix CDATA #IMPLIED>
<!ELEMENT URL EMPTY>
<!ATTLIST URL Name CDATA #REQUIRED>
<!ELEMENT Database EMPTY>
<!ATTLIST Database Location CDATA #IMPLIED
   Name CDATA #IMPLIED
   Query CDATA #REQUIRED>
<!ELEMENT Mapping (Selection, Target)>
<!ATTLIST Mapping Weight CDATA #IMPLIED>
<!ELEMENT Selection (Query | Fixed)+>
<!ELEMENT Query (Target)>
<!ATTLIST Query XPath CDATA #IMPLIED
```

```
    Xql CDATA #IMPLIED>
<!ELEMENT Fixed (Target)>
<!ATTLIST Fixed Value CDATA #REQUIRED>
<!ELEMENT Target (JavaRef)>
<!ELEMENT JavaRef EMPTY>
<!ATTLIST JavaRef Codebase CDATA #IMPLIED
    Class CDATA #REQUIRED
    Method CDATA #IMPLIED>
<!ELEMENT Evaluation (Target)>
```

Listing A.5: *XJML*-DTD

A.6 Ein Konfigurationsdokument für die CAN-Roboterzelle

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE DeMML SYSTEM
  'http://www-sr.informatik.uni-tuebingen.de/Insight/DeMML.dtd' [
  <!ENTITY suvretta
    "Codebase='http://suvretta.informatik.uni-tuebingen.de/classes'">
  <!ENTITY JCanPort
    "<PortClass␣InitArgs='500000,2,0'><JavaRef
      Codebase='http://robo16.fh-reutlingen.de/classes'
      Class='de.wsi.devin.JCanPort'/>
      <PortMethod>disconnectRemoteNode</PortMethod>
      <PortMethod>startRemoteNode</PortMethod>
      <PortMethod>stopRemoteNode</PortMethod>
      <PortMethod>enterPreOperationalState</PortMethod>
      <PortMethod>resetNode</PortMethod>
      <PortMethod>resetCommunication</PortMethod></PortClass>">
  <!ENTITY MoogPort
    "<PortClass␣InitArgs='500000,2,1,0'><JavaRef
      Codebase='http://robo16.fh-reutlingen.de/classes'
      Class='de.wsi.devin.MoogPort'/></PortClass>">
  <!ENTITY CANopenSDOParameter
    "<ParameterClass><JavaRef
      Codebase='http://robo16.fh-reutlingen.de/classes'
      Class='de.wsi.devin.CANopenSDOParameter'/></ParameterClass>">
  <!ENTITY HttpImagePort
    "<PortClass
      InitArgs='http://robo16.fh-reutlingen.de:8888/video/push'><JavaRef
      Codebase='http://robo16.fh-reutlingen.de/classes'
      Class='de.wsi.devin.HttpImagePort'/></PortClass>">
  <!ENTITY SonyVISCAPort
    "<PortClass␣InitArgs='COM2,synchronous,1'><JavaRef
      Class='de.wsi.devin.SonyVISCAPort'
      Codebase='http://robo16.fh-reutlingen.de/classes'/>
      <PortMethod>initCam</PortMethod>
      <PortMethod>getStatistics</PortMethod>
      <PortMethod>clearInput</PortMethod></PortClass>">
  <!ENTITY CANopenUnsigned8PDOParameter
    "<ParameterClass␣InitArgs='false'><JavaRef
      Codebase='http://robo16.fh-reutlingen.de/classes'
      Class='de.wsi.devin.CANopenUnsigned8PDOParameter'/></ParameterClass>">
```

```
  <!ENTITY DS401IOModuleParameter
    "<ParameterClass><JavaRef
      Codebase='http://robo16.fh-reutlingen.de/classes'
      Class='de.wsi.devin.DS401IOModuleParameter'/></ParameterClass>">
  <!ENTITY HttpImageParameter
    "<ParameterClass><JavaRef
      Class='de.wsi.devin.HttpImageParameter'
      Codebase='http://robo16.fh-reutlingen.de/classes'/></ParameterClass>">
  <!ENTITY SonyVISCAParameter
    "<ParameterClass><JavaRef
      Codebase='http://robo16.fh-reutlingen.de/classes'
      Class='de.wsi.devin.SonyVISCAParameter'/></ParameterClass>">
  <!ENTITY MoogParameter
    "<ParameterClass><JavaRef
      Codebase='http://robo16.fh-reutlingen.de/classes'
      Class='de.wsi.devin.MoogParameter'/></ParameterClass>">
  <!ENTITY CANMsgParameter
    "<ParameterClass><JavaRef
      Codebase='http://robo16.fh-reutlingen.de/classes'
      Class='de.wsi.devin.CANMsgParameter'/></ParameterClass>">
]>
<DeMML>
  <SystemConfiguration>
    <FileInfo CreationTime="2001-03-19␣03:52:38.253"/>
    <StateInfo>
      <Description>Cell running by JFCF (Version 0.1c)</Description>
    </StateInfo>
    <SystemInfo>
      <Description>Configuration file for the CANopen robot cell
        at the University of Applied Sciences of Reutlingen</Description>
      <Location Id="13" Country="Germany" Address="Institut␣fuer␣Angewandte
        Forschung in der Automatisierung, FH Reutlingen, Alteburgstr. 150,
        72762 Reutlingen"␣ServerIP="134.103.36.166"
        ServerName="cellserv.fh-reutlingen.de"/>
      <Contact Id="0001" FirstName="Dieter" Surname="Buehler"
        Email="buehler@informatik.uni-tuebingen.de"
        Phone="0049␣(0)7071␣29␣77365" Fax="0049␣(0)7071␣295060"
        Homepage="http://www-sr.informatik.uni-tuebingen.de/~buehler"/>
    </SystemInfo>

    <!-- CANopenDevice element pruduced by EDS2XML from file DIOC711.eds -->
    <CANopenDevice>
      <Device DeviceId="0F">
        <Description>Lift IO (sensors + flaps)</Description>
        <FileInfo CreatedBy="Juergen␣Klueser,␣Vector␣Informatik␣GmbH"
          CreationDate="05-21-96" CreationTime="07:00AM"
          Description="EDS␣for␣Selectron␣DIOC␣711" FileRevision="4"
          FileVersion="2" FileName="R:\PCO\EDS\dioc711.eds">
          <FileInfoExt ModifiedBy="Juergen␣Klueser,␣Vector␣Informatik␣GmbH"
            ModificationDate="04-11-97" ModificationTime="08:24AM"/>
        </FileInfo>
        <DeviceInfo>
          <Vendor Name="Selectron" VendorId="0"/>
          <Product Name="DIOC711" Revision="1" ProductId="0" Version="1"/>
          <DeviceInfoExt Granularity="8" DynamicChannelsSupported="0x0"
            LSS_Supported="0" GroupMessaging="0"/>
        </DeviceInfo>
        <Parameters>
          <SupportedDataTypes>
            <SupportedDataType DataTypeId="0x0004"/>
```

```xml
          <SupportedDataType DataTypeId="0x0005"/>
          <SupportedDataType DataTypeId="0x0006"/>
          <SupportedDataType DataTypeId="0x0008"/>
        </SupportedDataTypes>
        <ParameterCategory Name="MandatoryObjects">
        <Parameter Name="Device␣Type" DataType="7" AccessType="ro"
          ParameterId="1000,0" Monitoring="1" Mute="0">
          <DefaultValue>0x30191</DefaultValue>
          <ParameterExt Index="1000" Subindex="0" PDOMapping="0"
            ObjectType="7"/>
        </Parameter>
        <Parameter Name="Error␣Register" DataType="5" AccessType="ro"
          ParameterId="1001,0" Monitoring="1" Mute="0">
          <DefaultValue>0</DefaultValue>
          <ParameterExt Index="1001" Subindex="0" PDOMapping="1"
            ObjectType="7"/>
        </Parameter>
        </ParameterCategory>
        <ParameterCategory Name="OptionalObjects">
          <Parameter Name="Manufacturer␣Status␣Register" DataType="7"
            AccessType="ro" ParameterId="1002,0" Monitoring="1" Mute="0">
            <ParameterExt Index="1002" Subindex="0" PDOMapping="1"
              ObjectType="7"/>
          </Parameter>
          <ParameterGroup Name="Predefined␣Error␣Field" DataType="7"
            AccessType="ro">
            <Parameter Name="Nr␣of␣Errors" DataType="7" AccessType="ro"
              ParameterId="1003,0" Monitoring="1" Mute="0">
              <DefaultValue>1</DefaultValue>
              <ParameterExt Index="1003" PDOMapping="0" ObjectType="7"
                Subindex="0"/>
            </Parameter>
            <Parameter Name="Standard␣Error␣Field" DataType="7"
              AccessType="ro" ParameterId="1003,1" Monitoring="1"
              Mute="0">
              <ParameterExt Index="1003" PDOMapping="0" ObjectType="7"
                Subindex="1"/>
            </Parameter>
          </ParameterGroup>
          <ParameterGroup Name="ReadInputByte_8InputLines" DataType="5"
            AccessType="ro">
            <Parameter Name="Value␣1.InputModule" DataType="5"
              AccessType="ro" ParameterId="6000,1" Monitoring="1"
              Mute="0">
              <ParameterExt Index="6000" PDOMapping="1" ObjectType="7"
                Subindex="1"/>
            </Parameter>
            <Parameter Name="Value␣2.InputModule" DataType="5"
              AccessType="ro" ParameterId="6000,2" Monitoring="1"
              Mute="0">
              <ParameterExt Index="6000" PDOMapping="1" ObjectType="7"
                Subindex="2"/>
            </Parameter>
          </ParameterGroup>
          <ParameterGroup Name="WriteOutputByte" DataType="5"
            AccessType="rw">
            <Parameter Name="NrOutputModules" DataType="5"
              AccessType="ro"
              ParameterId="6200,0" Monitoring="1" Mute="0">
              <DefaultValue>2</DefaultValue>
```

```
            <ParameterExt Index="6200" PDOMapping="0" ObjectType="7"
              Subindex="0"/>
          </Parameter>
          <Parameter Name="Output1" DataType="5" AccessType="rw"
            ParameterId="6200,1" Monitoring="1" Mute="0">
            <DefaultValue>0</DefaultValue>
            <ParameterExt Index="6200" PDOMapping="1" ObjectType="7"
              Subindex="1"/>
          </Parameter>
          <Parameter Name="Output2" DataType="5" AccessType="rw"
            ParameterId="6200,2" Monitoring="1" Mute="0">
            <DefaultValue>0</DefaultValue>
            <ParameterExt Index="6200" PDOMapping="1" ObjectType="7"
              Subindex="2"/>
          </Parameter>
          <Parameter Name="Output3" DataType="5" AccessType="rw"
             ParameterId="6200,3" Monitoring="1" Mute="0">
             <DefaultValue>0</DefaultValue>
             <ParameterExt Index="6200" PDOMapping="1" ObjectType="7"
               Subindex="3"/>
          </Parameter>
        </ParameterGroup>
      </ParameterCategory>
    </Parameters>
    <DataAccess>
      &JCanPort;
      &CANopenSDOParameter;
    </DataAccess>
  </Device>
  <CANopenDeviceExt/>
</CANopenDevice>

<!-- CANopenDevice element produced by EDS2XML from file DIOC711.eds -->
<CANopenDevice>
  <Device DeviceId="05">
    <Description>Robot IO (comp. air)</Description>
    <FileInfo CreatedBy="Juergen␣Klueser,␣Vector␣Informatik␣GmbH"
      CreationDate="05-21-96"
      CreationTime="07:00AM" Description="EDS␣for␣Selectron␣DIOC␣711"
      FileRevision="4" FileVersion="2" FileName="R:\PCO\EDS\dioc711.eds">
      <FileInfoExt ModifiedBy="Juergen␣Klueser,␣Vector␣Informatik␣GmbH"
        ModificationDate="04-11-97" ModificationTime="08:24AM"/>
    </FileInfo>
    <DeviceInfo>
      <Vendor Name="Selectron" VendorId="0"/>
      <Product Name="DIOC711" Revision="1" ProductId="0" Version="1"/>
      <DeviceInfoExt Granularity="8" DynamicChannelsSupported="0x0"
        LSS_Supported="0"
        GroupMessaging="0"/>
    </DeviceInfo>
    <Parameters>
      <SupportedDataTypes>
        <SupportedDataType DataTypeId="0x0004"/>
        <SupportedDataType DataTypeId="0x0005"/>
        <SupportedDataType DataTypeId="0x0006"/>
        <SupportedDataType DataTypeId="0x0008"/>
      </SupportedDataTypes>
      <ParameterCategory Name="MandatoryObjects">
        <Parameter Name="Device␣Type" DataType="7" AccessType="ro"
          ParameterId="1000,0" Monitoring="1" Mute="0">
```

```
          <DefaultValue>0x30191</DefaultValue>
          <ParameterExt Index="1000" Subindex="0" PDOMapping="0"
            ObjectType="7"/>
        </Parameter>
        <Parameter Name="Error␣Register" DataType="5" AccessType="ro"
          ParameterId="1001,0" Monitoring="1" Mute="0">
          <DefaultValue>0</DefaultValue>
          <ParameterExt Index="1001" Subindex="0" PDOMapping="1"
            ObjectType="7"/>
        </Parameter>
      </ParameterCategory>
      <ParameterCategory Name="OptionalObjects">
        <Parameter Name="Manufacturer␣Status␣Register" DataType="7"
          AccessType="ro" ParameterId="1002,0" Monitoring="1" Mute="0">
          <ParameterExt Index="1002" Subindex="0" PDOMapping="1"
            ObjectType="7"/>
        </Parameter>
        <ParameterGroup Name="Predefined␣Error␣Field" DataType="7"
          AccessType="ro">
          <Parameter Name="Nr␣of␣Errors" DataType="7" AccessType="ro"
            ParameterId="1003,0" Monitoring="1" Mute="0">
            <DefaultValue>1</DefaultValue>
            <ParameterExt Index="1003" PDOMapping="0" ObjectType="7"
              Subindex="0"/>
          </Parameter>
          <Parameter Name="Standard␣Error␣Field" DataType="7"
            AccessType="ro"
            ParameterId="1003,1" Monitoring="1" Mute="0">
            <ParameterExt Index="1003" PDOMapping="0" ObjectType="7"
              Subindex="1"/>
          </Parameter>
        </ParameterGroup>
        <ParameterGroup Name="WriteOutputByte" DataType="5"
          AccessType="rw">
          <Parameter Name="Output1" DataType="5" AccessType="rw"
            ParameterId="6200,1" Monitoring="1" Mute="0">
            <DefaultValue>0</DefaultValue>
            <ParameterExt Index="6200" PDOMapping="1" ObjectType="7"
              Subindex="1"/>
          </Parameter>
        </ParameterGroup>
      </ParameterCategory>
    </Parameters>
    <DataAccess>
      &JCanPort;
      &CANopenSDOParameter;
    </DataAccess>
  </Device>
  <CANopenDeviceExt/>
</CANopenDevice>

<Device DeviceId="12">
  <Description>Separation IO (sensors + flaps)</Description>
  <DeviceInfo>
    <Vendor Name="Selectron" VendorId="0"/>
    <Product Name="DIOC"/>
    <DeviceInfoExt Granularity="8" DynamicChannelsSupported="0x0"
      LSS_Supported="0" GroupMessaging="0"/>
  </DeviceInfo>
  <Parameters>
```

```
      <SupportedDataTypes>
        <SupportedDataType DataTypeId="0x0004"/>
        <SupportedDataType DataTypeId="0x0005"/>
        <SupportedDataType DataTypeId="0x0006"/>
        <SupportedDataType DataTypeId="0x0008"/>
      </SupportedDataTypes>
      <ParameterCategory Name="Optional␣Parameters">
        <Parameter Name="PDO␣Status␣Input␣Lines" DataType="5"
          AccessType="ro" ParameterId="RPDO1" Monitoring="1" Mute="0"/>
      </ParameterCategory>
    </Parameters>
    <DataAccess>
      &JCanPort;
      &CANopenUnsigned8PDOParameter;
    </DataAccess>
  </Device>

  <Device DeviceId="09">
    <Description>Transfer IO (sensors + raiser)</Description>
    <DeviceInfo>
      <Vendor Name="Leukardt" VendorId="0"/>
      <Product Name="DEASY"/>
    </DeviceInfo>
    <Parameters>
      <ParameterCategory Name="Optional␣Parameters">
        <Parameter Name="Drive␣Status" DataType="5" AccessType="ro"
          ParameterId="208" Monitoring="1" Mute="0"/>
      </ParameterCategory>
    </Parameters>
    <DataAccess>
      &JCanPort;
      &CANMsgParameter;
    </DataAccess>
  </Device>

  <Device DeviceId="08">
    <Description>Transfer Drive (left, right, stop)</Description>
    <DeviceInfo>
      <Vendor Name="Bisch" VendorId="0"/>
      <Product Name="NN"/>
    </DeviceInfo>
    <Parameters>
      <ParameterCategory Name="Optional␣Parameters">
        <Parameter Name="PDO␣Status␣Input␣Lines" DataType="5"
          AccessType="ro" ParameterId="RPDO1" Monitoring="1" Mute="0"/>
      </ParameterCategory>
    </Parameters>
    <DataAccess>
      &JCanPort;
      &CANopenUnsigned8PDOParameter;
    </DataAccess>
  </Device>

  <!--- the scara robot: ----------------->
  <Device DeviceId="1">
    <Description>SCARA robot drive 1</Description>
    <DeviceInfo>
      <Vendor Name="Moog" VendorId="0"/>
      <Product Name="NN"/>
    </DeviceInfo>
```

```
    <Parameters>
      <ParameterCategory Name="Drive␣Parameters">
        <Parameter Name="Position" DataType="5" AccessType="ro"
          ParameterId="409" Monitoring="1" Mute="0"/>
      </ParameterCategory>
    </Parameters>
    <DataAccess>
      &MoogPort;
      &MoogParameter;
    </DataAccess>
  </Device>

  <Device DeviceId="2">
    <Description>SCARA robot drive 2</Description>
    <DeviceInfo>
      <Vendor Name="Moog" VendorId="0"/>
      <Product Name="NN"/>
    </DeviceInfo>
    <Parameters>
      <ParameterCategory Name="Drive␣Parameters">
       <Parameter Name="Position" DataType="5" AccessType="ro"
         ParameterId="411" Monitoring="1" Mute="0"/>
     </ParameterCategory>
    </Parameters>
    <DataAccess>
      &MoogPort;
      &MoogParameter;
    </DataAccess>
  </Device>

  <Device DeviceId="3">
    <Description>SCARA robot drive 3</Description>
    <DeviceInfo>
      <Vendor Name="Moog" VendorId="0"/>
      <Product Name="NN"/>
    </DeviceInfo>
    <Parameters>
      <ParameterCategory Name="Drive␣Parameters">
        <Parameter Name="Position" DataType="5" AccessType="ro"
          ParameterId="419" Monitoring="1" Mute="0"/>
      </ParameterCategory>
    </Parameters>
    <DataAccess>
      &MoogPort;
      &MoogParameter;
    </DataAccess>
  </Device>

  <Device DeviceId="4">
    <Description>SCARA robot drive 4</Description>
    <DeviceInfo>
      <Vendor Name="Moog" VendorId="0"/>
      <Product Name="NN"/>
    </DeviceInfo>
    <Parameters>
      <ParameterCategory Name="Drive␣Parameters">
        <Parameter Name="Position" DataType="5" AccessType="ro"
          ParameterId="421" Monitoring="1" Mute="1"/>
      </ParameterCategory>
    </Parameters>
```

```
      <DataAccess>
        &MoogPort;
        &MoogParameter;
      </DataAccess>
    </Device>

    <!--- the interactive web cam: ---------------->
    <Device DeviceId="Cam1">
      <DeviceInfo>
        <Vendor Name="Arlt"/>
        <Product Name="Sony␣EVI␣D31"/>
      </DeviceInfo>
      <Parameters>
        <ParameterCategory Name="ImageParameters">
          <Parameter ParameterId="Image1" DataType="image/jpeg"
            Name="Overview" AccessType="rw" Monitoring="1" Mute="0">
          <DataAccess>
            &HttpImagePort;
            &HttpImageParameter;
          </DataAccess>
        </Parameter>
        </ParameterCategory>
        <ParameterCategory Name="CamControlParameters">
          <Parameter ParameterId="Pan" DataType="Integer"
            Name="Pan␣Position" AccessType="ro" Monitoring="1" Mute="0"/>
          <Parameter ParameterId="Tilt" DataType="Integer"
            Name="Tilt␣Position" AccessType="ro" Monitoring="1" Mute="0"/>
          <Parameter ParameterId="Zoom" DataType="Integer"
            Name="Zoom" AccessType="ro" Monitoring="1" Mute="0"/>
        </ParameterCategory>
      </Parameters>
      <DataAccess>
        &SonyVISCAPort;
        &SonyVISCAParameter;
      </DataAccess>
    </Device>
  </SystemConfiguration>
</DeMML>
```

Listing A.6: Ein Konfigurationsdokument für die CAN-Roboterzelle

A.7 Ein *Snapshot*-Dokument für die CAN-Roboterzelle

```
<?xml version="1.0" encoding="UTF-8"?>

<DeMML>
  <Snapshot StartOfQuery="2001-03-19␣11:07:35.119"
    EndOfQuery="2001-03-19␣11:07:36.511" LocationId="13"
    SystemConfigurationCreationTime="2001-03-19␣10:47:38.253">
    <Description>Error State: transfer system left sensor offline
    </Description>
    <ScannedDevice DeviceId="0F">
      <LogEntry ParameterId="1000,0" Valid="1" Value="30191" />
```

```
      <LogEntry ParameterId="1001,0" Valid="1" Value="0" />
      <LogEntry ParameterId="1002,0" Valid="1" Value="50000" />
      <LogEntry ParameterId="1003,0" Valid="1" Value="1" />
      <LogEntry ParameterId="1003,1" Valid="1" Value="0" />
      <LogEntry ParameterId="6000,1" Valid="1" Value="0" />
      <LogEntry ParameterId="6000,2" Valid="1" Value="25" />
      <LogEntry ParameterId="6200,0" Valid="1" Value="3" />
      <LogEntry ParameterId="6200,1" Valid="1" Value="0" />
      <LogEntry ParameterId="6200,2" Valid="1" Value="9a" />
      <LogEntry ParameterId="6200,3" Valid="1" Value="94" />
    </ScannedDevice>
    <ScannedDevice DeviceId="05">
      <LogEntry ParameterId="1000,0" Valid="1" Value="30191" />
      <LogEntry ParameterId="1001,0" Valid="1" Value="0" />
      <LogEntry ParameterId="1002,0" Valid="1" Value="50000" />
      <LogEntry ParameterId="1003,0" Valid="1" Value="1" />
      <LogEntry ParameterId="1003,1" Valid="1" Value="0" />
      <LogEntry ParameterId="6200,1" Valid="1" Value="1" />
    </ScannedDevice>
    <ScannedDevice DeviceId="12">
      <LogEntry ParameterId="RPDO1" Valid="1" Value="50" />
    </ScannedDevice>
    <ScannedDevice DeviceId="09">
      <LogEntry ParameterId="208" Valid="1" Value="6␣9␣40␣0␣0␣0␣0␣0␣" />
    </ScannedDevice>
    <ScannedDevice DeviceId="08">
      <LogEntry ParameterId="RPDO1" Valid="1" Value="18" />
    </ScannedDevice><ScannedDevice DeviceId="1">
      <LogEntry ParameterId="409" Valid="0" Value="170637" />
    </ScannedDevice><ScannedDevice DeviceId="2">
      <LogEntry ParameterId="411" Valid="0" Value="-59479" />
    </ScannedDevice><ScannedDevice DeviceId="3">
      <LogEntry ParameterId="419" Valid="0" Value="0" />
    </ScannedDevice>
    <ScannedDevice DeviceId="Cam1">
      <LogEntry ParameterId="Image1" Valid="1"
        Value="http://suvretta.informatik.uni-tuebingen.de/tamino/DeMML/\
binary/Pics/13-Image1-2001-03-19-11-07-35.119"␣/>
      <LogEntry ParameterId="Pan" Valid="1" Value="861" />
      <LogEntry ParameterId="Tilt" Valid="1" Value="65340" />
      <LogEntry ParameterId="Zoom" Valid="1" Value="175" />
    </ScannedDevice>
  </Snapshot>
</DeMML>
```

Listing A.7: Ein *Snapshot*-Dokument für die CAN-Roboterzelle

A.8 Ein *XJML*-Dokument für die CAN-Roboterzelle

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE XJML SYSTEM
  'http://www-sr.informatik.uni-tuebingen.de/Insight/XJML.dtd' [
  <!ENTITY suvretta "http://suvretta.informatik.uni-tuebingen.de/classes">
  <!ENTITY base "de.wsi.xjml">
  <!ENTITY fixed8 "<Fixed␣Value='8'>
```

```
    <Target><JavaRef Class='java.lang.Integer'/></Target></Fixed>">
  <!ENTITY fixed255 "<Fixed␣Value='255'>
    <Target><JavaRef Class='java.lang.Integer'/></Target></Fixed>">
  <!ENTITY fixed1000000 "<Fixed␣Value='1000000'>
    <Target><JavaRef Class='java.lang.Long'/></Target></Fixed>">
  <!ENTITY fixed70 "<Fixed␣Value='70'>
    <Target><JavaRef Class='java.lang.Integer'/></Target></Fixed>">
  <!ENTITY fixed100 "<Fixed␣Value='100'>
    <Target><JavaRef Class='java.lang.Integer'/></Target></Fixed>">
]>
<XJML>
  <Statement>
    <XMLDataRepository RootElement="Snapshot">
      <FileSystem Directory="cell_log" Suffix="xml"/>
    </XMLDataRepository>
    <!-- Device 0F -->
    <Mapping>
      <!-- standard error reg. -->
      <Selection>
        <Query XPath="ScannedDevice[@DeviceId='0F']/LogEntry[@ParameterId=\
'1001,0']/@Value">
          <Target>
            <JavaRef Class="java.lang.Integer"/>
          </Target>
        </Query>
      </Selection>
      <Target>
        <JavaRef Codebase="&suvretta;" Class="&base;.eval.Methods"
          Method="integerEqual"/>
      </Target>
    </Mapping>
    <Mapping>
      <!-- node status -->
      <Selection>
        <Query XPath="ScannedDevice[@DeviceId='0F']/LogEntry[@ParameterId=\
'1002,0']/@Value">
          <Target>
            <JavaRef Codebase="&suvretta;"
              Class="de.uni_tuebingen.can.CanProtocol.CANopenNodeState"/>
          </Target>
        </Query>
      </Selection>
      <Target>
        <JavaRef Codebase="&suvretta;" Class="&base;.eval.Methods"
          Method="CANopenNodeStateDistance"/>
      </Target>
    </Mapping>
    <Mapping>
      <!-- no. of errors -->
      <Selection>
        <Query XPath="ScannedDevice[@DeviceId='0F']/LogEntry[@ParameterId=\
'1003,0']/@Value">
          <Target>
            <JavaRef Class="java.lang.Integer"/>
          </Target>
        </Query>
        &fixed255;
      </Selection>
      <Target>
        <JavaRef Codebase="&suvretta;" Class="&base;.eval.Methods"
```

```
        Method="integerDistance"/>
      </Target>
    </Mapping>
    <Mapping>
      <!-- first input  byte  ( channels 0-7) -->
      <Selection>
        <Query XPath="ScannedDevice[@DeviceId='0F']/LogEntry[@ParameterId=\
'6000,1']/@Value">
          <Target>
            <JavaRef Codebase="&suvretta;" Class="&base;.IntegerHex"/>
          </Target>
        </Query>
        &fixed8;
      </Selection>
      <Target>
        <JavaRef Codebase="&suvretta;" Class="&base;.eval.Methods"
          Method="hammingDistance"/>
      </Target>
    </Mapping>
    <Mapping>
      <!-- second input byte (channels 8-15) -->
      <Selection>
        <Query XPath="ScannedDevice[@DeviceId='0F']/LogEntry[@ParameterId=\
'6000,2']/@Value">
          <Target>
            <JavaRef Codebase="&suvretta;" Class="&base;.IntegerHex"/>
          </Target>
        </Query>
        &fixed8;
      </Selection>
      <Target>
        <JavaRef Codebase="&suvretta;" Class="&base;.eval.Methods"
          Method="hammingDistance"/>
      </Target>
    </Mapping>
    <Mapping>
      <!-- first output  byte  ( channels 0-7) -->
      <Selection>
        <Query XPath="ScannedDevice[@DeviceId='0F']/LogEntry[@ParameterId=\
'6200,1']/@Value">
          <Target>
            <JavaRef Codebase="&suvretta;" Class="&base;.IntegerHex"/>
          </Target>
        </Query>
        &fixed8;
      </Selection>
      <Target>
        <JavaRef Codebase="&suvretta;" Class="&base;.eval.Methods"
          Method="hammingDistance"/>
      </Target>
    </Mapping>
    <Mapping>
      <!-- second output byte (channels 8-15) -->
      <Selection>
        <Query XPath="ScannedDevice[@DeviceId='0F']/LogEntry[@ParameterId=\
'6200,2']/@Value">
          <Target>
            <JavaRef Codebase="&suvretta;" Class="&base;.IntegerHex"/>
          </Target>
        </Query>
```

```
        &fixed8;
      </Selection>
      <Target>
        <JavaRef Codebase="&suvretta;" Class="&base;.eval.Methods"
          Method="hammingDistance"/>
      </Target>
    </Mapping>
    <Mapping>
      <Selection>
      <!-- third output byte (channels 16-23) -->
        <Query XPath="ScannedDevice[@DeviceId='0F']/LogEntry[@ParameterId=\
'6200,3']/@Value">
          <Target>
            <JavaRef Codebase="&suvretta;" Class="&base;.IntegerHex"/>
          </Target>
        </Query>
        &fixed8;
      </Selection>
      <Target>
        <JavaRef Codebase="&suvretta;" Class="&base;.eval.Methods"
          Method="hammingDistance"/>
      </Target>
    </Mapping>
    <!-- Device 05 -->
    <Mapping>
      <!-- standard error reg. -->
      <Selection>
        <Query XPath="ScannedDevice[@DeviceId='05']/LogEntry[@ParameterId=\
'1001,0']/@Value">
          <Target>
            <JavaRef Class="java.lang.Integer"/>
          </Target>
        </Query>
      </Selection>
      <Target>
        <JavaRef Codebase="&suvretta;" Class="&base;.eval.Methods"
          Method="integerEqual"/>
      </Target>
    </Mapping>
    <Mapping>
      <!-- node status -->
      <Selection>
        <Query XPath="ScannedDevice[@DeviceId='05']/LogEntry[@ParameterId=\
'1002,0']/@Value">
          <Target>
            <JavaRef Codebase="&suvretta;"
              Class="de.uni_tuebingen.can.CanProtocol.CANopenNodeState"/>
          </Target>
        </Query>
      </Selection>
      <Target>
        <JavaRef Codebase="&suvretta;" Class="&base;.eval.Methods"
          Method="CANopenNodeStateDistance"/>
      </Target>
    </Mapping>
    <Mapping>
      <!-- no. of errors -->
      <Selection>
        <Query XPath="ScannedDevice[@DeviceId='05']/LogEntry[@ParameterId=\
'1003,0']/@Value">
```

```
     <Target>
      <JavaRef Class="java.lang.Integer"/>
     </Target>
    </Query>
    &fixed255;
   </Selection>
   <Target>
    <JavaRef Codebase="&suvretta;" Class="&base;.eval.Methods"
     Method="integerDistance"/>
   </Target>
  </Mapping>
  <Mapping>
   <!-- first output byte (channels 0-7) -->
   <Selection>
    <Query XPath="ScannedDevice[@DeviceId='05']/LogEntry[@ParameterId=\
'6200,1']/@Value">
     <Target>
      <JavaRef Codebase="&suvretta;" Class="&base;.IntegerHex"/>
     </Target>
    </Query>
    &fixed8;
   </Selection>
   <Target>
    <JavaRef Codebase="&suvretta;" Class="&base;.eval.Methods"
     Method="hammingDistance"/>
   </Target>
  </Mapping>
  <!-- Device 12 -->
  <Mapping>
   <!-- first input byte (channels 0-7) -->
   <Selection>
    <Query XPath="ScannedDevice[@DeviceId='12']/LogEntry[@ParameterId=\
'RPDO1']/@Value">
     <Target>
      <JavaRef Codebase="&suvretta;" Class="&base;.IntegerHex"/>
     </Target>
    </Query>
    &fixed8;
   </Selection>
   <Target>
    <JavaRef Codebase="&suvretta;" Class="&base;.eval.Methods"
     Method="hammingDistance"/>
   </Target>
  </Mapping>
  <!-- Device 09 -->
  <Mapping>
   <!-- transfer system raiser/sensor status -->
   <Selection>
    <Query XPath="ScannedDevice[@DeviceId='09']/LogEntry[@ParameterId=\
'208']/@Value">
     <Target>
      <JavaRef Codebase="&suvretta;" Class="&base;.eval.CanMessage"/>
     </Target>
    </Query>
   </Selection>
   <Target>
    <JavaRef Codebase="&suvretta;" Class="&base;.eval.Methods"
     Method="CANMessageDistance"/>
   </Target>
  </Mapping>
```

```
    <!-- Device 08 -->
    <Mapping>
      <Selection>
        <!-- transfer system drive status -->
        <Query XPath="ScannedDevice[@DeviceId='08']/LogEntry[@ParameterId=\
'RPDO1']/@Value">
          <Target>
            <JavaRef Codebase="&suvretta;" Class="&base;.IntegerHex"/>
          </Target>
        </Query>
        &fixed8;
      </Selection>
      <Target>
        <JavaRef Codebase="&suvretta;" Class="&base;.eval.Methods"
          Method="hammingDistance"/>
      </Target>
    </Mapping>

    <!-- robot position -->
    <Mapping>
      <!-- axis 1-4 -->
      <Selection>
        <Query XPath="ScannedDevice[@DeviceId='1']/LogEntry[@ParameterId='\
409']/@Value">
          <Target>
            <JavaRef Class="java.lang.Long"/>
          </Target>
        </Query>
        &fixed1000000;
        <Query XPath="ScannedDevice[@DeviceId='2']/LogEntry[@ParameterId='\
411']/@Value">
          <Target>
            <JavaRef Class="java.lang.Long"/>
          </Target>
        </Query>
        &fixed1000000;
        <Query XPath="ScannedDevice[@DeviceId='3']/LogEntry[@ParameterId='\
419']/@Value">
          <Target>
            <JavaRef Class="java.lang.Long"/>
          </Target>
        </Query>
        &fixed1000000;
        <Fixed Value="320,300,200,140,150,200,">
          <Target>
            <JavaRef Codebase="&suvretta;"
              Class="&base;.SCARAGeometry"></JavaRef>
          </Target>
        </Fixed>
      </Selection>
      <Target>
        <JavaRef Codebase="&suvretta;" Class="&base;.eval.Methods"
          Method="SCARASpatialDistance"/>
      </Target>
    </Mapping>
    <Evaluation>
      <Target>
        <JavaRef Codebase="&suvretta;" Class="&base;.eval.Methods"
          Method="doubleAverage"/>
      </Target>
```

```
    </Evaluation>
  </Statement>
</XJML>
```

Listing A.8: Ein *XJML*-Dokument für die CAN-Roboterzelle

A.9 Ausschnitt aus einer *XSL*-Sicht für *DeMML*-Dokumente

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
  version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="SystemConfiguration">
    <html><head><title>Parameter</title>
      <link rel="stylesheet" type="text/css" href="style.css"/></head>
      <body>
      <h1>Parameter</h1>
      <table border="0">
        <tr>
          <th class="blind" align="left"> StartOfQuery: </th>
          <td class="blind">
            <xsl:value-of select="StateInfo/@StartOfQuery"/></td>
        </tr><tr>
          <th class="blind" align="left"> EndOfQuery: </th>
          <td class="blind">
            <xsl:value-of select="StateInfo/@EndOfQuery"/></td>
        </tr>
      </table><br/>
      <b>Devices:</b>
      <xsl:for-each select=".//Device">
        <xsl:variable name="DeviceId" select="@DeviceId"/>
        <a href='#{$DeviceId}'><xsl:value-of select="$DeviceId␣"/></a>
        <xsl:text> </xsl:text>
      </xsl:for-each>
      <br/>
      <table border="0">
        <xsl:for-each select=".//Device">
          <xsl:variable name="DeviceId" select="@DeviceId"/>
          <tr>
          <th colspan="7" class="blind">
            <a name= '{$DeviceId}'>DeviceID :
              <xsl:value-of select="$DeviceId"/></a>
          </th>
          </tr>
          <!-- ... -->
        </xsl:for-each>
       </table>
       <p><i> -- This page was created dynamically
         by the Insight system -- </i></p>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

Listing A.9: Ausschnitt aus einer *XSL*-Sicht für *DeMML*-Dokumente

Literaturverzeichnis

[ABCS96] AKTAN, B., C. A. BOHUS, L. A. CROWL und M. H. SHOR: *Distance Learning Applied to Control Engineering Laboratories*. IEEE Transactions on Education, 39, August 1996.

[ABK95] AVITZUR, R., O. BACHMANN und N. KAJLER: *From Honest to Intelligent Plotting*. In: LEVELT, A.H.M. (Herausgeber): *Proc. of the International Symposium on Symbolic and Algebraic Computation (ISSAC '95)*, Montreal, Kanada, 1995. ACM Press.

[ACHK93] ARENS, YIGAL, CHIN Y. CHEE, CHUN-NAN HSU und CRAIG A. KNOBLOCK: *Retrieving and Integrating Data from Multiple Information Sources*. International Journal of Intelligent & Cooperative Information Systems, 2(2):127–158, 1993.

[Apaa] THE APACHE SOFTWARE FOUNDATION, http://xml.apache.org: *The Apache XML Project*.

[Apab] THE APACHE XML PROJECT, http://xml.apache.org/cocoon/: *Cocoon*.

[Apac] THE APACHE XML PROJECT, http://xml.apache.org/crimson: *Crimson 1.1*.

[Apad] THE APACHE XML PROJECT, http://xml.apache.org/xalan-j/: *Xalan-Java Version 2.2.D6*.

[APK96] ABOWD, G., J. PITKOW und R. KAZMAN: *Analyzing Differences between Internet Information System Software Architectures*. In: *Proc. of the IEEE ICC/SUPERCOMM '96 - International Conference on Communications*, New York, NY, USA, 1996. IEEE.

[AQM+97] ABITEBOUL, SERGE, DALLAN QUASS, JASON MCHUGH, JENNIFER WIDOM und JANET L. WIENER: *The Lorel Query Language for Semistructured Data*. International Journal on Digital Libraries, 1(1):68–88, 1997.

[ARS92] ARS INNOVANDI, Providencia 2184, Piso 3, Santiago, Chile: *Search City 1.1 Text Retrieval for Windows Power Users*, Mai 1992.

[BBB+97] BERCHTOLD, STEFAN, CHRISTIAN BÖHM, BERNHARD BRAUNMÜLLER, DANIEL A. KEIM und HANS-PETER KRIEGEL: *Fast Parallel Similarity Search in Multimedia Databases*. In: *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Seiten 1 – 12, Tucson, AZ, USA, Mai 1997.

[BC00] BONIFATI, ANGELA und STEFANO CERI: *Comparative Analysis of Five XML Query Languages*. SIGMOD Record, 29(1):68–79, 2000.

[BCK00] BÜHLER, DIETER, CORINNE CHAUVIN und WOLFGANG KÜCHLIN: *Plotting Functions and Singularities with Maple and Java on a Component-based Web Architecture*. In: GANZHA, V. G., E. W. MAYR und E.V. VOROZHTSOV (Herausgeber): *Computer Algebra in Scientific Computing – CASC 2000*, Samarkand, Usbekistan, Oktober 2000. Springer-Verlag.

[Bec96] BECKER, PETER: *Verteiltes Modell-Management und Objektbanken für diskrete Probleme und diskrete Strukturen*. Doktorarbeit, Eberhard-Karls-Universität, Tübingen, Deutschland, 1996.

[Bey01] BEYERLE, MARC: *The Crypto Triangle*. Studienarbeit, Eberhard-Karls-Universität, Tübingen, Deutschland, Januar 2001.

[BG00a] BOLLELLA, GREG und JAMES GOSLING: *The Real-Time Specification for Java*. IEEE Computer, 33(6):47–54, Juni 2000.

[BG00b] BRICKLEY, DAN und R.V. GUHA: *Resource Description Framework (RDF) Schema Specification 1.0*. World Wide Web Consortium (W3C), http://www.w3.org/TR/rdf-schema/, März 2000.

[BG00c] BÜHLER, DIETER und GERHARD GRUHLER: *XML-based Representation and Monitoring of CAN Devices*. In: *Proc. of the 7th International CAN Conference (ICC 2000)*, Amsterdam, The Netherlands, Oktober 2000. CAN in Automation (www.can-cia.com).

[BGNP99] BECKER, L. B., M. GERGELEIT, E. NETT und C. E. PEREIRA: *An Integrated Environment for the Complete Development Cycle of an Object-oriented Distributed Real-Time System*. In: *Proc. of the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 99)*, Saint-Malo, Frankreich, May 1999.

[BGV$^+$00] BÜHLER, DIETER, MICHAEL GROSSMANN, ELIAS VOLANAKIS, WOLFGANG WESTJE und SVEN SCHULZ: *Mathe-Tools - Source Code Documentation*. http://mfb.informatik.uni-tuebingen.de/-JavaDoc/, 2000.

[BHK00] BRICH, PETER, GERHARD HINSKEN und KARL-HEINZ KRAUSE: *Echtzeitprogrammierung in Java*. Siemens, Publics MCD Verlag, Erlangen, Deutschland, 2000.

[BK00] BÜHLER, DIETER und WOLFGANG KÜCHLIN: *Remote Fieldbus System Management with Java and XML*. In: *Proc. of the IEEE International Symposium on Industrial Electronics (ISIE 2000)*, Puebla, Mexiko, Dezember 2000. IEEE.

[BK01] BÜHLER, DIETER und WOLFGANG KÜCHLIN: *Flexible Similarity Assessment for XML Documents Based on XQL and Java Reflection*. In: MONOSTRI, LÁZLÓ, VÁNCZA JÓSEF und ALI MOONIS (Herausgeber): *Engineering of Intelligent Systems – Proc. of the 14th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEA/AIE 2001)*, Seiten 175 – 186, Budapest, Ungarn, Juni 2001. Springer-Verlag (LNCS/LNAI).

[BKGN00] BÜHLER, DIETER, WOLFGANG KÜCHLIN, GERHARD GRUHLER und GERD NUSSER: *The Virtual Automation Lab - Web Based Teaching of Automation Engineering Concepts*. In: *Proc. of the 7th IEEE International Conference on the Engineering of Computer Based Systems (ECBS 2000)*, Edinburgh, Schottland, April 2000. IEEE Computer Society Press.

[BLCGP92] BERNERS-LEE, TIM, R. CAILLIAU, J. F. GROFF und B. POLLERMANN: *The World-Wide Web: The Information Universe*. Electronic Networking: Research, Applications and Policy, 2(1), 1992.

[BLCL$^+$94] Berners-Lee, Tim, Robert Cailliau, Ari Luotonen, Henrik Frystyk Nielsen und Arthur Secret: *The World-Wide Web*. Communications of the ACM, 37(8):76 – 82, 1994.

[BLHL01] Berners-Lee, Tim, James Handler und Ora Lassila: *The Semantic Web*. Scientific American, Mai 2001.

[BLMM94] Berners-Lee, T., L. Masinter und M. McCahill: *Uniform Resource Locators*. RFC1738, http://www.ietf.org/rfc/rfc1738.txt, Dezember 1994.

[BLRIM98] Berners-Lee, T., R.Fielding, U.C. Irvine und L. Masinter: *Uniform Resource Identifiers (URI): Generic Syntax*. RFC2396, http://www.ietf.org/rfc/rfc2396.txt, August 1998.

[BM00] Behme, Henning und Stefan Mintert: *XML in der Praxis*. Addison-Wesley, Reading, Mass, USA, 2000.

[BMA97] Brugali, D., G. Menga und A. Aarsten: *The Framework Life Span*. Communications of the ACM, 40(10):60 – 64, Oktober 1997.

[BN00] Bühler, Dieter und Gerd Nusser: *The Java CAN API – A Java Gateway to Fieldbus Communication*. In: *Proc. of the 2000 IEEE International Workshop on Factory Communication Systems (WFCS 2000)*, Porto, Portugal, September 2000. IEEE.

[BNGK99] Bühler, Dieter, Gerd Nusser, Gerhard Gruhler und Wolfgang Küchlin: *A Java Client/Server System for Accessing Arbitrary CANopen Fieldbus Devices via the Internet*. South African Computer Journal, (24):239–243, November 1999.

[BNKG01] Bühler, Dieter, Gerd Nusser, Wolfgang Küchlin und Gerhard Gruhler: *The Java Fieldbus Control Framework - Object-oriented Control of Fieldbus Devices*. In: *Proc. of the 4th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 01)*, Magdeburg, Deutschland, Mai 2001.

[Bos97] Bosak, John: *XML, Java and the Future of the Web*. http://www.ibiblio.org/pub/sun-info/standards/xml/why/xmlapps.htm, 1997.

[Box98] Box, Don: *Essential COM*. Addison-Wesley, Reading, Mass, USA, 1998.

[Büh98] BÜHLER, DIETER: *Ein flexibles Werkzeug zur Extraktion strukturierter Information aus HTML-Dokumenten und zur Erzeugung von XML-Dokumenten*. Diplomarbeit, Eberhard-Karls-Universität, Tübingen, Deutschland, 1998.

[Büh00] BÜHLER, DIETER: *The CANopen Markup Language – Representing Fieldbus Data with XML*. In: *Proc. of the 26th Annual Conference of the IEEE Industrial Electronics Society (IECON 2000)*, Nagoya, Japan, Oktober 2000. IEEE.

[Büh01] BÜHLER, DIETER: *The INSIGHT System - Source Code Documentation*. http://www-sr.informatik.uni-tuebingen.de/Insight/-JavaDoc/, 2001.

[Can96a] CAN IN AUTOMATION (CIA) E.V., Erlangen, Deutschland: *CAL Based Device Profile for I/O Modules*, 1996. CiA Draft Standard 401.

[Can96b] CAN IN AUTOMATION (CIA) E.V., Erlangen, Deutschland: *CANopen Communication Profile for Industrial Systems, Based on CAL*, 1996. CiA Draft Standard 301.

[Can96c] CAN IN AUTOMATION (CIA) E.V., Erlangen, Deutschland: *CMS Data Types and Encoding Rules*, 1996. CiA Draft Standard 202-3.

[Can96d] CAN IN AUTOMATION (CIA) E.V., Erlangen, Deutschland: *NMT Protocol Specification, Draft Standard 203-1, 203-2*, 1996.

[Can99] CAN IN AUTOMATION (CIA) E.V., Erlangen, Deutschland: *Electronic Data Sheet Specification for CANopen*, September 1999. CiA Work Draft 306, Revision 0.3.

[CCD$^+$99] CERI, S., S. COMAI, E. DAMIANI, P. FRATERNALI, S. PARABOSCHI und L. TANCA: *XML-GL: A Graphical Language for Querying and Restructuring XML documents*. In: *Proc. of the 8th International World Wide Web Conference (WWW8)*, Totonto, Kanada, Mai 1999. Elsevier Science.

[CCH$^+$95] CAO, Y. U., T.-W. CHEN, M. D. HARRIS, A. B. KAHNG, M. A. LEWIS und A.D. STECHERT: *A Remote Robotics Laboratory on the Internet*. In: *Proc. of the Annual Internet Society Conference (INET '95)*, Honolulu, Hawaii, USA, 1995.

[CD99] CLARK, J. und S. DEROSE: *XML Path Language (XPath) Version 1.0*. World Wide Web Consortium (W3C), http://www.w3c.org/-TR/xpath, November 1999.

[CGMH+94] CHAWATHE, SUDARSHAN, HECTOR GARCIA-MOLINA, JOACHIM HAMMER, KELLY IRELAND, YANNIS PAPAKONSTANTINOU, JEFFREY D. ULLMAN und JENNIFER WIDOM: *The TSIMMIS Project: Integration of Heterogeneous Information Sources*. In: *Proc. of the Meeting of the Information Processing Society of Japan*, Seiten 7–18, Tokyo, Japan, 1994.

[Che88] CHEN, PETER PIN-SHAN: *The Entity-Relationship Model - Toward a Unified View of Data*. In: STONEBRAKER, M. (Herausgeber): *Readings in Database Systems*, San Mateo, CA, USA, 1988. Morgan Kaufman.

[Cla97] CLARKSON, K. L.: *Nearest Neighbor Queries in Metric Spaces*. In: *Proc. of the 29th ACM Symposium on Theory of Computing (STOC 97)*, El Paso, TX, USA, Mai 1997.

[CPRW97] CHANG, GEORG J. S., GIRISH PATEL, LIAM RELIHAN und JASON T. L. WANG: *A Graphical Environment for Change Detection in Structured Documents*. In: *Proc. of the 21st International Computer Software and Applications Conference (COMPSAC '97)*, Seiten 75 – 86, Washington, DC, USA, August 1997. IEEE.

[CS98] CLAUSEN, AXEL und ANDREAS SPANIAS: *An Internet-based Computer Laboratory for DSP Courses*. In: *Proc. of the IEEE Frontiers in Education Conference*, Seiten 206–210, Tempe, AZ, USA, November 1998.

[CVM99] CIANCARINI, P., F. VITALI und C. MASCOLO: *Managing Complex Documents over the WWW: A Case Study for XML*. IEEE Transactions on Knowledge and Data Engineering, 11(4):629 – 638, 1999.

[Dat95] DATE, C. J.: *An Introduction to Database Systems*. Addison-Wesley, Reading, Mass, USA, 6. Auflage, 1995.

[DFF+98] DEUTSCH, D., M. FERNANDEZ, D. FLORESCU, A. LEVY und D. SUCIU: *XML-QL: A Query Language for XML*. World Wide Web Consortium (W3C), http://www.w3c.org/TR/NOTE-xml-ql/, 1998.

[DMNS97] DEMEYER, S, T.D. MEIJLER, O. NIERSTRASZ und P. STAYAERT: *Design Guidelines for Tailorable Frameworks*. Communications of the ACM, 40(10):60 – 64, Oktober 1997.

[Dow98] DOWNING, TROY BRYAN: *Java RMI: Remote Method Invocation*. IDG Books Worldwide, Foster City, CA, USA, 1998.

[DW98] DANIELSON, RONALD und SALLY WOOD: *Stimulating Introductory Engineering Courses with Java*. In: *Proc. of the IEEE Frontiers in Education Conference*, Seiten 897–902, Tempe, AZ, USA, November 1998.

[EE98] EDDON, GUY und HENRY EDDON: *Inside Distributed COM*. Microsoft Press, Redmond, USA, 1998.

[EKW00] EL KAHOUI, M. und A. WEBER: *Deciding Hopf Bifurcations by Quantifier Elimination in a Software-Component Architecture*. Journal of Symbolic Computation, 30(2):161 – 179, August 2000.

[ENR97] ELINSON, ALEXEI, DANA S. NAU und WILLIAM C. REGLI: *Feature-based Similarity Assessment of Solid Models*. In: *Proceedings of the fourth Symposium on Solid Modeling and Applications*, Seiten 297–310, Atlanta, GA, USA, Mai 1997.

[Fal01] FALLSIDE, DAVID C.: *XML Schema – Part 0: Primer*. World Wide Web Consortium (W3C), http://www.w3.org/TR/xmlschema-0/, Mai 2001.

[Fat92] FATEMAN, R.: *Honest Plotting, Global Extrema, and Interval Arithmetic*. In: WANG, P. (Herausgeber): *Proc. of the International Symposium on Symbolic and Algebraic Computation (ISSAC '92)*, Berkeley, USA, Juli 1992. ACM Press.

[Fau00] FAULSTICH, RAINER: *Internet Portals for Electronic Comerce*. Diplomarbeit, Universität Karlsruhe, Deutschland, 2000.

[FB99] FARSI, MOHAMMAD und MANUEL BERNARDO BARBOSA: *CANopen: Implementation Made Simple*. Research Studies Press Limited, Baldock, Hertfordshire, UK, 1999.

[FBF+94] FALOUTSOS, CHRISTOS, RON BARBER, MYRON FLICKNER, JIM HAFNER, WAYNE NIBLACK, DRAGUTIN PETKOVIC und WILLIAM EQUITZ: *Efficient and Effective Querying by Image Content*. Journal of Intelligent Information Systems, 3(3/4):231–262, 1994.

[FS97] FAYAD, MOHAMED E. und DOUGLAS C. SCHMIDT: *Object-oriented Application Frameworks*. Communications of the ACM, 40(10):32 – 38, Oktober 1997.

[FZ94] FORMAN, GEORGE H. und JOHN ZAHORJAN: *The Challenges of Mobile Computing*. IEEE Computer, 27(6):38 – 47, 1994.

[GHJV95] GAMMA, ERICH, RICHARD HELM, RALPH JOHNSON und JOHN VLISSIDES: *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley, Reading, Mass, USA, 1995.

[GKBN01] GRUHLER, GERHARD, WOLFGANG KÜCHLIN, DIETER BÜHLER und GERD NUSSER: *Internet-based Lab Assignments in Automation Engineering and Computer Science*. In: SCHILLING, K., H. ROTH und O. ROESCH (Herausgeber): *Proc. of the International Worshop on Tele-Education in Mechatronics Based on Virtual Laboratories*, Ellwangen, Deutschland, Juli 2001. R. Wimmer Verlag.

[GKNB99] GRUHLER, GERHARD, WOLFGANG KÜCHLIN, GERD NUSSER und DIETER BÜHLER: *Internet-basiertes Labor für Automatisierungstechnik und Informatik*. In: SCHMID, DIETMAR (Herausgeber): *Virtuelles Labor: Ihr Draht in die Zukunft*, Seiten 27–36, Ellwangen, Deutschland, Oktober 1999. R. Wimmer Verlag.

[GMD] GMD – *XML* COMPETENCE CENTER, Darmstadt, http://xml.-darmstadt.gmd.de/xql: *GMD-IPSI XQL-Engine*.

[GNBK99] GRUHLER, GERHARD, GERD NUSSER, DIETER BÜHLER und WOLFGANG KÜCHLIN: *Teleservice of CAN Systems via Internet*. In: *Proc. of the 6th International CAN Conference (ICC 99)*, Torino, Italy, November 1999. CAN in Automation (www.can-cia.com).

[GSNK94] GERTZ, M., D. STEWART, B. NELSON und P. KHOSLA: *Using Hypermedia and Reconfigurable Software Aassembly to Support Virtual Laboratories and Factories*. In: *Proc. of the 5th International Symposium on Robotics and Manufacturing (ISRAM '94)*, Seiten 493–500, Maui, Hawaii, USA, August 1994.

[Han] HANISCH, FRANK: *Computergraphik spielend lernen - Java 1.0 3D-Bibliothek*. Graphisch Interaktive Systeme, Eberhard-Karls-Universität, Tübingen, Deutschland, http://www.gris.uni-tuebingen.de/projects/grdev/doc/html/Overview.html.

[Har01] HARDIN, DAVID S.: *Real-Time Objects on the Bare Metal: An Efficient Hardware Realization of the Java Virtual Machine*. In: *Proc. of the 4th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 01)*, Magdeburg, Deutschland, Mai 2001.

[HGMN+97] HAMMER, JOACHIM, HECTOR GARCÍA-MOLINA, SVETLOZAR NESTOROV, RAMANA YERNENI, MARCUS BREUNIG und VASILIS VASSALOS: *Template-based Wrappers in the TSIMMIS System*. In: *Proc. of the ACM SIGMOD International Conference on Management of Data*, Seiten 7–18, Tucson, AZ, USA, 1997.

[HLP99] HUANG, ANDREW C., BENJAMIN C. LING und SHANKAR PONNEKANTI: *Pervasive Computing: What is it Good for?* In: *Proc. of the ACM International Workshop on Data Engineering for Wireless and Mobile Access*, Seiten 84–91, Seattle, WA, USA, August 1999.

[Hoq98] HOQUE, REAZ: *Programming Java Beans 1.1*. Hands-On Web Development. McGraw-Hill, New York, NY, USA, 1998.

[Int86] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION (ISO): *ISO 8879 – Information Processing – Text and Office Systems – Standard Generalized Markup Language (SGML)*, 1986.

[Int93] INTERNATIONAL ORGANIZATION FOR STANDARDIZATION (ISO): *ISO 11898 – Road Vehicles, Interchange of Digital Information - Controller Area Network (CAN) for High-Speed Communication*, 1993.

[Int00] INTERNATIONAL ELECTROTECHNICAL COMMISSION (IEC), www.iec.ch: *Low-Voltage Switch-Gear and Controlgear – Profiles for Networked Industrial Devices, Draft IEC 61915*, Februar 2000.

[IPR98] ITSCHNER, ROBERT, CLAUDE POMMERELL und MARTIN RUTISHAUSER: *GLASS: Remote Monitoring of Embedded Systems in Power Engineering*. IEEE Internet Computing, 2(3):46 – 52, 1998.

[JC92] JACOBSON, IVAR und MAGNUS CHRISTENSEN: *Object-oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, Reading, Mass, USA, 1992.

[JK00] JUNGHWAN, OH und A. HUA KIEN: *Efficient and Cost-Effective Techniques for Browsing and Indexing Large Video Databases*. In:

Proceedings of the 2000 ACM SIGMOD Conference on Management of Data, Seiten 127–132, Dallas, TX, USA, Mai 2000.

[JMM95] JAGADISH, H. V., ALBERTO O. MENDELZON und TOVA MILO: *Similarity-based Queries*. In: *Proc. of the 14th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, Seiten 36 – 45, San Jose, CA, USA, 1995.

[Joh97] JOHNSON, E. RALPH: *Frameworks = (Components + Patterns)*. Communications of the ACM, 40(10):39 – 42, Oktober 1997.

[JS00] JUNGE, T. F. und C. SCHMID: *Web-based Remote Experimentation using a Laboratory-scale Optical Tracker*. In: *Proc. of the 2000 American Control Conference (ACC)*, Seiten 2951–2954, Chicago, Illinois, USA, Juni 2000.

[JWZ94] JIANG, T., L. WANG und K. ZHANG: *Alignment of Trees – An Alternative to Tree Edit*. In: CROCHEMORE, M. und D. GUSFIELD (Herausgeber): *Combinatorial Pattern Matching, Lecture Notes in Computer Science*, Seiten 75 – 86. Springer-Verlag, 1994.

[KGLS97] KÜCHLIN, WOLFGANG, GERHARD GRUHLER, THOMAS LUMPP und ANDREAS SPECK: *HIGHROBOT: Distributed Object-oriented Real-Time Systems*. In: *14th ITG/GI Conference on Architecture of Computer Systems (ARCS '97)*, Rostock, Deutschland, September 1997. VDE-Verlag.

[KGSL97] KÜCHLIN, WOLFGANG, GERHARD GRUHLER, ANDREAS SPECK und THOMAS LUMPP: *HIGHROBOT: A High-performance Universal Robot Control on Parallel Workstations*. In: *Proc. of the IEEE International Conference on the Engineering of Computer Based Systems (ECBS 97)*, Monterey, CA, USA, März 1997. IEEE Computer Society Press.

[KK00] KASTNER, WOLFGANG und CHRISTOPHER KRÜGEL: *A New Approach for Java in Embedded Networks*. In: *Proceedings of the 3rd IEEE Workshop on Factory Communication Systems (WFCS 2000)*, Porto, Portugal, September 2000. IEEE.

[KM93] KILPELÄINEN, PEKKA und HEIKKI MANNILA: *Retrieval from Hierarchical Texts by Partial Patterns*. In: *Proc. of the 16th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, Seiten 214 – 222. ACM Press, Juni/Juli 1993.

[KS88] KRASNER, G. E. und T. P. STEPHEN: *A Cookbook for Using the Model View Controller User Interface Paradigm in Smalltalk-80*. Journal of Object Oriented Programming, 1(3):26 – 49, 1988.

[KWD97] KUSHMERICK, NICKOLAS, DANIEL S. WELD und ROBERT B. DOORENBOS: *Wrapper Induction for Information Extraction*. In: *International Joint Conference on Artificial Intelligence (IJCAI)*, Seiten 729–737, 1997.

[LB99] LAURENT, SIMON S. und ROBERT BIGGAR: *Inside XML DTDs*. Application Development. McGraw-Hill, New York, NY, USA, 1999.

[LGK98] LUMPP, THOMAS, GERHARD GRUHLER und WOLFGANG KÜCHLIN: *Virtual Java Devices—Integration of Fieldbus based Systems in the Internet*. In: *Proc. of the 24th Annual Conference of the IEEE Industrial Electronics Society (IECON '98)*, Aachen, Deutschland, September 1998. IEEE Computer Society Press.

[LHB+99] LIU, LING, WEI HAN, DAVID BUTTLER, CALTON PU und WEI TANG: *An XML-based Wrapper Generator for Web Information Extraction*. In: *International Conf. on Management of Data and Symposium on Principles of Database Systems*, Seiten 540 – 543, Philadelphia, PA, USA, 1999. ACM.

[Lia99] LIANG, SHENG: *The Java Native Interface – Programmer's Guide and Specification*. The Java Series. Addison-Wesley, Reading, Mass, USA, 1999.

[LS99] LASSILA, ORA und RALPH R. SWICK: *Resource Description Framework (RDF) Model and Syntax Specification*. World Wide Web Consortium (W3C), http://www.w3.org/TR/REC-rdf-syntax/, Februar 1999.

[Lum99] LUMPP, THOMAS: *CORBA und Java in der Automatisierungstechnik - Anbindung von Steuerungssystemen an das World-Wide-Web*. Doktorarbeit, Eberhard-Karls-Universität, Tübingen, Deutschland, 1999.

[LYYB96] LEE, YONG KYU, SEONG-JOON YOO, KYOUNGRO YOON und P. BRUCE BERRA: *Index Structures for Structured Documents*. In: *Proceedings of the 1st ACM International Conference on Digital Libraries*, Seiten 91 – 99, Bethesda, MD, USA, März 1996.

[MF00] MARCHIORI, M. und M. FERNANDEZ: *XML Query*. World Wide Web Consortium (W3C), http://www.w3.org/2000/Talks/www9-xmlquery/, 2000.

[Mic97] MICROSOFT CORPORATION (Herausgeber): *Microsoft ODBC 3.0 Software Development Kit and Programmer's Reference*. Microsoft Press, Redmond, USA, Februar 1997.

[MLDea98] MATHEWS, BRIAN, DANIEL LEE, BRIAN DISTER und JOHN BOWLER ET AL.: *Vector Markup Language (VML)*. World Wide Web Consortium (W3C), http://www.w3.org/TR/NOTE-VML, Mai 1998.

[MS96] MUSSER, DAVID R. und ATUL SAINI: *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Addison Wesley Professional Computing Series. Addison-Wesley, Reading Mass, USA, 1996.

[MSS99] MATTOX, DAVID, LEONARD J. SELIGMAN und KENNETH SMITH: *Rapper: A Wrapper Generator with Linguistic Knowledge*. In: *Proc. of the 2nd International Workshop on Web Information and Data Management*, Seiten 6–11, Kansas City, MO, USA, 1999. ACM.

[MT00] MAXIM, M. und S. K. TSO: *Human-Robot Interface Using Agents Communicating in an XML-Based Markup Language*. In: *Proc. of the 9th IEEE International Workshop on Robot and Human Interactive Communication (IEEE RO-MAN 2000)*, Piscataway, NJ, USA, September 2000.

[Myk98] MYKA, ANDREAS: *Fehlertolerante und effiziente automatische Analyse digitalisierter Dokumente zur Gewinnung von Hypertextstrukturen*. Doktorarbeit, Eberhard-Karls-Universität, Tübingen, Deutschland, 1998.

[NBGK01] NUSSER, GERD, DIETER BÜHLER, GERHARD GRUHLER und WOLFGANG KÜCHLIN: *Reality-driven Visualization of Automation Systems via the Internet based on Java and XML*. In: *Proc. of the 1st IFAC Conference on Telematics Applications in Automation and Robotics*, Weingarten, Deutschland, Juli 2001.

[NBY97] NAVARRO, GONZALO und RICARDO BAEZA-YATES: *Proximal Nodes: A Model to Query Document Databases by Content and*

Structure. ACM Transactions on Information Systems, 15(4):400 – 435, 1997.

[Nie99] NIEVA, TAXOMIN: *Automatic Configuration for Remote Diagnosis and Monitoring of Railway Equipments*. In: *Proc. of the 17th IASTED International Conference on Applied Informatics (AI 99)*, Innsbruck, Österreich, 1999.

[Nie01] NIEVA, TAXOMIN: *Remote Data Acquisition of Embedded Systems Using Internet Technologies: A Role-based Generic System Specification*. Doktorarbeit, Swiss Federal Insititute of Technology Lausanne, Lausanne, Schweiz, 2001.

[Obj99] OBJECT MANAGEMENT GROUP (OMG), http://www.omg.org/-techprocess/meetings/schedule/Data_Acquisition_RFP.html: *Data Acquisition from Industrial System - Request For Proposal*, 1999.

[ODL93] OBRACZKA, KATIA, PETER B. DANZIG und SHIH-HAO LI: *Internet Resource Discovery Services*. IEEE Computer, 26(9):8–22, 1993.

[Ome01] OMELAYENKO, BORYS: *Ontology Integration Tasks in Business-to-Business E-Commerce*. In: MONOSTRI, LÁZLÓ, VÁNCZA JÓSEF und ALI MOONIS (Herausgeber): *Engineering of Intelligent Systems – Proc. of the 14th International Conference on Industrial and Engineering Applications of Artificial Intelligence and Expert Systems (IEA/AIE 2001)*, Seiten 119 – 124, Budapest, Ungarn, Juni 2001. Springer-Verlag (LNCS/LNAI).

[OPC98a] OPC TASK FORCE: *OPC Common Definitions and Interfaces*. OPC Foundation, http://www.opcfoundation.org, Oktober 1998. Version 1.0.

[OPC98b] OPC TASK FORCE: *OPC Overview*. OPC Foundation, http://-www.opcfoundation.org, Oktober 1998. Version 1.0.

[OPC99] OPC FOUNDATION: *OPC and Microsoft Start XML Initiative*. OPC Quarterly, 2(4), Dezember 1999.

[OZL96] OOMMEN, B.J., K. ZHANG und W. LEE: *Numerical Similarity and Dissimilarity Measures Between Two Trees*. IEEE Transactions on Computers, 45(12), Dezember 1996.

[Per94] PEREIRA, C. E.: *Real Time Active Objects in C++/Real-Time-UNIX*. In: *Proc. of the ACM SIGPLAN Workshop on Languages, Compiler, and Tool Support for Real-Time Systems (LCT-RTS)*, Orlando, FL, USA, 1994.

[PSZ99] PELILLO, MARCELLO, KALEEM SIDDIQI und STEVEN W. ZUCKER: *Matching Hierarchical Structures Using Association Graphs*. IEEE Transactions on Pattern Analysis and Machine Intelligence, Seiten 889 – 895, November 1999.

[RKV95] ROUSSOPOULOS, N., S. KELLEY und F. VINCENT: *Nearest Neighbor Queries*. In: *Proc. of the ACM International Conference on Management of Data (SIGMOD 95)*, San Jose, CA, USA, Mai 1995.

[RLS98] ROBIE, JONATHAN, JOE LAPP und DAVID SCHACH: *XML Query Language (XQL)*. World Wide Web Consortium (W3C), http://-www.w3c.org/TandS/QL/QL98/pp/xql.html, 1998.

[Rob91] ROBERT BOSCH GMBH, Stuttgart, Deutschland: *CAN Specification 2.0 Part A+B*, 1991.

[RV99] ROBINSON, MATTHEW und PAVEL A. VOROBIEV: *Swing*. Manning Publications Company, Greenwich, Connecticut, USA, Dezember 1999.

[SA99] SAHUGUET, ARNAUD und FABIEN AZAVANT: *Building Light-Weight Wrappers for Legacy Web Data Sources Using W4F*. In: *Proc. of 25th International Conference on Very Large Data Bases (VLDB)*, Edinburgh, Schottland, 1999.

[Sch98a] SCHMID, CHRISTIAN: *New Trends in Control Engineering Education*. In: KOZAK, S. und M. KIDLINGTON HUBA (Herausgeber): *2nd IFAC Workshop on New Trends in Design and Control Systems*, Seiten 127–132, Smolenice, Slovakei, September 1998.

[Sch98b] SCHMIDT, UWE: *Teleservice - Stand und Entwicklungsansätze*. Bundesministerium für Bildung und Forschung, http://www.iid.de/-informationen/teleservice/, 1998.

[Sch99] SCHMID, CHRISTIAN: *Simulation and Virtual Reality for Education on the Web*. In: *Proc. of the EUROMEDIA '99*, Seiten 181–8, San Diego, CA, USA, April 1999.

[Sed92] SEDGEWICK, ROBERT: *Algorithmen in C++*. Addison-Wesley, Reading, Mass, USA, 1992.

[SEKN92] SCHWARTZ, MICHAEL F., ALAN EMTAGE, BREWSTER KAHLE und B. CLIFFORD NEUMAN: *A Comparison of Internet Resource Discovery Approaches*. Computing Systems, 5(4):461–493, 1992.

[SHKK00] SCHIMKAT, RALF-DIETER, MATTHIAS HÄUSSER, WOLFGANG KÜCHLIN und RAINER KRAUTTER: *Web Application Middleware to Support XML-Based Monitoring in Distributed Systems*. In: *Proc. of the ISCA 13th International Conference on Computer Applications in Industry and Engineering (CAINE-2000)*, Honolulu, Hawaii, USA, November 2000.

[SN00] SCHLIEDER, T. und F. NAUMANN: *Approximate Tree Embedding for Querying XML Data*. In: *ACM SIGIR Workshop on XML and Information Retrieval*, Athen, Griechenland, Juli 2000.

[SNB00] SCHIMKAT, RALF-DIETER, GERD NUSSER und DIETER BÜHLER: *Scalability and Interoperability in Service-Centric Architectures for the Web*. In: *Proc. of the 11th International Workshop on Database and Expert Systems Applications (DEXA 2000)*, London, UK, September 2000. IEEE Computer Society Press.

[SNM01] SNMP RESEARCH INTERNATIONAL, http://www.snmp.com/-snmpv3/v3white.html: *SNMPv3 White Paper*, 2001.

[Spe00] SPECK, ANDREAS: *Component-Based Control System*. In: *Proc. of the 7th IEEE International Conference on the Engineering of Computer Based Systems (ECBS 2000)*. IEEE Computer Society Press, April 2000.

[SRR01] SCHILLING, K., H. ROTH und O. ROESCH (Herausgeber): *Tele-Education in Mechatronics Based on Virtual Laboratories*. R. Wimmer Verlag, Ellwangen, Deutschland, Juli 2001.

[Suna] SUN MICROSYSTEMS, http://java.sun.com/xml/xml_jaxp.html: *Java API for XML Processing*.

[Sunb] SUN MICROSYSTEMS, http://java.sun.com/products/javacomm/: *Java Communications API*.

[Sun98] SUN MICROSYSTEMS, http://java.sun.com/products/jdk/1.2/docs/-guide/rmi/spec/rmiTOC.doc.html: *Remote Method Invocation Specification*, 1998.

[Sun00] SUN MICROSYSTEMS, http://jcp.org/aboutJava/community-process/final/jsr003/index.html: *Java Management Extensions – Instrumentation and Agent Specification, v1.0*, Juli 2000.

[Syn00] SYNCML.ORG, http://www.syncml.org/docs/syncml_represent_-v10_20001207.pdf: *SyncML Representation Protocol, Version 1.0*, 2000.

[Tai79] TAI, K. C.: *The Tree-to-Tree Correction Problem*. Journal of the ACM, 26(3):422–433, Juli 1979.

[Tan96] TANENBAUM, ANDREW S.: *Computer Networks*. Prentice Hall, 3. Auflage, 1996.

[TBMM01] THOMPSON, HENRY S., DAVID BEECH, MURRAY MALONEY und NOAH MENDELSOHN: *XML Schema – Part 1: Structures*. World Wide Web Consortium (W3C), http://www.w3.org/TR/xmlschema-1/, Mai 2001.

[The96] THE UNICODE CONSORTIUM: *The Unicode Standard, Version 2.0*. Addison Wesley Developers Press, Reading, Mass, USA, 1996.

[Ues01] UESBECK, MECHTHILD: *Effiziente Strategien und Werkzeuge zur Generierung und Verwaltung von e-Learning-Systemen*. Doktorarbeit, Eberhard-Karls-Universität, Tübingen, Deutschland, 2001.

[WFC+99] WHITE, SETH, MAYDENE FISHER, RICK CATTELL, GRAHAM HAMILTON und MARK HAPNER: *JDBC API Tutorial and Reference - Universal Data Access for the Java 2 Platform*. Addison-Wesley, Reading, Mass, USA, 2. Auflage, 1999.

[Wir99] WIRELESS APPLICATION PROTOCOL FORUM LTD., http://www.-wapforum.org/DTD/wml_1.1.xml: *Wireless Markup Language (WML) Document Type Definition*, 1999.

[WKE98] WEBER, A., W. KÜCHLIN und B. EGGERS: *Parallel Computer Algebra Software as a Web Component*. Concurrency: Practice and Experience, 10(11–13):1179–1188, 1998.

[Wor98a] WORLD WIDE WEB CONSORTIUM (W3C), http://www.w3c.org/-TR/REC-DOM-Level-1/: *Document Object Model (DOM) Level 1 Specification*, 1998.

[Wor98b] WORLD WIDE WEB CONSORTIUM (W3C), http://www.w3c.org/-TR/REC-xml: *Extensible Markup Language (XML) 1.0*, 1998.

[Wor99a] WORLD WIDE WEB CONSORTIUM (W3C), http://www.w3c.org/-TR/DOM-Level-2/: *Document Object Model (DOM) Level 2 Specification*, 1999.

[Wor99b] WORLD WIDE WEB CONSORTIUM (W3C), http://www.w3.org/-TR/html4/: *HTML 4.01 Specification*, Dezember 1999.

[Wor99c] WORLD WIDE WEB CONSORTIUM (W3C), http://www.w3.org/-TR/xslt.html: *XSL Transformations (XSLT), 1.0, W3C Recommendation*, November 1999.

[Wor00a] WORLD WIDE WEB CONSORTIUM (W3C), http://www.w3c.org/-TR/xsl: *Extensible Stylesheet Language (XSL), 1.0, Candidate Recommendation*, November 2000.

[Wor00b] WORLD WIDE WEB CONSORTIUM (W3C), http://www.w3.org/-TR/xhtml1/: *XHTML 1.0: The Extensible HyperText Markup Language - A Reformulation of HTML 4 in XML 1.0*, Januar 2000.

[Wor01a] WORLD WIDE WEB CONSORTIUM (W3C), http://www.w3.org/-TR/DOM-Level-3-Core/: *Document Object Model (DOM) Level 3 Core Specification (W3C Working Draft)*, Juni 2001.

[Wor01b] WORLD WIDE WEB CONSORTIUM (W3C), http://www.w3.org/-TR/MathML2/: *Mathematical Markup Language (MathML) Version 2.0*, Februar 2001.

[Wor01c] WORLD WIDE WEB CONSORTIUM (W3C), http://www.w3.org/-TR/xptr: *XML Pointer Language (XPointer), Last Call Working Draft*, Januar 2001.

[Wor01d] WORLD WIDE WEB CONSORTIUM (W3C), http://www.w3.org/-TR/xmlschema-2/: *XML Schema – Part 2: Datatypes*, Mai 2001.

[WZJS94] WANG, JASON TSONG-LI, KAIZHONG ZHANG, KARPJOO JEONG und DENNIS SHASHA: *A System for Approximate Tree Matching*. IEEE Transactions on Knowledge and Data Engineering, 6(4):559–571, 1994.

[Yia93] YIANILOS, P. N.: *Data Structures and Algorithms for Nearest Neighbor Search in General Metric Spaces*. In: *Proc. of the ACM-SIAM Symposium on Discrete Algorithms*, Austin, TX, USA, Januar 1993.

[ZS89] ZHANG, KAIZHONG und DENNIS SHASHA: *Simple Fast Algorithms for the Editing Distance Between Trees and Related Problems*. SIAM Journal on Computing, 18(6):1245–1262, 1989.

[ZWS95] ZHANG, K., J. T. L. WANG und D. SHASHA: *On the Editing Distance Between Undirected Acyclic Graphs and Related Problems*. In: GALIL, Z. und E. UKKONEN (Herausgeber): *Proceedings of the 6th Annual Symposium on Combinatorial Pattern Matching*, Seiten 395–407, Espoo, Finnland, 1995. Springer-Verlag.

Lebenslauf

Persönliche Daten

Name	Dieter Bühler
Geburtstag	09.06.1971
Geburtsort	Überlingen
Familienstand	ledig
Staatsangehörigkeit	deutsch

Ausbildung und beruflicher Werdegang

1977 - 1981	Besuch der Grundschule
1981 - 1990	Besuch des Leibniz-Gymnasiums in Rottweil, Abitur Mai 1990
1990 - 1992	Zivildienst in Rottweil
Okt. 1992 - Sept. 1998	Studium der Informatik mit Nebenfach Biologie an der Universität Tübingen
Okt. 1995 - Sept. 1998	Wissenschaftliche Hilfskraft am Wilhelm-Schickard-Institut für Informatik der Universität Tübingen in den Arbeitsbereichen *Computer Algebra* und *Datenbanken und Informationssysteme*
Sept. 1998	Diplom (mit Auszeichnung)
Okt. 1998 - Nov. 2001	Wissenschaftlicher Angestellter am Arbeitsbereich *Symbolisches Rechnen* des Wilhelm-Schickard-Instituts für Informatik der Universität Tübingen (Prof. Dr. W. Küchlin)