Diplomarbeit

Entwurf und Implementierung einer retargierbaren Programmiersprache für fernprogrammierbare Roboter

Andreas Wagner
andwagne@studcs.uni-sb.de

Betreuer: Prof. Dr. Stephan Diehl

Fachbereich Informatik
Lehrstuhl Prof. Dr. R. Wilhelm

19. Juli 2005

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Abstract

In this diploma thesis, we develop and implement a *XML*-based programming language (*roboXL*). *roboXL* allows for changing the programming of *Java*-capable robots over the network at runtime. In this context, robots (devices) are primarily programmable micro controllers like the Lego Mindstorms RCX. The crucial factor is, that the device has sensors and motors (effectors) and that it is *Java*-capable. *Java*-capable means, that a *Java Virtual Machine* is available for the device. *roboXL* programs are sent over the internet to a server. A compiler on serverside translates a *roboXL* program to a *Java* program for the specific robot. The *Java* program is then transfered to and executed by the robot. Furthermore, *roboXL* allows for reconfiguring of motors and for requesting sensors and variables at runtime of a generated *Java* program.
To make *roboXL* usable for different robots (retargetable), the device specific properties of each robot, you want to program with *roboXL*, are specified using a *XML*-based device specification language. The device specification is processed by a so-called Retargeting Generator.
To transform *roboXL* and the *XML*-based device specification language, we combine *XSLT* and aspect-oriented programming. Both the Retargeting Generator and the compiler are implemented by *XSLT* stylesheets that produce *Inject/J* scripts (aspects). The *Java* program, which is transfered to and executed by the robot, is generated out of these aspects.

1.2 Zusammenfassung

Diese Diplomarbeit befasst sich mit der Entwicklung und Implementierung einer *XML*-basierten Programmiersprache (*roboXL*), mit deren Hilfe die Programmierung von *Java*-fähigen Robotern zur Laufzeit über das Netzwerk geändert werden kann. Unter Robotern (Devices) versteht man in diesem Zusammenhang in erster Linie programmierbare Mikrocontoller wie den Lego Mindstorms RCX. Entscheidend ist, dass das Device Sensoren und Motoren (Effektoren) zur Verfügung stellt und *Java*-fähig ist. *Java*-fähig bedeutet, dass

eine *Java Virtual Machine* für das Device vorhanden ist. *roboXL* Programme werden über das Internet zu einem Server gesendet. Ein Compiler auf Serverseite übersetzt ein *roboXL* Programm in ein *Java* Programm für den entsprechenden Roboter. Das erzeugte *Java* Programm wird auf den Roboter geladen und ausgeführt. Mit *roboXL* ist es darüber hinaus möglich, zur Laufzeit eines solchen Programms Motoren neu zu konfigurieren und Sensoren bzw. Variablen abzufragen.
Um zu erreichen, dass *roboXL* für verschiedene Roboter einsetzbar (retargierbar, von engl. `retargetable`) ist, werden die spezifischen Eigenschaften jedes Roboters, den man mit *roboXL* programmieren möchte, über eine *XML*-basierte Devicespezifikationssprache angegeben. Die Devicespezifikation wird durch einen sogenannten Retargeting Generator verarbeitet.
Zur Transformation von *roboXL* und der *XML*-basierten Devicespezifikationssprache werden *XSLT* und Aspektorientierte Programmierung kombiniert. Sowohl der Retargeting Generator als auch der Compiler werden in Form von *XSLT* Stylesheets implementiert, die *Inject/J* Skripte (Aspekte) erzeugen. Aus diesen erzeugten Aspekten entsteht letztenendes das *Java* Programm, das auf den Roboter geladen und ausgeführt wird.

1.3 Aufgabenstellung

Ziel dieser Diplomarbeit ist die Entwicklung und Implementierung einer retargierbaren, *XML*-basierten Programmiersprache für fernprogrammierbare Roboter sowie des entsprechenden Compilers. Die Sprache trägt den Namen *roboXL*. Ein Roboter ist im Rahmen dieser Arbeit dabei einfach als Device definiert, das Sensoren und Motoren (Effektoren) zur Verfügung stellt. Der Begriff Roboter ist demnach nicht nach den Definitionen zu interpretieren, die in der klassischen Literatur zu finden sind. Im Folgenden werden die Begriffe Device und Roboter synonym gebraucht. Retargierbar (von engl. `retargetable`) bedeutet, dass die Programmiersprache für verschiedene Devices einsetzbar ist. Voraussetzung ist, dass eine geeignete *Java Virtual Machine* für das Device verfügbar ist. Zu entwerfen ist zum einen die *XML*-Programmiersprache *roboXL* und zum anderen eine *XML*-basierte Devicespezifikationssprache, mit der die individuellen Eigenschaften eines Devices spezifiziert werden können.
Mit Hilfe von *XSLT* und Aspektorientierter Programmierung (AOP) werden *roboXL* Programme in *Java* Programme für das Device übersetzt. Die *roboXL* Programme werden im Folgenden als ***roboXL* Kontrollprogramme** bezeichnet, da sie tatsächlich auf dem Device ablaufen und es direkt kontrollieren. Darüber hinaus soll es möglich sein, auf ein *roboXL* Kontrollprogramm zur Laufzeit zuzugreifen, d.h. Werte von Sensoren, Motoren und Variablen abzufragen und Motoren neu zu konfigurieren. Dazu werden sogenannte ***roboXL* Monitoringprogramme** erstellt, bei deren Verarbeitung ein Datenaustausch mit dem Device durchgeführt wird.
Die Programmierung der Devices soll direkt über das Internet erfolgen. Dazu wird die gesamte Funktionalität in eine Web Applikation eingebettet, die das Retargeting, das Compilieren und den Download der Kontrollprogramme auf das Device, sowie die

Verarbeitung der Monitoringprogramme übernimmt. Als Referenzdevice wird der Lego Mindstorms RCX [16] verwendet.

1.4 Technische Grundlagen

1.4.1 *XML*

Die *Extendable Markup Language* (*XML*) ist ein einfaches und zugleich sehr flexibles Textformat. *XML* spielt eine immer wichtigere Rolle beim Austausch verschiedenster Daten im Web und anderswo [12]. Seine Leistung besteht darin, dass man mit den Konzepten und Regeln, die es bereitstellt, eigene Sprachen definieren kann, die ähnlich funktionieren wie *HTML*. All diese Sprachen bestehen immer wieder aus Elementen, markiert durch Tags, deren Verschachtelungsregeln, und aus Attributen denen Werte zugewiesen werden können [18]. Um die Elemente und Attributnamen, die in *Extensible Markup Language*-Dokumenten verwendet werden können, eindeutig zu benennen, werden *XML*-Namensräume verwendet. Die Element- und Attributnamen werden mit Namensräumen verknüpft, die durch URI-Verweise identifiziert werden [8].
Dazu ein erstes vereinfachtes Beispiel aus der *XML*-Programmiersprache *roboXL*:

```
<?xml version="1.0"?>
<ROBOXL>
  <LOOP counter="10">
    <SENSOR id="1" type="int"/>
  </LOOP>
</ROBOXL>
```

Die *XML*-Deklaration in der erste Zeile stellt den Bezug zu *XML* her. Eine Regel in *XML* ist, dass jedes Dokument ein äusseres Tag enthält, das sogenannte Dokument-Element. Alle anderen Tags dürfen nur unterhalb des Dokument-Elements auftreten. Hält ein Dokument sämtliche Regeln für *XML* ein, so spricht man von einem wohlgeformten *XML*-Dokument. Das Dokument-Element im obigen Beispiel ist `ROBOXL`. Innerhalb von `ROBOXL` werden zwei weitere Elemente verwendet. Das Element `LOOP`, markiert durch das Tag `<LOOP>`, definiert eine Schleife. Das Attribut `counter` gibt die Anzahl der Schleifendurchläufe an. Die Verschachtelungsregeln für *roboXL* besagen nun, dass innerhalb des Tags `<LOOP>` das Tag `<SENSOR>` auftreten darf. Das Element `SENSOR` dient dem Lesen eines Sensors. Das Attribut `id` gibt an, dass der Sensor am ersten Sensoranschluss des Devices gelesen werden soll. Das Attribut `type` gibt an, dass der gelesene Sensorwert von Typ int ist. Das Beispiel definiert also, dass der Sensor zehnmal hintereinander abgefragt wird.

1.4.2 *XML Schema*

Wie in Abschnitt 1.4.1 gesehen, bestehen *XML*-Sprachen aus Elementen, die wiederum Attribute besitzen können und Verschachtelungsregeln, die definieren, welche Kindelemente jedes Element haben darf. Die Frage, die sich stellt ist, wie und wo werden diese Bestandteile definiert? Genau diese Aufgabe übernimmt *XML Schema*.
XML Schema ist eine Empfehlung des *World Wide Web Consortiums (W3C)*. *XML Schema* ermöglicht eine detaillierte Definition der Struktur und des Inhalts von *XML*-Dokumenten [12]. Ein konkretes *XML Schema* wird auch als eine *XML Schema Definition (XSD)* bezeichnet und hat üblicherweise die Dateiendung ".xsd". Neben *XML Schema* gibt es jedoch auch die weitverbreiteten *Document Type Definitions (DTDs)*. *XML Schema* bieten mehrere Vorteile im Vergleich zu *DTDs*.

- Eine *XML Schema Definition* hat selbst auch die Form eines *XML*-Dokuments. *DTDs* hingegen verwenden eine eigene Syntax, erfordern also das Erlernen einer weiteren Sprache.
- *XML Schema* bietet Vererbungstechniken an. *XML*-Sprachen lassen sich dadurch leicht erweitern.
- *XML Schema* unterstützt eine große Anzahl von Datentypen.

Es ist anzunehmen, dass *DTDs* einmal vollständig von *XML Schema* abgelöst werden [22].
Kommen wir nun noch einmal auf das Beispiel aus Abschnitt 1.4.1 zurück. Das dort angegebene *XML*-Dokument ist wohlgeformt. Wir möchten das Dokument nun aber auch auf Gültigkeit prüfen (validieren), d.h. überprüfen ob alle Elemente und Attribute auch tatsächlich Bestandteile der *XML*-Sprache *roboXL* sind und ob die Elemente den Verschachtelungsregeln entsprechen. Es muss also der Bezug zu einem Namensraum angegeben werden, der wiederum in einer *XML Schema Definition* definiert ist. Nehmen wir einmal an, die *XML*-Sprache *roboXL* sei im Namensraum roboXL definiert.

```
 <?xml version="1.0"?>
<ROBOXL xmlns ="roboXL"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="roboXL
                            roboXL.xsd">
  <LOOP counter="10">
    <SENSOR id="1" type="int"/>
  </LOOP>
</ROBOXL>
```

Alle Angaben zu den Namensräumen werden im Dokument-Element gemacht. In unserem Beispiel ist als Standard-Namensraum (Attribut xmlns) roboXL angegeben. Dies besagt, dass alle in diesem *XML*-Dokument verwendeten Elemente aus dem Namensraum roboXL

stammen. Möchte man ein *XML Schema* in einem *XML*-Dokument verwenden, benutzt man die Attribute xmlns:xsi und xsi:schemaLocation, um das *XML Schema* einem Namensraum zuzuweisen. Im Beispiel ist angegeben, dass das *XML Schema* für den Namensraum roboXL in der Datei roboXL.xsd zu finden ist. Wir setzen das Beispiel mit der in roboXL.xsd angegebenen *XML Schema Definition* fort, auch hier wieder in vereinfachter Form:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="roboXL"
            xmlns="roboXL">
<xsd:element name="ROBOXL">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="LOOP"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:element name="LOOP">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="SENSOR" minOccurs="0" maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="counter" type="xsd:nonNegativeInteger"
                   use="required"/>
  </xsd:complexType>
</xsd:element>

<xsd:element name="SENSOR">
  <xsd:complexType>
    <xsd:attribute name="id" type="xsd:integer" use="required"/>
    <xsd:attribute name="type" use="required">
      <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:enumeration value="int"/>
              <xsd:enumeration value="float"/>
              <xsd:enumeration value="bool"/>
            </xsd:restriction>
      </xsd:simpleType>
    </xsd:attribute>
  </xsd:complexType>
</xsd:element>
</xsd:schema>
```

Eine *XML Schema Definition* hat immer das Dokument-Element `xsd:schema`. Zu beachten ist, dass hier zwei verschiedene Namensräume verwendet werden. Einmal der Standard-Namensraum roboXL (alle Elemente, die ohne Prefix angesprochen werden entstammen diesem Namensraum). Desweiteren der Namensraum, der die Elemente der *XML Schema Definition* enthält (xmlns:xsd). Alle Elemente der *XML Schema Definition* werden deshalb mit dem Prefix xsd angesprochen. Mit dem Attribut targetNamespace wird der Namensraum festgelegt, dem die hier definierten Elemente zugeordnet werden sollen.
Ein Element wird über das Tag `<xsd:element>` definiert. Über das optionale Attribut minOccurs kann angegeben werden, wie häufig ein Element mindestens vorkommen muss (maxOccurs analog). Sobald ein Element Kindelemente bzw. Attribute besitzen soll, benötigt man `<xsd:complexType>`. `<xsd:sequence>` definiert eine Sequenz von Elementen, d.h. alle dort angegebenen Elemente müssen in exakt dieser Reihenfolge auftreten. Ein Attribut wird über das Tag `<xsd:attribute>` definiert. Es müssen ein Name und der Typ des Attributs angegeben werden. use ist optional. Will man einen Typ definieren der eine Redefinition eines schon vorhandenen Typs ist, so benötigt man das Tag `<xsd:simpleType>`. Im Beispiel wird der Typ xsd:string dabei auf die Strings int,float und bool eingeschränkt (`<xsd:restriction base="xsd:string">`). Man beachte, dass die momentane Defintion der Sprache *roboXL* lediglich erlaubt, eine Schleife zu definieren, in der beliebig viele Sensorwerte abgfragt werden können.
Eine ausführliche Beschreibung der Verwendung von *XML Schema* würde über den Rahmen dieser Arbeit hinausgehen. Es sei daher auf das Tutorial von R.L.Costello verwiesen [10], das sich eingehends mit *XML Schema* beschäftigt. Dieses Tutorial und weitere nützliche Materialien sind auf den Seiten des *W3C* zu finden [12].

1.4.3 *XSLT*

XSLT ist der wichtigste Teil der *Extensible Stylesheet Language (XSL)* Standards. *XSL* ist eine Gruppe von Empfehlungen des *W3C* zur Definition der Transformation und Presentation von *XML*-Dokumenten. Die beiden anderen Standards sind *XML Path Language (XPath)* und *XSL Formatting Objects (XSL-FO)* [12].
XSLT (XSL Transformations) ist eine Programmiersprache zur Transformation von *XML*-Dokumenten. Sie baut auf die logische Baumstruktur eines *XML*-Dokuments auf und erlaubt die Definition von Umwandlungsregeln. *XSLT* Programme, sogenannte *XSLT* Stylesheets, sind dabei ebenfalls nach den Regeln des *XML*-Standards aufgebaut. Spezielle *XSLT* Prozessoren lesen *XSLT* Stylesheets ein und transformieren eine *XML*-Eingabedatei nach den Stylesheet-Regeln in das gewünschte Ausgabeformat. Eine Transformation besteht dazu aus einer Reihe von einzelnen Transformationsregeln, genannt Templates [23].
XSLT benutzt *XPath*, um Informationen in einem *XML*-Dokument zu finden. *XPath* wird zum Navigieren durch die Elemente und Attribute eines *XML*-Dokuments benutzt. Die Templates benutzen *XPath*, um zu beschreiben, für welche Knoten sie gelten. Die Knoten werden dann in den im Template angegebenen Inhalt transformiert.

Sehen wir uns als Beispiel einmal an, wie das <LOOP> Tag aus unserem Beispiel in ein Schleifenkonstrukt aus *Java* übersetzt werden kann:

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0"
                xmlns:robo="roboXL">

  <xsl:output method="text"/>

  <xsl:template match="/robo:ROBOXL/robo:LOOP">
    <xsl:choose>
      <xsl:when test="@counter=0">
         while(true)
         {
      </xsl:when>
      <xsl:otherwise>
         for(int i=0;i&lt;<xsl:value-of select="@counter"/>;i++)
         {
      </xsl:otherwise>
    </xsl:choose>

    <xsl:for-each select="*">
      ...
    </xsl:for-each>
    }
  </xsl:template>

</xsl:stylesheet>
```

Verwendet werden zwei Namensräume. xmlns:xsl enthält die Elemente von *XSLT*. xmlns:robo enthält die *roboXL* Elemente. *XSLT* erlaubt die Transformation eines *XML*-Dokuments in ein *HTML*-, *XML*- bzw. Text-Dokument. Ein Text-Dokument kann dabei z.B. auch eine *Java* Datei sein. Das Ausgabeformat wird mit dem Element `xsl:output` über das Attribut method festgelegt. Mit Hilfe des *XPath* /robo:ROBOXL/robo:LOOP wird angegeben, dass sich das Template auf das <LOOP> Element bezieht. Mittels <xsl:choose> wird danach eine Fallunterscheidung definiert. Dabei wird der Wert des Attributs counter überprüft. Falls der Wert 0 ist, erzeugen wir in *Java* eine *while*-Endlosschleife, andernfalls eine *for*-Schleife. <xsl:choose> folgt <xsl:for-each>, das jedoch keine Anweisungen enthält. Würde man dort weitere Verarbeitungsschritte definieren, so würden diese auf jedes Kindelement (select="*") von <LOOP>, also somit die Sensorabfragen, angewendet. Um das Beispiel knapp zu halten, wurde darauf verzichtet.

Für *XSLT* gilt das gleiche wie schon für *XML Schema*: Eine ausführliche Beschreibung der Verwendung von *XSLT* würde über den Rahmen dieser Arbeit hinausgehen. Es sei auch hier auf ein Tutorial von R.L.Costello verwiesen [11], das sich eingehends mit *XSLT* beschäftigt. Dieses Tutorial und weitere nützliche Materialien sind auf den Seiten des *W3C* zu finden [12].

1.4.4 *Aspektorientierte Programmierung (AOP)*

Aspektorientierte Programmierung (AOP) ist eine relativ neue Programmiertechnik. Es hat sich gezeigt, dass einige wichtige Punkte beim Entwurf vieler programmiertechnischer Probleme weder durch prozedurale noch durch *Objektorientierte Programmierung (OOP)* hinreichend abgedeckt werden können. Dies führt dazu, dass die Implementierung dieser Teile im gesamten Code verstreut sind. Das Problem liegt also in der Überschneidung mit der gesamten Grundfunktionalität des Systems. *AOP* erlaubt es die Eigenschaften solcher kritischer Punkte beim Entwurf in sogenannten **Aspekten** zu definieren [15].
Aspekte sind modulübergreifende Sachverhalte, die in komplexen Systemen ständig auftreten. Ein Beispiel wäre ein Logging Mechanismus, der Bestandteil von nahezu allen Klassen eines mit *OOP* entwickelten Systems ist. Führt man nun ein neues Loggin Format ein, so müsste man Änderungen in all diesen Klassen durchführen. Weitere Beispiele wären Synchronisation oder Zugriffskontrolle.
AOP führt Aspekte als eigene syntaktische Strukturen ein [24]. Es gibt somit Aspektsprachen mit denen sich die Aspekte definieren lassen. Bleiben wir einmal bei einem mit *OOP* entwickelten System. Die Programmiersprache mit der ein solches System entwickelt wurde bezeichnet man als Komponentensprache. Aspekte werden mit Hilfe des sogenannten **Aspect Weaver** in den Quellcode des Systems eingewebt.

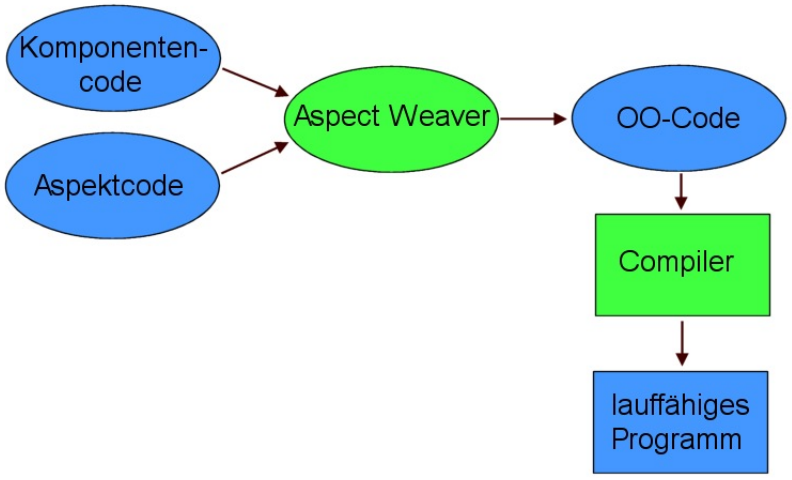


Abbildung 1.1: Aspect Weaver

Der Aspect Weaver orientiert sich dabei an sogenannten **Join Points**. Das sind diejenigen Elemente der Komponentensprache, mit denen der Aspect Weaver interagiert. Die im Rahmen dieser Arbeit verwendete Aspektsprache ist *Inject/J*. *Inject/J* stellt einen Aspect Weaver für die Programmiersprache *Java* zur Verfügung [13].
Schauen wir uns nun an, wie ein mit *Inject/J* verfasster Aspekt aussieht, der den oben beschriebenen Logging Mechanismus für die Programmiersprache *Java* implementiert. Jede Methode jeder Klasse unseres Systems soll, wenn sie aufgerufen wird, ihre Signatur ausgeben. Anstatt dies mühsam in jede Methode aufzunehmen, definieren wir einfach folgenden Aspekt:

```
script logging {
  foreach class '*' do {
    foreach method '*(*)' <=m> do {
      before ${
        System.out.println("<m.signature>");
        }$;
    }
  }
}
```

Die Aspekte werden bei *Inject/J* in sogenannten Skripten definiert (`script logging`). In jeder Klasse unseres Systems (`foreach class '*'`), fügen wir in jeder Methode (`foreach method '*(*)'`) eine Ausgabe ein, die uns die Signatur auf dem Standardout ausgibt. *Inject/J* stellt dafür ein vordefiniertes Konstrukt zur Verfügung. Jede Methode wird temporär in der Variablen `m` abgespeichert. Mit `<m.signature>` wird automatisch deren Signatur ermittelt. Das Schlüsselwort `before` bewirkt, dass der im Aspekt definierte Code vor allen anderen Anweisungen eingewebt wird, die bereits im Rumpf der Methode definiert sind. Weitere wichtige Konstrukte von *Inject/J* sind der Zugriff auf einzelne Methoden/Klassen (`in method/class do`) aber auch das Hinzufügen von Importanweisungen, Membervariablen oder Methoden (`add to imports/members`).

1.5 Übersicht

roboXL ist eine *XML*-Programmiersprache für Roboter. Die grundlegende Frage, die sich stellt ist, warum gerade *XML* verwendet wird? Ein primäres Ziel der Arbeit ist es zu erreichen, dass die Programmierung von Robotern über das Netzwerk verändert werden kann. Ein Server stellt die Verbindung zu einem Roboter über eine Web Applikation her. Das bedeutet, dass die Clients über ein Netzwerk Daten mit dem Server austauschen, wie z.B. ein neues Kontrollprogramm für einen Roboter. Hier kommen nun die Vorteile von *XML* zum tragen. *XML* ist inzwischen ein weitverbreiteter Standard für die Darstellung und den Austausch von Daten. Das *XML*-Format ermöglicht einen simplen und effizienten Datenaustausch zwischen Server und Client. Hinzu kommt, dass ein Client zum Erstellen

von *roboXL* Kontroll- oder Monitoringprogrammen lediglich einen Editor zum Erstellen von *XML*-Dokumenten benötigt. Zur Erstellung und Bearbeitung von *XML*-Dokumenten existieren mittlerweile unzählige Tools. Ein zusätzlicher Compiler ist nicht erforderlich. Die Verarbeitung von *roboXL* erfolgt auf dem Server und betrifft einen Client somit nicht. Alle *roboXL* Kontrollprogramme werden letztendlich serverseitig in *Java* Code umgewandelt, der auf den Roboter geladen wird. Es liegt daher auf der Hand, beim Design von *roboXL* nahe bei den Konstrukten der Programmiersprache *Java* zu bleiben. Doch auch viele Konzepte von Lego's visueller Programmiersprache *Robolab* werden wiederverwendet. *roboXL* stellt Konstrukte zur sequentiellen, parallelen, wiederholten und bedingten Ausführung zur Verfügung und bietet Benutzern damit die Möglichkeit auch komplexe Programme zu erstellen.

Natürlich ist *roboXL* nicht die erste *XML*-basierte Sprache für Roboter. Es gibt beispielsweise die Sprache *RoboML*, die wesentlich komplexer als *roboXL* ist und deren Hauptaugenmerk darauf liegt, eine gemeinsame Sprache für die Datendarstellung und den Datenaustausch zwischen verschiedenen Robotern zu definieren [17].

roboXL ist eine retargierbare Programmiersprache. Bevor ein Client *roboXL* Kontroll- bzw. Monitoringprogramme für ein bestimmtes Device an den Server senden kann, muss serverseitig eine Spezifikation des Devices angelegt werden. Dazu dient die Devicespezifikationssprache. Mit ihr können die individuellen Eigenschaften eines Devices spezifiziert werden. Dies muss für jedes Device einmal durchgeführt werden. Die Devicespezifikationssprache ist ebenfalls eine *XML*-Sprache.

roboXL und die Devicespezifikationssprache werden mit Hilfe von *XML Schema Definitions (XSD)* definiert (s. 1.4.2). Die Entscheidung für *XML Schema* resultiert aus entscheidenden Vorteilen, die *XML Schema* gegenüber *DTDs* bieten. Zum einen die Vererbungstechniken. Diese werden beim Retargeting, d.h. der Anpassung an das aktuelle Device, eingesetzt. Zum anderen die Möglichkeit der Definition verschiedenster Datentypen. So lässt sich beispielsweise definieren, dass ein *XML*-Element nur einen Integer im Bereich 1 bis 7 enthalten darf.

Ein zentraler Teil dieser Arbeit ist neben dem Entwurf von *roboXL* und der Devicespezifikationssprache die Entwicklung eines *roboXL* Compilers. Grundsätzlich gibt es mindestens drei Ansätze, um *roboXL* Kontrollprogramme in *Java* Programme für das jeweilige Device zu übersetzen:

- **Spezifischer Compiler**
  Der Compiler arbeitet nur mit einem bestimmten Device zusammen. Aus einem *roboXL* Kontrollprogramm wird ein devicespezisches *Java* Programm generiert. Folgt man diesem Ansatz, so ist es offensichtlich, dass n Compiler für n verschiedene Devices geschrieben werden müssen.

- **Spezifischer Kernel + Generischer Compiler**
  Dieser Ansatz trennt die devicespezifischen Teile der Implementierung von den allgemeinen. Der generische Compiler erzeugt *Java* Code, der die in einem spezifischen Kernel *Java* Programm implementierte devicespezifische Funktionalität anspricht. Bei diesem Ansatz müssten demnach nur ein Compiler, jedoch n spezifische Kernel für n verschiedene Devices implementiert werden.

- **Generierter Kernel + Generischer Compiler**
  Wiederum wird ein generischer Compiler verwendet. Hinzu kommt, dass die generischen (wiederverwendbaren) und spezifischen Teile des Kernels getrennt werden. Die generischen Teile des Kernels werden als eigenständiger generischer Kernel definiert, die spezifischen Teile in einer Devicespezifikation. Dieser Ansatz erlaubt es, das Device zu wechseln, ohne dass der Kernel neu geschrieben werden muss. Stattdessen wird der spezifische Kernel aus der Devicespezifikation und dem generischen Kernel heraus generiert. Dieser Ansatz reduziert das Retargeting auf das Schreiben von n Devicespezifikationen für n Devices. Der generische Kernel und der generische Compiler müssen nur einmal geschrieben werden. Dies reduziert nicht nur den Arbeitsaufwand, sondern auch mögliche Fehlerquellen.

Die in dieser Arbeit beschriebene Lösung orientiert sich am letzten Ansatz. Der Vorteil dieses Ansatzes besteht darin, dass die größtmögliche Anzahl von Komponenten wiederverwendet werden kann und somit der Arbeitsaufwand auf ein Minimum reduziert wird. Die *roboXL* Kontrollprogramme und die Devicespezifikation werden jedoch nicht direkt in *Java* Sourcecode übersetzt. Dies wäre theoretisch zwar denkbar, aber vielmehr möchte man eine klare Trennung der Codefragmente des generischen Kernels, der devicespezifischen Erweiterungen und des eigentlichen Programmcodes (Seperation of concerns), um eine leichte Wartbarkeit zu garantieren. Um dies zu erreichen, werden Aspekte definiert. *Aspektorientierte Programmierung (AOP)* (s. 1.4.4) erlaubt es uns, verschiedene Codefragmente für verschiedene Eigenschaften (concerns, aspects) des Systems seperat zu definieren und anschliessend zu einem Gesamtsystem zusammenzusetzen ("weben"). Die Aspekte ermöglichen es, einem existierenden System nach Belieben Code hinzuzufügen, um es um die dort definierte Funktionalität zu erweitern [24]. Auf diese Weise hat man den generischen Kernel als Ausgangspunkt, dem dann die devicespezifischen Erweiterungen und der eigentliche Programmcode mittels Aspekten hinzugefügt werden.

Die Frage die sich anschliesst ist, mit welcher Technologie die *roboXL* Kontrollprogramme und die Devicespezifikation verarbeitet werden können? Mit anderen Worten ausgedrückt, wie lassen sich der generische ***roboXL* Compiler** für die *roboXL* Kontrollprogramme und der **Retargeting Generator**, das ist die Komponente, die die Devicespezifikation verarbeitet, implementieren? Ein gängiger Standard zur Transformation von *XML*-Dokumenten ist die *XML Stylesheet Language for Transformations*, kurz *XSLT* (s. 1.4.3). *XSLT* Stylesheets ermöglichen es auf eine gleichermaßen simple wie elegante Art und Weise, Compiler zu implementieren. Der Einsatz von *XSLT* als Tool zur Programmgenerierung wird ausführlich in [9] diskutiert. Wir benutzen demnach *XSLT* Stylesheets um die Aspekte

zu generieren. Solange keine komplexen Programmanalysen notwendig sind, erreicht man mit dem vorgeschlagenen Lösungsansatz eine leichtere Wartbarkeit und Erweiterbarkeit verglichen mit einer Implementierung, die sich an den klassischen Compilertechniken in *Java* orientiert [3]. Interessant im Kontext *XSLT* und *AOP* ist SmartTools [4]. Hier werden *XSLT* und *AOP* zur Generierung von Programmierumgebungen verwendet. *XSLT* und *AOP* werden hier jedoch nicht direkt kombiniert, d.h. die Aspekte werden nicht mit *XSLT* erzeugt.
Als Referenzdevice wird der Lego Mindstorms RCX [16] verwendet. Empfehlenswert zum Thema Lego Mindstorms RCX ist das Buch *Jin Sato's Lego Mindstorms* [20].

Die weiteren Kapitel gliedern sich wie folgt: Kapitel 2 beschreibt, aufbauend auf dem oben dargelegten Lösungsansatz, die Architektur des Systems zur Verarbeitung von *roboXL* Kontroll- und Monitoringprogrammen und der Devicespezifikationen sowie die Beziehungen der einzelnen Komponenten im Detail. In Kapitel 3 erfolgt eine Bewertung und ein Rückblick auf den Verlauf der Arbeit. In Kapitel 4 ist eine vollständige Beschreibung der definierten *XML*-Sprachen zu finden.

Kapitel 2

Architektur und Komponenten

Dieses Kapitel gibt eine detaillierte Beschreibung der Architektur des Systems zur Verarbeitung von *roboXL*. Es wird Bezug auf die einzelnen Komponenten genommen und deren Zusammenspiel erläutert.
roboXL dient in erster Linie dazu, Experimente durchzuführen. Das gesamte System bildet somit eine Entwicklungsumgebung für Experimente. In diesem Zusammenhang wird der Begriff "Experiment Operating System", kurz **ExOS** verwendet, der auch bei der Namensgebung vieler Systemkomponenten Anwendung findet.
Grundsätzlich werden zwei Anwendungsfälle unterschieden: Die Verarbeitung von *roboXL* Kontrollprogrammen und die Verarbeitung von *roboXL* Monitoringprogrammen. Die folgenden Abschnitte zeigen die Architektur für diese beiden Fälle und benutzen jeweils ein konkretes Beispiel zur Veranschaulichung des Verarbeitungsablaufes.

2.1 Verarbeitung eines *roboXL* Kontrollprogramms

Abbildung 2.1 zeigt die Komponenten, die in die Verarbeitung eines *roboXL* Kontrollprogramms involviert sind. Dazu sind zwei Verarbeitungsschritte notwendig. Einmal das sogenannte Retargeting, d.h. die Anpassung an das aktuelle Device, und schliesslich das Compilieren des eigentlichen Kontrollprogramms. Die gestrichelten Pfeile in Abbildung 2.1 stellen dabei Abhängigkeiten zwischen Komponenten dar, z.B. dass eine Komponente die andere einbindet. Die durchgezogenen Pfeile representieren einzelne Verarbeitungsschritte.
Ausgangspunkt ist ein generisches Kernel *Java* Programm (**ExOSKernel.java**). Diese Klasse besitzt ausser Variablen und Methodendeklarationen noch keinerlei Funktionalität. Sie bildet das Grundgerüst für alle *Java* Programme, die vom *roboXL* Compiler aus *roboXL* Kontrollprogrammen erzeugt werden. **ExOSKernel.java** entspricht somit dem in Kapitel 1.5 beschriebenen generischen Kernel und beinhaltet die Teile, die in jedem *Java* Programm enthalten sind, das aus einem *roboXL* Kontrollprogramm generiert wird. Der generische Kernel ist eine Komponente, die nur einmal erstellt werden muss, aber immer wieder verwendet wird.

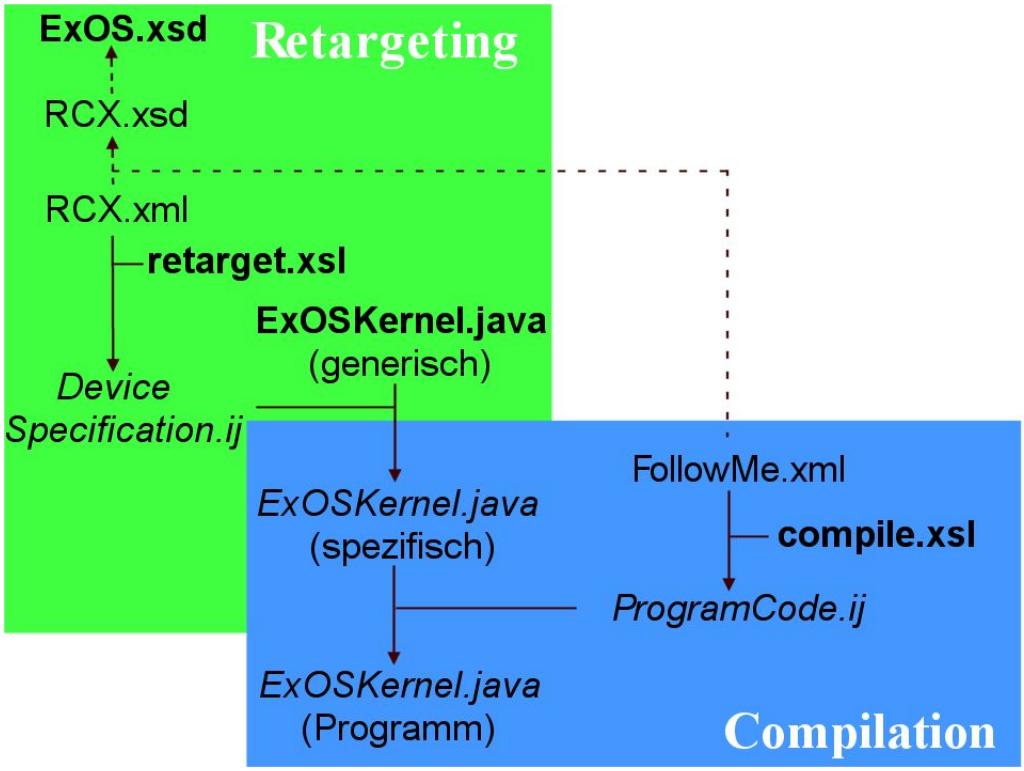


Abbildung 2.1: Verarbeitung eines *roboXL* Kontrollprogramms

Um eine erste Vorstellung von einem in *roboXL* geschriebenen Kontrollprogramm zu bekommen, folgt nun ein erstes Beispiel. Das Kontrollprogramm ist in der Datei **FollowMe.xml** definiert (s. Abbildung 2.2). Es soll einen Lego Mindstorms RCX Roboter so steuern, dass er einer schwarzen Linie folgt. Der Roboter hat einen Lichtsensor und zwei Motoren. Bewegen sich beide Motoren in dieselbe Richtung, so bewegt sich auch der Roboter in diese Richtung. Bewegen sich die Motoren entgegengesetzt, so dreht sich der Roboter.

Das Programm testet in einer Endlosschleife (`counter` hat den Wert 0), ob der Wert des Lichtsensors (`<KIND>` hat den Wert `light`) kleiner als 50% ist. Ist dies der Fall, so befindet sich der Roboter weiterhin auf der schwarzen Linie und die Fahrtrichtung wird beibehalten. Andernfalls muss der Roboter gedreht werden. Es genügt dabei den Sensor alle 500 msec zu testen.
Die wichtigsten Elemente in *roboXL* Kontrollprogrammen sind `<SENSOR>` und `<MOTOR>`. `<SENSOR>` dient der Abfrage eines Sensors, `<MOTOR>` der Kontrolle eines Motors.

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="compile.xsl"?>

<EXOS xmlns ="ExOSNamespace"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="ExOSNamespace
                          RCX.xsd">
  <ROBOXL>
    <PROGRAM>
      <LOOP counter="0">

        <SWITCH>
          <EXPRESSION op="less" type="bool">
            <SENSOR id="1" type="int">
              <KIND>light</KIND>
              <MODE>pct</MODE>
            </SENSOR>
            <VALUE>
              <INT>50</INT>
            </VALUE>
          </EXPRESSION>
          <CASE>
            <VALUE><BOOL>true</BOOL></VALUE>
            <MOTOR id="1">
              <POWER>1</POWER>
              <FWD/>
            </MOTOR>
            <MOTOR id="2">
              <POWER>1</POWER>
              <FWD/>
            </MOTOR>
          </CASE>
          <CASE>
            <VALUE><BOOL>false</BOOL></VALUE>
            <MOTOR id="1">
              <POWER>1</POWER>
              <FWD/>
            </MOTOR>
            <MOTOR id="2">
              <POWER>1</POWER>
              <BWD/>
            </MOTOR>
          </CASE>
        </SWITCH>
        <SLEEP unit="msec">500</SLEEP>
      </LOOP>
    </PROGRAM>
  </ROBOXL>
</EXOS>
```

Abbildung 2.2: *roboXL* Kontrollprogramm “FollowMe.xml”

2.1.1 Retargeting

Alle Elemente von *roboXL*, die nicht devicespezifisch sind, sind in *XML Schema* **ExOS.xsd** definiert. **ExOS.xsd** definiert die generische Version von *roboXL*. Devices unterscheiden sich in erster Linie darin, welche Sensoren und Motoren sie anbieten und wie diese angesteuert werden. Demnach gibt es Elemente, gerade diejenigen für Sensoren und Motoren, die in **ExOS.xsd** noch nicht vollständig definiert werden können. Diese Elemente besitzen zwar Teile, die für jedes Device gleich sind, sozusagen einen kleinsten gemeinsamen Nenner, aber auch Teile, die für jedes Device unterschiedlich sind. Um dem gerecht zu werden, wird für jedes Device ein *XML Schema* definiert, das **ExOS.xsd** vollständig importiert und diese Elemente durch die in *XML Schema* verfügbaren Vererbungstechniken devicespezifisch definiert. Für den RCX ist dies das *XML Schema* **RCX.xsd**. Allen *roboXL* Instanzdokumenten liegt deshalb das devicespezifische *XML Schema* **RCX.xsd** zugrunde, da **ExOS.xsd** aus oben genannten Gründen kein vollständiges *XML Schema* ist und deshalb nicht zum Validieren von Dokumenten verwendet werden kann. Die Erstellung des devicespezifischen *XML Schema* ist der erste Teil des Retargeting. Der zweite Teil ist die eigentliche Devicespezifikation (s. Abbildung 2.3 auf Seite 26), die mit der Devicespezifikationssprache erstellt wird.

- **Retargeting Teil1: *XML Schema* RCX.xsd**

  – Definition von `<MOTOR>` in *XML Schema* **RCX.xsd**
  ExOS.xsd definiert lediglich einen complexType Motor, der bei der devicespezifischen Motordefinition erweitert wird.

```
<xsd:complexType name="Motor">
   <xsd:attribute name="id" type="MotorId" use="required"/>
</xsd:complexType>
```

  Dieser complexType legt fest, dass jeder Motor über eine id angesprochen wird. Diese Eigenschaft gilt für Motoren eines jeden Devices und ist deshalb in **ExOS.xsd** definiert. Der Typ dieser id kann jedoch von Device zu Device unterschiedlich sein. Deshalb kann der Typ des Attributs id durch Definition des Typs MotorId im devicespezifischen *XML Schema* flexibel an die Anforderungen des jeweiligen Devices angepasst werden. Für den RCX ist MotorId wie folgt definiert:

```
<xsd:simpleType name="MotorId">
  <xsd:restriction base="xsd:integer">
    <xsd:minInclusive value="1"/>
    <xsd:maxInclusive value="3"/>
  </xsd:restriction>
</xsd:simpleType>
```

Der RCX stellt drei Motoren zur Verfügung. Als zulässige ids werden deshalb die Integer 1,2 und 3 definiert. *XML Schema* stellt mit restriction eine Vererbungstechnik zur Verfügung, die es erlaubt, nur einen eingeschränkten Wertebereich eines schon existierenden Datentyps (hier integer) zu vererben. Jeder Motor kann über die id des <MOTOR> Elements eindeutig identifiziert werden. Als nächstes wird das Element <MOTOR> durch Vererbung des complexType Motor definiert:

```
<xsd:element name="MOTOR">
  <xsd:complexType>
    <xsd:complexContent>
      <xsd:extension base="Motor">
        <xsd:choice minOccurs="1" maxOccurs="unbounded">
                <xsd:element name="FWD"/>
                <xsd:element name="BWD"/>
                <xsd:element name="STOP"/>
                <xsd:element name="FLT"/>
                <xsd:element ref="POWER"/>
        </xsd:choice>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
</xsd:element>
```

Durch den complexType Motor erbt das Element <MOTOR> das Attribut id. Desweiteren wird durch Erweiterung (<xsd:extension>) des complexType Motor jede Option, die bei der Konfiguration eines RCX Motors angegeben werden kann, als Kindelement von <MOTOR> definiert. Erweitert man einen complexType, so benötigt man in *XML Schema* <xsd:complexContent>. Die Optionen bei der Konfiguration eines Motors sind: Bewegung vorwärts (FWD), Bewegung rückwärts (BWD), Stoppen des Motors (STOP), Motorbewegung langsam auslaufen lassen (FLT) und schliesslich das Setzen der Motorenstärke (POWER). Ein RCX Motor hat acht verschiedene Stärken:

```
<xsd:element name="POWER">
  <xsd:simpleType>
    <xsd:restriction base="xsd:integer">
        <xsd:minInclusive value="0"/>
        <xsd:maxInclusive value="7"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>
```

– Definition von <SENSOR> in *XML Schema* **RCX.xsd**
ExOS.xsd definiert genau wie beim Element <MOTOR> nur einen complexType:

```
<xsd:complexType name="Sensor">
   <xsd:attribute name="id" type="SensorId"
                  use="required"/>
   <xsd:attribute name="type" use="required">
      <xsd:simpleType>
         <xsd:restriction base="xsd:string">
            <xsd:enumeration value="int"/>
            <xsd:enumeration value="float"/>
            <xsd:enumeration value="bool"/>
         </xsd:restriction>
      </xsd:simpleType>
   </xsd:attribute>
</xsd:complexType>
```

Durch den complexType Sensor ist festgelegt, dass jeder Sensor über eine id angesprochen wird. Genau wie bei Motoren gilt diese Eigenschaft für Sensoren eines jeden Devices und ist deshalb in **ExOS.xsd** definiert. Auch hier kann der Typ dieser id wieder von Device zu Device unterschiedlich sein. Aus diesem Grund wird der Typ SensorId devicespezifisch definiert. Desweiteren muss für jeden Sensor sein type angegeben werden, d.h. der Typ des Wertes, den der Sensor zurückliefert.

```
<xsd:simpleType name="SensorId">
  <xsd:restriction base="xsd:integer">
     <xsd:minInclusive value="1"/>
     <xsd:maxInclusive value="3"/>
  </xsd:restriction>
</xsd:simpleType>
```

Der RCX stellt neben seinen drei Motoren genauso drei Sensoren zur Verfügung.

Jeder Sensor kann über die id des <SENSOR> Elements eindeutig identifiziert werden. Das Element <SENSOR> wird schliesslich durch Erweiterung des complexType Sensor definiert:

```
<xsd:element name="SENSOR">
  <xsd:complexType>
    <xsd:complexContent>
      <xsd:extension base="Sensor">
        <xsd:sequence>
          <xsd:element ref="KIND"/>
          <xsd:element ref="MODE"/>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
</xsd:element>
```

Durch den complexType Sensor erbt das Element <SENSOR> die Attribute id und type. Um den Wert eines RCX Sensors abzufragen, muss dieser zuerst konfiguriert werden. Dazu müssen zwei Eigenschaften des Sensors angegeben werden, die durch Erweiterung des complexType Sensor als Kindelemente von <SENSOR> definiert werden. Zum einen ist die Art des Sensors anzugeben:

```
<xsd:element name="KIND">
  <xsd:simpleType>
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="raw"/>
      <xsd:enumeration value="touch"/>
      <xsd:enumeration value="temp"/>
      <xsd:enumeration value="light"/>
      <xsd:enumeration value="rot"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>
```

Der RCX hat Sensoren für Rohdaten bzw. absolute Werte (raw), Berührung (touch), Temperatur (temp), Licht (light) und Drehung (rot).

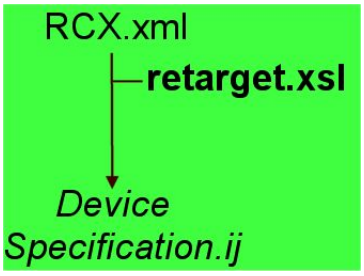
Zum anderen muss der Modus des Sensors angegeben werden:

```
<xsd:element name="MODE">
  <xsd:simpleType>
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="raw"/>
      <xsd:enumeration value="bool"/>
      <xsd:enumeration value="edge"/>
      <xsd:enumeration value="pulse"/>
      <xsd:enumeration value="pct"/>
      <xsd:enumeration value="degc"/>
      <xsd:enumeration value="degf"/>
      <xsd:enumeration value="angle"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>
```

Dabei wird unterschieden zwischen

- raw - Rohdatenmessung, Werte im Bereich 0-1023
- bool - misst, ob Sensor gedrückt ist oder nicht (nur Berührungssensor)
- edge - zählt, wieoft sich der Zustand eines Sensors ändert (nur Berührungssensor)
- pulse - zählt, wieoft ein Sensor gedrückt wurde (nur Berührungssensor)
- pct - misst Helligkeit in Prozent (nur Lichtsensor)
- degc - misst Temperatur in ° Celsius (nur Temperatursensor)
- degf - misst Temperatur in Fahrenheit (nur Temperatursensor)
- angle - Winkelmessung (nur Drehungssensor)

- **Retargeting Teil2: Devicespezifikation**



Für jedes Device muss eine *XML*-basierte Devicespezifikation erstellt werden. In der Devicespezifikation wird die Implementierung der spezifischen Teile des Kernels

angegeben. Von zentraler Bedeutung sind dabei Methoden, die spezifizieren, wie Motoren und Sensoren devicespezifisch angesprochen werden.
Zum Erstellen der Devicespezifikation werden die Elemente der Devicespezifikationssprache verwendet, die in **RCX.xsd** definiert sind und für alle Devices gleich sind (Genauergesagt ist die Devicespezifikationssprache bereits vollständig in *XML Schema* **ExOS.xsd** definiert, das von **RCX.xsd** importiert wird). Die Devicespezifikation für den RCX steht in der Datei **RCX.xml** (s. Abbildung 2.3)

```
<SPECIFICATION>
  <IMPORTS>josx.platform.rcx.Motor,josx.platform.rcx.Sensor,...</IMPORTS>
  <VARIABLES>
    private String strFWD="FWD"; private String strPOWER="POWER";
    private String strlight= "light"; private String strpct="pct";
    private Boolean objReturnBoolean = new Boolean(false);
    private Integer objReturnInteger = new Integer(0);
    ...
  </VARIABLES>
  <INIT>
    arrayMotors = new Motor[3];
    arrayMotors[0]= Motor.A; arrayMotors[1]= Motor.B; arrayMotors[2]= Motor.C;
    arraySensors = Sensor.SENSORS;
  </INIT>

  <CONTROLMOTOR>
    <RETURN>void</RETURN>
    <ARGS>String motor, String op, String value</ARGS>
    <CODE>if (op.equals(strFWD)) {
            ((Motor)arrayMotors[stringToInt(motor) - 1]).forward();
            return;}
        if (op.equals(strPOWER)) {
            ((Motor)arrayMotors[stringToInt(motor) - 1]).setPower(stringToInt(value));
            return;}
        ...
     </CODE>
  </CONTROLMOTOR>
```

```
<READSENSOR>
  <ARGS>String sensor, String type, String kind, String mode</ARGS>
  <CODE>int intKind = 0; int intMode = 0;
  if (kind.equals(strlight)) intKind = SensorConstants.SENSOR_TYPE_LIGHT;
  ...
  if (mode.equals(strpct))   intMode = SensorConstants.SENSOR_MODE_PCT;
  ...
  if (type.equals(strint)) {
  objReturnInteger.setValue(readSensorValue(stringToInt(sensor)-1,
                                            intKind,intMode));
  return objReturnInteger;}
  else if (type.equals(strbool)) {
  objReturnBoolean.setValue(readSensorValue(stringToInt(sensor)-1,
                                            intKind,intMode));
  return objReturnBoolean;}
  ...
  </CODE>
</READSENSOR>

<METHODS>
  <METHOD name="stringToInt">
    <MODIFIER>public</MODIFIER> <RETURN>int</RETURN>
    <ARGS>String s</ARGS> <CODE>...</CODE>
  </METHOD>

  <METHOD name="readSensorValue">
    <MODIFIER>public synchronized</MODIFIER>
    <RETURN>int</RETURN>
    <ARGS>int index,int kind,int mode</ARGS>
    <CODE>((Sensor) arraySensors[index]).setTypeAndMode(kind, mode);
          ((Sensor) arraySensors[index]).activate();
          return ((Sensor) arraySensors[index]).readValue();
    </CODE>
  </METHOD>
</METHODS>
...
```

Abbildung 2.3: Devicespezifikation des RCX

Die Devicespezifikation besteht aus Importanweisungen für *Java* Klassen, die für die Implementierung benötigt werden (`<IMPORTS>`), Variablendeklarationen (`<VARIABLES>`), Initialisierungscode (`<INIT>`), der Spezifikation der Methoden *controlMotor()* (`<CONTROLMOTOR>`) und *readSensor()* (`<READSENSOR>`) und gegebenenfalls aus weiteren Hilfsmethoden (`<METHODS>`). Die Methode *controlMotor()* implementiert das devicespezifische Ansteuern eines Motors, *readSensor()* entsprechend das Ansteuern eines Sensors. Für *controlMotor()* müssen Rückgabewert, Argumente und Implementierung angegeben werden. Die Implementierung von *controlMotor()* für den RCX verwendet die Hilfsmethode *stringToInt()* (`<METHODS>`), da die *leJOS Java API* keine solche Funktion bereitstellt. Bei der Definition einer Hilfsmethode müssen die Modifier (`<MODIFIER>`), der Rückgabewert (`<RETURN>`), die Argumente (`<ARGS>`) und der Rumpf der Methode (`<CODE>`) angegeben werden. Der Rückgabewert von *readSensor()* ist fest vorgegeben und immer vom Typ *Object*, um zu ermöglichen, dass ein Sensor jeden beliebigen Typ zurückgeben kann (s. Abbildung 2.4). Für *readSensor()* müssen demnach nur die Argumente und die Implementierung angegeben werden.
Zum Steuern von Motoren und Sensoren stellt die *leJOS Java API* die Klassen *Motor* und *Sensor* zur Verfügung. Die Klasse *Motor* besitzt unter anderem die Methoden *forward()* und *setPower(int)*. Mit der Methode *forward()* startet man einen Motor in Vorwärtsbewegung. Mit der Methode *setPower(int)* lässt sich die Motorenstärke einstellen. Die Konfiguration eines Sensors erfolgt über die Methode *setTypeAndMode(int,int)* der Klasse *Sensor*, der als Argumente vordefinierte Konstanten (Klasse *SensorConstants*) übergeben werden. Der Sensor wird anschliessend durch die Methode *activate()* aktiviert und mit der Methode *readValue()* kann der aktuelle Wert des Sensors abgefragt werden. Die Abfrage eines Sensors ist dabei in einer eigenen Hilfsmethode *readSensorValue()* (`<METHODS>`) gekapselt.
Die *Java Virtual Machine* des RCX unterstützt leider keine Garbage Collection. Aus genau diesem Grund werden alle benötigten Strings ebenso wie mögliche Rückgabewerte der Methode *readSensor()* als globale Variablen deklariert, da eine Reallokation zum Speicherüberlauf auf dem RCX führt.
Die Transformation einer Devicespezifikation erfolgt durch ein *XSLT* Stylesheet, den sogenannten **Retargeting Generator**. Dieser ist in der Datei **retarget.xsl** definiert (s. Abbildung 2.4).

Die Devicespezifikation ist im Grunde genommen ein mit *XML* verfasster Aspekt. Der *XSLT* Prozessor ersetzt die *XSLT* Elemente durch den in der Devicespezifikation definierten *Java* Code und erzeugt das *Inject/J* Skript bzw. den Aspekt **DeviceSpecification.ij** (s. Abbildung 2.5).

```
<xsl:output method="text"/>

<xsl:template match="/">
script DeviceSpecification {
  in class 'ExOSKernel' do {

add to imports ${
  <xsl:value-of select="/exos:EXOS/exos:SPECIFICATION/exos:IMPORTS"/>
}$;

in method 'ExOSKernel()' do {
     before ${
     <xsl:value-of select="/exos:EXOS/exos:SPECIFICATION/exos:INIT"/>
     }$;
}

add to members ${
 <xsl:value-of select="/exos:EXOS/exos:SPECIFICATION/exos:VARIABLES"/>
}$;
...

add to members ${
public synchronized
<xsl:value-of select="/exos:EXOS/exos:SPECIFICATION/exos:CONTROLMOTOR/exos:RETURN"/>
controlMotor
(<xsl:value-of select="/exos:EXOS/exos:SPECIFICATION/exos:CONTROLMOTOR/exos:ARGS"/>)
{
  <xsl:value-of select="/exos:EXOS/exos:SPECIFICATION/exos:CONTROLMOTOR/exos:CODE"/>
}
}$;

add to members ${
public synchronized Object readSensor
(<xsl:value-of select="/exos:EXOS/exos:SPECIFICATION/exos:READSENSOR/exos:ARGS"/>)
{
  <xsl:value-of select="/exos:EXOS/exos:SPECIFICATION/exos:READSENSOR/exos:CODE"/>
}
}$;
...
```

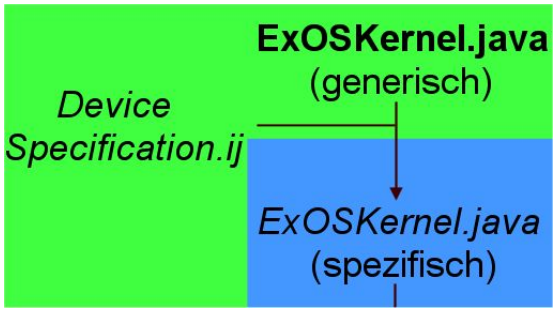
Abbildung 2.4: Ausschnitt von retarget.xsl

```
script DeviceSpecification {
  in class 'ExOSKernel' do {

    add to imports ${josx.platform.rcx.Motor,josx.platform.rcx.Sensor,...}$;

    in method 'ExOSKernel()' do {
          before ${
          arrayMotors = new Motor[3];
          arrayMotors[0]= Motor.A;
          arrayMotors[1]= Motor.B;
          arrayMotors[2]= Motor.C;

          arraySensors = Sensor.SENSORS;
          }$;
    }

    add to members ${
      private String strFWD="FWD"; private String strPOWER="POWER";
      private String strlight= "light"; private String strpct="pct";
      private Boolean objReturnBoolean = new Boolean(false);
      private Integer objReturnInteger = new Integer(0);
      ...
    }$;

    ...
```

```
add to members ${
  public synchronized void controlMotor
  (String motor, String op, String value) {
  if (op.equals(strFWD)) {
      ((Motor)arrayMotors[stringToInt(motor) - 1]).forward();
      return;
  }
  if (op.equals(strPOWER)) {
      ((Motor)arrayMotors[stringToInt(motor) - 1]).setPower(stringToInt(value));
      return;
  }
  ...
}$;

add to members ${
  public synchronized Object readSensor
  (String sensor, String type, String kind, String mode) {
  int intKind = 0; int intMode = 0;
  if (kind.equals(strlight)) intKind = SensorConstants.SENSOR_TYPE_LIGHT;
  ...
  if (mode.equals(strpct))   intMode = SensorConstants.SENSOR_MODE_PCT;
  ...
  ((Sensor) arraySensors[stringToInt(sensor) - 1]).setTypeAndMode(intKind,
                                                                  intMode);
  ((Sensor) arraySensors[stringToInt(sensor) - 1]).activate();
  if (type.equals(strint)) {
  objReturnInteger.setValue(
        ((Sensor) arraySensors[stringToInt(sensor) - 1]).readValue());
  return objReturnInteger;
  }
  else if (type.equals(strbool)) {
  objReturnBoolean.setValue(
        ((Sensor) arraySensors[stringToInt(sensor) - 1]).readValue());
  return objReturnBoolean;
  }
  ...
}$;
...
```

Abbildung 2.5: Ausschnitt von DeviceSpecification.ij

Durch den Aspekt **DeviceSpecification.ij** wird der generische Kernel **ExOSKernel.java** erweitert (`in class 'ExOSKernel' do`). Wie man sehen kann, werden alle Variablen als Membervariablen deklariert (`add to members`). Der Initialiserungscode wird später in den Rumpf des Konstruktors eingewebt (`in method 'ExOSKernel()' do`). Dazu werden die Methoden *readSensor()* und *controlMotor()* in die Klasse **ExOSKernel.java** aufgenommen (`add to members`).

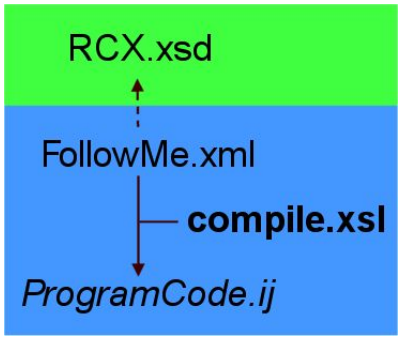


Der *Inject/J* Compiler webt den Aspekt **DeviceSpecification.ij** in das generische Kernel *Java* Programm, das dadurch zu einem spezifischen Kernel wird. Der *roboXL* Compiler erzeugt später aus *roboXL* Kontrollprogrammen *Java* Code, der die im spezifischen Kernel implementierte Funktionalität nutzt.
Neben dem bislang Beschriebenen umfasst die Devicespezifikation jedoch noch einen weiteren Punkt. Wir werden später sehen, wie *roboXL* Monitoringprogramme verarbeitet werden. Bei der Verarbeitung von Monitoringprogrammen erfolgt ein Datenaustausch mit dem Kontrollprogramm, das auf dem Device abläuft. Dazu müssen die Kontrollprogramme um ein Kommunikationsprotokoll erweitert werden. Die Implementierung dieses Protokolls ist ebenfalls Bestandteil der Devicespezifikation. Weitere Informationen hierzu sind in Kapitel 2.2 auf Seite 42 zu finden.

2.1.2 Compilation

Die Verarbeitung eines *roboXL* Kontrollprogramms obliegt dem ***roboXL* Compiler**. Dieser Compiler ist genau wie der **Retargeting Generator** ein *XSLT* Stylesheet (**compile.xsl**), das ein *Inject/J* Skript (**ProgramCode.ij**) erzeugt. Der Compiler ist generisch, d.h. für alle Devices gleich. Dies wird dadurch erreicht, dass der *Java* Code, den der Compiler letztendlich aus einem *roboXL* Kontrollprogramm erzeugt, auf die Methoden des durch das **Retargeting** erzeugten spezifischen Kernels zugreift. Das **Retargeting** wird deshalb immer vor dem Compilieren ausgeführt.

Die Transformation des *roboXL* Kontrollprogramms **FollowMe.xml** (s. Abbildung 2.2 auf Seite 19) in ein *Inject/J* Skript durch den *roboXL* Compiler wird nun im Einzelnen erkärt. Dabei soll die grundlegende Arbeitsweise des *roboXL* Compilers veranschaulicht



werden. Dabei kann selbstverständlich nicht auf alle Implementierungsdetails eingegangen werden. Bestimmte Teile des *roboXL* Compilers werden deshalb vereinfacht dargestellt.

Der *roboXL* Compiler löst die verschachtelte *XML*-Struktur durch den rekursiven Aufruf einer *XSLT* Template Funktion (`transform`) auf:

```
<xsl:template match="/exos:EXOS/exos:ROBOXL/exos:PROGRAM">
    script ProgramCode
    {
      in class 'ExOSKernel' do {
        foreach method 'main(*)' do {
          before ${
            ExOSKernel kernel = new ExOSKernel();

            <xsl:for-each select="*">
              <xsl:call-template name="transform">
                <xsl:with-param name="elem" select="name(.)"/>
              </xsl:call-template>
            </xsl:for-each>
          }$;
        }
      }
    }
```

Das Kontrollprogramm startet mit einer Endlosschleife.

```
<LOOP counter="0">
```

Durch Angabe des *XPath* /exos:EXOS/exos:ROBOXL/exos:PROGRAM wird zum `<PROGRAM>` Element des *roboXL* Kontrollprogramms verzweigt. Mittels `<xsl:foreach>` wird die Template Funktion `transform` für jedes Kindelement von `<PROGRAM>` aufgerufen. Als Parameter wird dem `transform` Template der Name (`name(.)`) des Kindelements

übergeben. Auf diese Weise wird das <LOOP> Element erfasst und transform verzweigt in den dazugehörigen Fall:

```
<xsl:template name="transform">
  <xsl:param name="elem"/>

  <xsl:choose>
    ...
    <xsl:when test="$elem='LOOP'">
      <xsl:choose>
        <xsl:when test="@counter=0">
          while(true)
          {
        </xsl:when>
        <xsl:otherwise>
          for(int i=0;i&lt;<xsl:value-of select="@counter"/>;i++)
          {
        </xsl:otherwise>
      </xsl:choose>

      <xsl:for-each select="*">
        <xsl:call-template name="transform">
          <xsl:with-param name="elem" select="name(.)"/>
        </xsl:call-template>
      </xsl:for-each>
      }
    </xsl:when>
    ...
  </xsl:choose>
</xsl:template>
```

Das Template überprüft den Wert des Attributs counter. Falls der Wert 0 ist, wird mit *while* eine Endlosschleife in *Java* erzeugt, andernfalls eine *for*-Schleife. Rekursiv wird anschliessend wiederum die Template Funktion transform für jedes Kindelement von <LOOP> aufgerufen. Innerhalb der Schleife tritt nun zuerst eine <SWITCH> Anweisung auf.

```
<LOOP counter="0">

<SWITCH>
  <EXPRESSION op="less" type="bool">
    ...
  </EXPRESSION>
```

```
  <CASE>
    <VALUE><BOOL>true</BOOL></VALUE>
    ...
  </CASE>
  <CASE>
    <VALUE><BOOL>false</BOOL></VALUE>
    ...
  </CASE>
 </SWITCH>
```

Der entsprechende Fall des `transform` Templates sieht wie folgt aus:

```
<xsl:when test="$elem='SWITCH'">
  <xsl:for-each select="*">
    <xsl:choose>
      <xsl:when test="name(.)='EXPRESSION'  ...  ">

        ...
            kernel.<xsl:value-of select="@type"/>Stack.push(
        ...

        <xsl:call-template name="transform">
          <xsl:with-param name="elem" select="name(.)"/>
        </xsl:call-template>
        );
      </xsl:when>

      <xsl:when test="name(.)='CASE'">
        <xsl:for-each select="*">
          <xsl:choose>
            <xsl:when test="name(.)='VALUE'">
              if(
              <xsl:choose>
                <xsl:when test="../../exos:EXPRESSION/@type != ''">
    kernel.<xsl:value-of select="../../exos:EXPRESSION/@type"/>Stack.top()
                </xsl:when>

                ...

              </xsl:choose>
              == <xsl:value-of select="*"/>) {
            </xsl:when>
```

```
            <xsl:otherwise>
              <xsl:call-template name="transform">
                <xsl:with-param name="elem" select="name(.)"/>
              </xsl:call-template>
            </xsl:otherwise>
          </xsl:choose>
        </xsl:for-each>
        }
      </xsl:when>
    </xsl:choose>
  </xsl:for-each>
  <xsl:for-each select="*">
    <xsl:if test="name(.)='EXPRESSION' ... ">
      ...
          kernel.<xsl:value-of select="@type"/>Stack.pop();
      ...
    </xsl:if>
  </xsl:for-each>
</xsl:when>
```

In unserem Fall ist das Argument der `<SWITCH>` Anweisung eine `<EXPRESSION>` und der Ausschnitt des *roboXL* Compilers zeigt nur die dafür relevanten Teile. Der Wert des Arguments wird einmal gespeichert und dann bei Eintritt in die Fallunterscheidungen (`<CASE>`) wieder abgefragt. Da `<SWITCH>` Anweisungen verschachtelt sein können, werden hierzu Stacks verwendet. Für jeden erlaubten Datentyp (int, boolean, float) gibt es einen eigenen Stack. Der Wert des Arguments der `<SWITCH>` Anweisung wird auf dem Stack mit `push()` abgelegt und für jedes `<CASE>` wieder mit `top()` abgefragt. Vor Verlassen der `<SWITCH>` Anweisung wird das Argument vom Stack entfernt (`pop()`). Die Auswahl des Stacks erfolgt über das `type` Attribut des `<EXPRESSION>` Elements.
`<EXPRESSION>` und `<CASE>` werden wie oben gesehen wieder durch den rekursiven Aufruf des `transform` Templates verarbeitet. Werfen wir zuerst einen Blick darauf, wie `<EXPRESSION>` verarbeitet wird.

```
<xsl:when test="$elem='EXPRESSION'">
  <xsl:for-each select="*">
    <xsl:choose>
      <!-- left side of the expression -->
      <xsl:when test="position()=1">
        (
        <xsl:call-template name="transform">
          <xsl:with-param name="elem" select="name(.)"/>
        </xsl:call-template>
```

```
        <!-- expression operator -->
        <xsl:call-template name="getOperator">
          <xsl:with-param name="op" select="../@op"/>
        </xsl:call-template>
      </xsl:when>

      <!-- right side of the expression -->
      <xsl:when test="position()=2">
        <xsl:call-template name="transform">
          <xsl:with-param name="elem" select="name(.)"/>
        </xsl:call-template>
        )
      </xsl:when>
    </xsl:choose>
  </xsl:for-each>
</xsl:when>
```

Zur Erinnerung, das `<EXPRESSION>` Element sieht wie folgt aus:

```
<EXPRESSION op="less" type="bool">
  <SENSOR id="1" type="int">
    <KIND>light</KIND>
    <MODE>pct</MODE>
  </SENSOR>
  <VALUE>
    <INT>50</INT>
  </VALUE>
</EXPRESSION>
```

Die linke Seite des Ausdrucks ist demnach ein Sensorwert, die rechte Seite ist ein einfacher Wert. Es soll getestet werden, ob die von einem Lichtsensor (der Inhalt des `<KIND>` Elements ist `light`) in Prozent (der Inhalt des `<MODE>` Elements ist `pct`) gemessene Helligkeit kleiner (`op` hat den Wert `less`) als 50% ist. Die Transformation von `<SENSOR>` erfolgt durch erneuten Aufruf des `transform` Templates:

```
<xsl:when test="$elem='SENSOR'">
  ((
  <xsl:choose>
    <xsl:when test="@type='bool'">
      Boolean
    </xsl:when>
```

```
      <xsl:when test="@type='int'">
        Integer
      </xsl:when>
      <xsl:when test="@type='float'">
        Float
      </xsl:when>
    </xsl:choose>
    )
    kernel.readSensor(
    kernel.strArg<xsl:value-of select="@id"/>,
    kernel.strArg<xsl:value-of select="@type"/>
    <xsl:for-each select="*">
      ,
      <xsl:choose>
        <xsl:when test=".=''">
          kernel.strArgEmpty
        </xsl:when>
        <xsl:otherwise>
          kernel.strArg<xsl:value-of select="."/>
        </xsl:otherwise>
      </xsl:choose>
    </xsl:for-each> )).getValue()
  </xsl:when>
```

An dieser Stelle kommt die Methode *readSensor()* zum Einsatz, die in der Devicespezifikation definiert wurde (s. Seite 26) und dadurch Bestandteil des spezifischen Kernels ist. Der Methode *readSensor()* werden die Werte der Attribute id und type des Elements <SENSOR> sowie der Inhalt eines jeden Kindelements von <SENSOR> als Parameter übergeben. Die Kindelemente von <SENSOR> sind in diesem Fall <KIND> und <MODE>. Alle Strings, die als Parameter der Methode *readSensor()* in Frage kommen, werden vom *roboXL* Compiler gesammelt (strArg) und als globale Variablen zum spezischen Kernel hinzugefügt. Der Grund dafür ist, dass man auch hier eine Reallokation von Objekten vermeiden muss, für den Fall, dass die *Java Virtual Machine* des Devices keine Garbage Collection unterstützt. Auf die dafür verantwortlichen Teile des *roboXL* Compilers wird hier jedoch nicht näher eingegangen.
Der *roboXL* Compiler kann nur deshalb als generischer Compiler implementiert werden, weil er für jedes Device zum Lesen eines Sensors einfach einen Aufruf der Methode *readSensor()* erzeugen kann, anstatt selbst eine devicespezifische Implementierung dieser Methode angeben zu müssen. Der Rückgabewert von *readSensor()* ist *Object* und den korrekten Typ erhält man durch einen Cast.

Als nächstes wird der Operator des Ausdrucks mit Hilfe der Template Funktion `getOperator` erzeugt:

```
<xsl:template name="getOperator">
  <xsl:param name="op"/>
  <xsl:choose>
    <xsl:when test="$op='less'">
      &lt;
    </xsl:when>
    ...
    <xsl:when test="$op='equal'">
      ==
    </xsl:when>
    <xsl:when test="$op='add'">
      +
    </xsl:when>
    ...
  </xsl:choose>
</xsl:template>
```

Schliesslich wird durch das `transform` Template die rechten Seite des Ausdrucks (`<VALUE>`) transformiert:

```
<xsl:when test="$elem='VALUE'">
  <xsl:value-of select="*"/>
</xsl:when>
```

Betrachten wir nun die Verarbeitung des ersten Falles (`<CASE>`):

```
<CASE>
  <VALUE><BOOL>true</BOOL></VALUE>
  <MOTOR id="1">
    <POWER>1</POWER>
    <FWD/>
  </MOTOR>
  <MOTOR id="2">
    <POWER>1</POWER>
    <FWD/>
  </MOTOR>
</CASE>
```

Das Argument der <SWITCH> Anweisung wird mit dem in <VALUE> definierten Wert verglichen (s. Seite 34). Dieser Vergleich wird in *Java* durch eine *if*-Anweisung realisiert. Alle Kindelemente von <CASE> (ausser <VALUE>) werden ebenfalls durch das transform Template verarbeitet. Der *Java* Code, der aus diesen Kindelementen erzeugt wird, bildet den Rumpf der *if*-Anweisung. Der Transformationsschritt, der noch fehlt, bezieht sich somit auf die <MOTOR> Anweisungen:

```
<xsl:when test="$elem='MOTOR'">
  <xsl:for-each select="*">
    kernel.controlMotor( kernel.strArg<xsl:value-of select="../@id"/>,
                         kernel.strArg<xsl:value-of select="name(.)"/>,
                         <xsl:choose>
                           <xsl:when test=".=''">
                             kernel.strArgEmpty
                           </xsl:when>
                           <xsl:otherwise>
                             kernel.strArg<xsl:value-of select="."/>
                           </xsl:otherwise>
                         </xsl:choose>
                       );
  </xsl:for-each>
</xsl:when>
```

Für jedes Kindelement von <MOTOR> werden das Attribut id von <MOTOR>, der Name des Kindelementes und der Wert des Kindelementes (falls vorhanden) als Parameter an die Funktion *controlMotor()* übergeben. Diese Strings werden ebenfalls vom *roboXL* Compiler als globale Variablen zum spezifischen Kernel hinzugefügt. Die Methode *controlMotor()* wurde in der Devicespezifikation (s. Seite 26) definiert und dient dem devicespezifischen Ansprechen eines Motors. Sie ist, genau wie die Methode *readSensor()*, Bestandteil des spezifischen Kernels.
Die Verarbeitung des zweiten <CASE> Elements verläuft vollkommen analog. Die Verarbeitung des <SWITCH> Elements ist damit abgeschlossen und zu guter letzt wird noch das <SLEEP> Element transformiert:

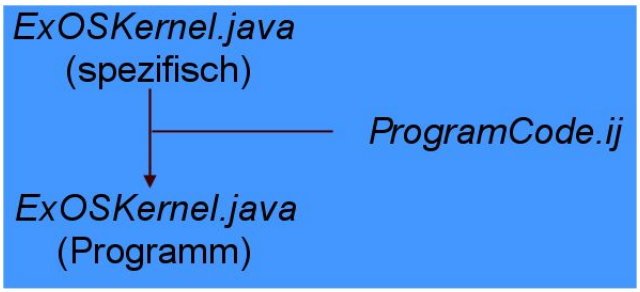
```
<SLEEP unit="msec">500</SLEEP>
```

Der entsprechende Fall im transform Template sieht wie folgt aus:

```
<xsl:when test="$elem='SLEEP'">
  try
  {
  Thread.sleep(
```

```
                <xsl:choose>
                  <xsl:when test="@unit='msec'">
                    <xsl:value-of select="."/>
                  </xsl:when>
                  <xsl:when test="@unit='sec'">
                    <xsl:value-of select="."/> * 1000
                  </xsl:when>
                  <xsl:when test="@unit='min'">
                    <xsl:value-of select="."/> * 60000
                  </xsl:when>
                  <xsl:when test="@unit='hr'">
                    <xsl:value-of select="."/> * 3600000
                  </xsl:when>
                </xsl:choose>
              );
    }
    catch (InterruptedException ie)
  </xsl:when>
```

Es wurden nun alle Teile des *roboXL* Compilers beschrieben, die für unser *roboXL* Kontrollprogramm **FollowMe.xml** relevant sind. Der *roboXL* Compiler übersetzt das *roboXL* Kontrollprogramm in das *Inject/J* Skript **ProgramCode.ij** (s. Abbildung 2.6).



Der Aspekt **ProgramCode.ij** wird nun in das spezifische Kernel *Java* Programm eingewebt. Der spezifische Kernel bekommt die vom *roboXL* Compiler zur Vermeidung von Reallokation gesammelten Strings als Membervariablen (`add to members`). Ausserdem wird der *Java* Code, der aus dem *roboXL* Kontrollprogramm erzeugt wurde, in den bislang noch leeren Rumpf der *main()* Methode eingefügt (`foreach method 'main(*)' do`).
Der Compilevorgang des *roboXL* Kontrollprogramms ist abgeschlossen und das erzeugte *Java* Program kann compiliert und auf das Device aufgespielt werden.

```
script ProgramCode
{
  ...
  in class 'ExOSKernel' do
  {
     add to members ${
       String strArg1="1";
       String strArgint="int";
       String strArglight="light";
       String strArgpct="pct";
       String strArgPOWER="POWER";
       String strArgFWD="FWD";
       String strArgEmpty="";
       String strArg2="2";
       String strArgBWD="BWD";
     }$;

     foreach method 'main(*)' do
     {
        before ${
         ExOSKernel kernel = new ExOSKernel();
         while(true)
         {
            kernel.boolStack.push((((Integer)
             kernel.readSensor( kernel.strArg1,kernel.strArgint,
                                kernel.strArglight,kernel.strArgpct )).getValue()
              < 50 ));

            if (kernel.boolStack.top() == true) {
              kernel.controlMotor(kernel.strArg1, kernel.strArgPOWER, kernel.strArg1);
              kernel.controlMotor(kernel.strArg1, kernel.strArgFWD, kernel.strArgEmpty);
              kernel.controlMotor(kernel.strArg2, kernel.strArgPOWER, kernel.strArg1);
              kernel.controlMotor(kernel.strArg2, kernel.strArgFWD, kernel.strArgEmpty);
            }
            if (kernel.boolStack.top() == false) {
              kernel.controlMotor(kernel.strArg1, kernel.strArgPOWER, kernel.strArg1);
              kernel.controlMotor(kernel.strArg1, kernel.strArgFWD, kernel.strArgEmpty);
              kernel.controlMotor(kernel.strArg2, kernel.strArgPOWER, kernel.strArg1);
              kernel.controlMotor(kernel.strArg2, kernel.strArgBWD, kernel.strArgEmpty);
            }
            kernel.boolStack.pop();
            try {
              Thread.sleep(500);
            } catch (InterruptedException ie) {}
          }
         }$;
     }
  }
}
```

Abbildung 2.6: Ausschnitt von ProgramCode.ij

2.2 Verarbeitung eines *roboXL* Monitoringprogramms

In Kapitel 2.1 haben wir gesehen, wie ein *roboXL* Kontrollprogramm aussieht und wie es transformiert wird. Dabei wurde noch nicht erwähnt, dass in jedes dieser Kontrollprogramme ein Kommunikationsprotokoll eingebettet ist. Mit Hilfe von *roboXL* Monitoringprogrammen soll es zur Laufzeit von Kontrollprogrammen möglich sein, Sensoren, Variablen und den aktuellen Zustand von Motoren abzufragen und Motoren neu zu konfigurieren. Das Kommunikationsprotokoll dient dazu, den dafür notwendigen Datenaustausch durchzuführen.
Abbildung 2.7 zeigt die Komponenten, die die Verarbeitung von *roboXL* Monitoringprogrammen übernehmen. Die gestrichelten Pfeile in Abbildung 2.7 stellen dabei Abhängigkeiten zwischen Komponenten dar, z.B. dass eine Komponente die andere einbindet. Die durchgezogenen Pfeile representieren einzelne Verarbeitungsschritte.

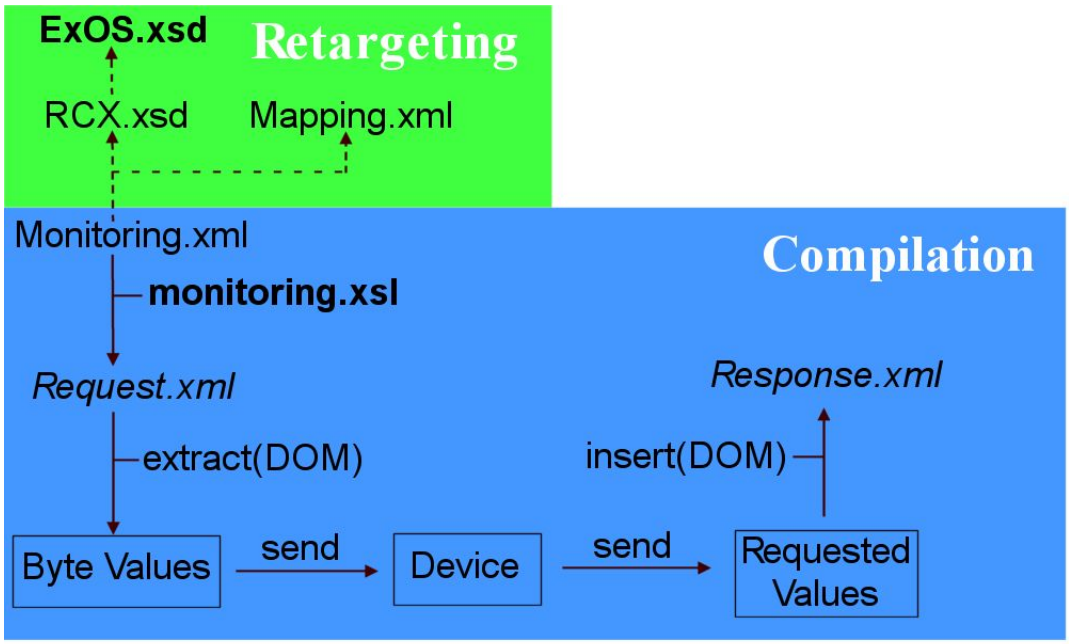


Abbildung 2.7: Verarbeitung eines *roboXL* Monitoringprogramms

2.2.1 Retargeting

Die Implementierung des Kommunikationsprotokolls ist ein weiterer Bestandteil der Devicespezifikation. Somit ist das Kommunikationsprotokoll bereits Teil des spezifischen Kernels. Dieser Abschnitt zeigt eine mögliche Implementierung des Kommunikationsprotokolls für den RCX. Es wird beschrieben, wie das Kommunikationsprotokolls in der Devicespezifikation anzugeben ist und durch den aus der Devicespezifikation erzeugten Aspekt Teil des spezifischen Kernels wird. Obwohl dieser Vorgang bereits beim **Retargeting** während der Verarbeitung eines *roboXL* Kontrollprogramms abläuft, wird er bewusst erst an dieser Stelle erklärt. Denn im Anschluss wird gezeigt, wie das *roboXL* Monitoringprogramm verarbeitet wird und dafür ist eben genau das Kommunikationsprotokoll des spezifischen Kernels relevant.
Kommen wir also nochmals auf die Devicespezifikation für den RCX zurück (s. Abbildung 2.3). Die Devicespezifikationssprache bietet neben den dort gesehenen Elementen auch Elemente zur Definition des Kommunikationsprotokolls. Die Implementierung des Kommunikationsprotokolls muss, wie die restlichen Teile der Devicespezifikation, nur einmal angegeben werden. Abbildung 2.8 zeigt die Implementierung des Protokolls für den RCX. Die Implementierung des Kommunikationsprotokolls ist später in eine eigene Klasse **MonitoringThread.java** gekapselt. Wie der Name schon sagt, ist das Kommunikationsprotokoll als Thread implementiert, der parallel zum Kontrollprogramm läuft. Innerhalb des Elements `<PROTOCOL>` werden die Eigenschaften der Klasse **MonitoringThread.java** angegeben. Zuerst einmal sind dies die *Java* Klassen, die man zur Implementierung des Kommunikationsprotokolls benötigt und importieren möchte (`<IMPORTS>`). Danach können Membervariablen (`<VARIABLES>`) angegeben werden sowie Code, der später in den Konstruktor der Klasse **MonitoringThread.java** eingewebt werden soll (`<INIT>`). Die eigentliche Implementierung wird innerhalb des Elements `<PROCESSBYTECODE>` angegeben. *roboXL* Monitoringprogramme werden im Endeffekt immer in eine Sequenz von Bytewerten übersetzt, die zum Device gesendet werden. Das Kommunikationsprotokoll (der Thread) nimmt die Werte entgegen und führt die gewünschten Operationen aus. Es werden Datenstrukturen benötigt, mit Hilfe derer das Device Werte empfangen und Werte senden kann. Im Falle des RCX bietet sich dazu die Klasse *RCXPort* der *leJOS Java API* an, die einen Input- und einen OutputStream zur Verfügung stellt. Schauen wir uns die Funktionsweise des Kommunikationsprotokolls einmal näher an:
Wird ein Bytewert vom Inputstream gelesen, so wird unterschieden, ob der Zustand eines Motors abgefragt werden soll (in==0), ein Motor neu konfiguriert werden soll (in==1), ein Sensor abgefragt werden soll (in==2) oder eine Variable abgefragt werden soll (in==3). Abbildung 2.8 zeigt nur die Teile, die für die Abfrage eines Sensors relevant sind (in==2). Über den Inputstream wird zuerst die id des Sensors gelesen, gefolgt von dem Typ des Wertes, den der Sensor zurückgeben soll (hierbei steht 4 für int, 5 für boolean und 6 für float), der Art des Sensors und dem Modus des Sensors (beides sind Konstanten der Klasse *SensorConstants* der *leJOS Java API*). Das Device benachrichtigt den Empfänger (die Web Applikation), dass es genau einen Wert senden möchte (objDataOutputStream.writeShort(1)). Zum Lesen des Sensors wird die Methode *readSensorValue()* des

```
<SPECIFICATION>
  ...
  <PROTOCOL>
    <IMPORTS>
      josx.platform.rcx.Sensor,josx.platform.rcx.Motor,josx.rcxcomm.RCXPort,
      java.io.DataInputStream,java.io.DataOutputStream
    </IMPORTS>
    <VARIABLES>
      private RCXPort objRCXPort; private DataInputStream objDataInputStream;
      private DataOutputStream objDataOutputStream;
    </VARIABLES>
    <INIT>
      try{ objRCXPort = new RCXPort();
            objDataInputStream = new DataInputStream(objRCXPort.getInputStream());
            objDataOutputStream = new DataOutputStream(objRCXPort.getOutputStream());
      } catch(IOException ioe) {System.exit(1);}
    </INIT>
    <PROCESSBYTECODE>
    try {
       int in = objDataInputStream.read();
       if(in==0){ ... }
       else if(in==1){ ... }
       else if(in==2){
        in = objDataInputStream.read()-1;
        int returnType = objDataInputStream.read();
        int kind = objDataInputStream.read();
        int mode = objDataInputStream.read();

        objDataOutputStream.writeShort(1);
        if(returnType == 4) {
          objDataOutputStream.writeShort(4);
          objDataOutputStream.writeInt(kernel.readSensorValue(in,kind,mode));
        }
        else if(returnType == 5) {
          objDataOutputStream.writeShort(5);
          if (kernel.readSensorValue(in,kind,mode)==0)
            objDataOutputStream.writeBoolean(false);
          else objDataOutputStream.writeBoolean(true);
        }
        else { objDataOutputStream.writeShort(4);
                objDataOutputStream.writeInt(0);
        }
        objDataOutputStream.flush();
       }
       else if(in==3){ ... }
    }
    catch(IOException ioe){System.exit(1);}
    </PROCESSBYTECODE>
  </PROTOCOL>
</SPECIFICATION>
```

Abbildung 2.8: Devicespezifikation des RCX

spezifischen Kernels verwendet. Danach wird der gelesene Wert entsprechend seines Typs an den Empfänger gesendet. Dabei zeigt 4 (objDataOutputStream.writeShort(4)) dem Empfänger wiederum an, dass der nächste gesendete Wert vom Typ int ist.

Die Klasse **MonitoringThread.java** ist von der Klasse *Thread* abgeleitet. Die `run()` Methode dieser Klasse sieht folgendermaßen aus:

```
public void run()
{
  while(true)
  {
    processByteCode();
  }
}
```

Der Thread ruft also immer wieder die Methode *processByteCode()* auf. In diese Methode wird die Implementierung des Kommunikationsprotokolls eingefügt, die über `<PROCESSBYTECODE>` angegeben wird. Jedes Kontrollprogramm besitzt letztendlich eine Instanz dieser Klasse **MonitoringThread.java** als Membervariable. Wie wir bereits in Kapitel 2.1 gesehen haben, transformiert der **Retargeting Generator** die Devicespezifikation in ein *Inject/J* Skript. Der für das Kommunikationsprotokoll zuständige Ausschnitt des **Retargeting Generators** ist in Abbildung 2.9 zu sehen.

Der *XSLT* Prozessor ersetzt wiederum die *XSLT* Elemente durch den in der Devicespezifikation definierten *Java* Code und erzeugt den Aspekt für das Kommunikationsprotokoll, der sich auf die Klasse **MonitoringThread.java** (`in class 'MonitoringThread' do`) bezieht. Der Aspekt ist somit ein weiterer Bestandteil von **DeviceSpecification.ij** (s. Abbildung 2.5). Das Kommunikationsprotokoll ist demnach bereits im spezifischen Kernel vorhanden.
Wie in Abbildung 2.7 zu sehen ist, wird die Transformation eines *roboXL* Monitoringprogramms durch das *XSLT* Stylesheet **monitoring.xsl** durchgeführt. Abbildung 2.10 zeigt, wie ein *roboXL* Monitoringprogramm aussehen kann. Es wird der Wert des Lichtsensors ermittelt, der auch im *roboXL* Kontrollprogramm **FollowMe.xml** (s. Abbildung 2.2) aus Kapitel 2.1 verwendet wurde. So kann man zur Laufzeit dieses Kontrollprogramms bestimmen, ob sich der Roboter zum aktuellen Zeitpunkt auf der schwarzen Linie befindet oder nicht. Die Syntax des `<SENSOR>` Elements zum Abfragen des Sensorwertes ist identisch mit der Syntax des `<SENSOR>` Elements, das wir bereits im *roboXL* Kontrollprogramm **FollowMe.xml** gesehen haben. Eine Übersicht aller weiteren Elemente von *roboXL* Monitoringprogrammen ist im Anhang zu finden.

```
<xsl:output method="text"/>

<xsl:template match="/">
script DeviceSpecification {
  in class 'MonitoringThread' do
  {
      add to imports ${
        <xsl:value-of select="/exos:EXOS/exos:SPECIFICATION/exos:PROTOCOL/exos:IMPORTS"/>
      }$;

      add to members ${
        <xsl:value-of select="/exos:EXOS/exos:SPECIFICATION/exos:PROTOCOL/exos:VARIABLES"/>
      }$;

      in method 'MonitoringThread(ExOSKernel)' do {
          before ${
            <xsl:value-of select="/exos:EXOS/exos:SPECIFICATION/exos:PROTOCOL/exos:INIT"/>
          }$;
      }

      in method 'processByteCode()' do {
          before ${
   <xsl:value-of select="/exos:EXOS/exos:SPECIFICATION/exos:PROTOCOL/exos:PROCESSBYTECODE"/>
          }$;
      }

  }
```

Abbildung 2.9: Ausschnitt von retarget.xsl

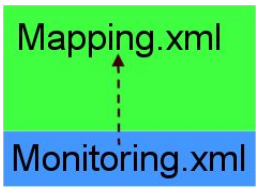
```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="monitoring.xsl"?>
<!DOCTYPE EXOS [ <!ENTITY mapping SYSTEM "Mapping.xml"> ]>

<EXOS xmlns ="ExOSNamespace"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="ExOSNamespace
                          RCX.xsd">

<ROBOXL>
  <MONITORING>
    &mapping;

    <SENSOR id="1" type="int">
      <KIND>light</KIND>
      <MODE>pct</MODE>
    </SENSOR>

  </MONITORING>
</ROBOXL>
</EXOS>
```

Abbildung 2.10: *roboXL* Monitoringprogramm “Monitoring.xml”

Das *XSLT* Stylesheet **monitoring.xsl** ist, wie der *roboXL* Compiler und der **Retargeting Generator**, generisch. Damit **monitoring.xsl** auch devicespezifische Elemente wie `<KIND>` und `<MODE>` verarbeiten kann, ist im Rahmen des **Retargeting** ein weiterer Schritt durchzuführen. Man benötigt ein Mapping zwischen den Inhalten devicespezifischer Elemente (`light` oder `pct`) und den Bytewerten, die zum Device gesendet werden. Dazu dient die Datei **Mapping.xml**:

```
<?xml version="1.0" encoding="UTF-8"?>
<MAPPING>
  ...
  <PAIR>
      <ITEM>light</ITEM>
      <BYTEVALUE>3</BYTEVALUE>
  </PAIR>
  ...
  <PAIR>
      <ITEM>pct</ITEM>
      <BYTEVALUE>128</BYTEVALUE>
  </PAIR>
  ...
</MAPPING>
```

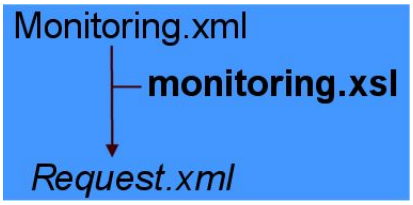
Für jeden möglichen Inhalt eines devicespezifischen Elements gibt es in **Mapping.xml** ein `<PAIR>` Element. Innerhalb dieses `<PAIR>` Elements wird die Zuordnung von Inhalt (`<ITEM>`) und entsprechendem Bytewert (`<BYTEVALUE>`) angegeben.
Warum aber werden gerade die Werte 3 für `light` und 128 für `pct` verwendet? Die *leJOS Java API*, die für die Programmierung des RCX verwendet wird, stellt zur Steuerung eines Sensors die Klasse *Sensor* zur Verfügung. Mit der Methode *setTypeAndMode* wird ein Sensor konfiguriert (s. Abbildung 2.3). Die Argumente dieser Methode sind vordefinierte Konstanten vom Typ int (Klasse *SensorConstants*). Die beim Mapping angegebenen Werte entsprechen exakt diesen Konstanten.



Mapping.xml wird über DTD Syntax mittels einer `ENTITY` automatisch in **Monitoring.xml** eingefügt (`<!DOCTYPE EXOS [ <!ENTITY mapping SYSTEM "Mapping.xml"> ]>` und `&mapping;`). Das Mapping ist eine Komponente, die nur einmal für ein Device angelegt werden muss, dann aber in jedem *roboXL* Monitoringprogramm automatisch eingebunden

wird.
Um ein *XML*-Dokument in ein anderes *XML*-Dokument einzubinden, gibt es in *XML Schema XML Inclusions (XInclude)*. *XInclusion* ist jedoch nicht Bestandteil von *XML* 1.0 und deshalb werden *XInclusions* nicht automatisch von *XML*-Parsern akzeptiert. Man benötigt einen *XInclude Processor*, der die *XInclusions* ersetzt. Die DTD Syntax hingegen wird von den gängigen *XML*-Parsern akzeptiert, ohne dass weitere Software installiert werden muss.

2.2.2 Compilation



Schauen wir uns nun an, wie das *roboXL* Monitoringprogramm transformiert wird. Diese Aufgabe übernimmt das *XSLT* Stylesheet **monitoring.xsl**:

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="1.0"
                xmlns:exos="ExOSNamespace">
<xsl:output method="xml"/>

<xsl:template match="/exos:EXOS">
  <xsl:element name="EXOS" namespace="{namespace-uri(.)}">
    <xsl:element name="ROBOXL" namespace="{namespace-uri(.)}">
      <xsl:element name="RESPONSE" namespace="{namespace-uri(.)}">
        <xsl:for-each select="exos:ROBOXL/exos:MONITORING/*">
          <xsl:choose>
            <xsl:when test="name(.)='GETMOTOR'">
            ...
            </xsl:when>
            <xsl:when test="name(.)='MOTOR'">
            ...
            </xsl:when>
            <xsl:when test="name(.)='SENSOR'">
            ...
            </xsl:when>
            <xsl:when test="name(.)='GETVAR'">
            ...
```

```
            </xsl:when>
          </xsl:choose>
        </xsl:for-each>
      </xsl:element>
    </xsl:element>
  </xsl:element>
</xsl:template>
```

Der Output des *XSLT* Stylesheets **monitoring.xsl** ist diesmal kein *Inject/J* Skript sondern wiederum ein *XML*-Dokument. Da Monitoringprogamme verwendet werden, um zur Laufzeit eines Kontrollprogramms bestimmte Werte abzufragen, müssen diese abgefragten Werte auch wieder an den Client, der das Monitoringprogramm gesendet hat zurückgegeben werden. Das aus einem *roboXL* Monitoringprogramm erzeugte *XML*-Dokument, das sogenannte *roboXL* Requestdokument, besitzt deshalb schon die Grundstruktur des Dokuments, das später als Antwort an den Client zurückgegeben wird, des sogenannten *roboXL* Responsedokuments. Deshalb trägt das erzeugte Kindelement des Elements `<ROBOXL>` bereits an dieser Stelle den Namen `<RESPONSE>`.

```
<EXOS>
  <ROBOXL>
    <RESPONSE>
    ...
    </RESPONSE>
  </ROBOXL>
</EXOS>
```

Das *roboXL* Requestdokument wird als Zwischenablage für die Bytewerte benutzt, die aus dem *roboXL* Monitoringprogramm erzeugt und zum Device gesendet werden. Der für unser Monitoringprogramm relevante Fall ist `SENSOR`.

```
<!-- Sensor Request -->
<xsl:when test="name(.)='SENSOR'">
  <xsl:element name="SENSORREQUEST" namespace="{namespace-uri(.)}">
    <!-- Parameters of the request -->
    <xsl:element name="PARAM" namespace="{namespace-uri(.)}">
      <xsl:attribute name="name">
        <xsl:value-of select="'id'"/>
      </xsl:attribute>
      <xsl:value-of select="@id"/>
    </xsl:element>
```

```
<xsl:element name="PARAM" namespace="{namespace-uri(.)}">
  <xsl:attribute name="name">
    <xsl:value-of select="'type'"/>
  </xsl:attribute>
  <xsl:value-of select="@type"/>
</xsl:element>

<xsl:for-each select="*">
  <xsl:element name="PARAM" namespace="{namespace-uri(.)}">
    <xsl:attribute name="name">
      <xsl:value-of select="name(.)"/>
    </xsl:attribute>
    <xsl:value-of select="."/>
  </xsl:element>
</xsl:for-each>

<!-- bytecode according to the request -->
<xsl:element name="REQUEST" namespace="{namespace-uri(.)}">

  <!-- 2 to indicate a sensor request -->
  <xsl:element name="BYTECODE" namespace="{namespace-uri(.)}">
    <xsl:value-of select="'2'"/>
  </xsl:element>

  <!-- id of the requested sensor -->
  <xsl:element name="BYTECODE" namespace="{namespace-uri(.)}">
    <xsl:value-of select="@id"/>
  </xsl:element>

  <!-- type of the requested sensor -->
  <xsl:element name="BYTECODE" namespace="{namespace-uri(.)}">
    <xsl:choose>
      <xsl:when test="@type='int'">
        <xsl:value-of select="'4'"/>
      </xsl:when>
      <xsl:when test="@type='bool'">
        <xsl:value-of select="'5'"/>
      </xsl:when>
      <xsl:when test="@type='float'">
        <xsl:value-of select="'6'"/>
      </xsl:when>
    </xsl:choose>
  </xsl:element>
```

```
        <!-- translate the additional parameters of the sensor request -->
        <xsl:for-each select="*">
          <xsl:call-template name="map">
            <xsl:with-param name="item" select="."/>
          </xsl:call-template>
        </xsl:for-each>
      </xsl:element>

    </xsl:element>
  </xsl:when>
```

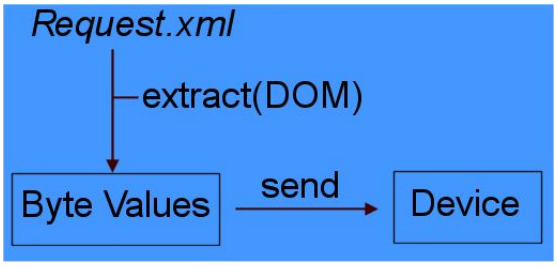
Die Angabe von `namespace="{namespace-uri(.)}"` ist notwendig, um im zu erzeugenden *roboXL* Requestdokument eine korrekte Zuordnung der Namensräume zu erreichen. Für jedes `<SENSOR>` Element wird im *roboXL* Requestdokument ein `<SENSORREQUEST>` Element erzeugt. Zum Erzeugen von Elementen steht in *XSLT* das Element `<xsl:element>` zur Verfügung. Für Attribute entsprechend `<xsl:attribute>`. Dem `<SENSORREQUEST>` Element werden zuerst die Parameter (`<PARAM>`) des `<SENSOR>` Elements hinzugefügt, d.h. id und type sowie die Werte aller Kindelemente von `<SENSOR>` (`<KIND>` und `<MODE>`). Nur so kann ein abgefragter Sensorwert später wieder eindeutig einer mit dem `<SENSOR>` Element definierten Abfrage zugeordnet werden.
Anschliessend wird ein `<REQUEST>` Element erzeugt. Zu Beginn dieses Kapitels haben wir gesehen, dass das Kommunikationsprotokoll spezielle Bytewerte erwartet (s. Abbildung 2.8). Die Abfrage eines Sensors wird durch das Senden einer Folge von Bytewerten zum Device getätigt. Diese Folge von Bytewerten wird aus dem `<SENSOR>` Element unseres *roboXL* Monitoringprogramms generiert und innerhalb des `<REQUEST>` Elements gespeichert. Um die devicespezischen Parameter des `<SENSOR>` Elements in Bytewerte zu übersetzen, bedient sich das **monitoring.xsl** Stylesheet nun dem in **Mapping.xml** definierten Mapping. Im *XSLT* Stylesheet geschieht dies in der Template Funktion `map`:

```
<xsl:template name="map">
<xsl:param name="item"/>

  <xsl:element name="BYTECODE" namespace="{namespace-uri(.)}">
  <xsl:for-each select=
  "/exos:EXOS/exos:ROBOXL/exos:MONITORING/exos:MAPPING/exos:PAIR/exos:ITEM">
      <xsl:if test="$item=.">
        <xsl:value-of select="../exos:BYTEVALUE"/>
      </xsl:if>
  </xsl:for-each>
  </xsl:element>

</xsl:template>
```

Alles in allem sieht unser durch **monitoring.xsl** erzeugtes *roboXL* Requestdokument **Request.xml** folgendermaßen aus:

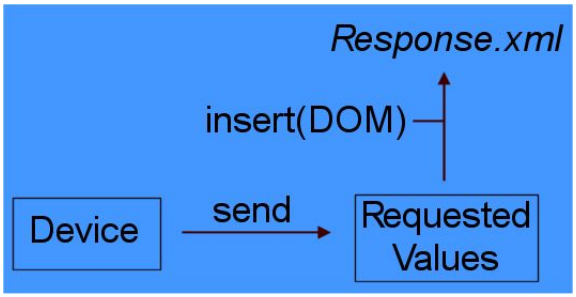
```
<?xml version="1.0" encoding="UTF-8"?>
<EXOS xmlns:exos="ExOSNamespace"
      xmlns="ExOSNamespace">

 <ROBOXL>
   <RESPONSE>
     <SENSORREQUEST>
       <PARAM name="id">1</PARAM>
       <PARAM name="type">int</PARAM>
       <PARAM name="KIND">light</PARAM>
       <PARAM name="MODE">pct</PARAM>

       <REQUEST>
         <BYTECODE>2</BYTECODE>
         <BYTECODE>1</BYTECODE>
         <BYTECODE>4</BYTECODE>
         <BYTECODE>3</BYTECODE>
         <BYTECODE>128</BYTECODE>
       </REQUEST>
     </SENSORREQUEST>
   </RESPONSE>
 </ROBOXL>
 </EXOS>
```

Momentan ist die Folge von Bytewerten für den Datenaustausch mit dem Device (Request) also im Element `<REQUEST>` gespeichert. Diese Bytewerte werden nun aus **Request.xml** extrahiert und zum Device gesendet.



Das Kommunikationsprotokoll nimmt die Werte entgegen und sendet den gelesenen Sensorwert zurück. Die gespeicherte Folge von Bytewerten wird durch die Werte ersetzt,

die das Kommunikationsprotokoll zurückliefert. Aus **Request.xml** wird dadurch das *roboXL* Responsedokument **Response.xml**, das an den Client zurückgegeben wird.



Möchte man die Struktur von *XML*-Dokumenten durch Hinzufügen oder Entfernen von Elementen verändern, bietet sich die *Document Object Model (DOM) API* an, die in der *Java API for XML Processing (JAXP)* als Parser Standard zur Verfügung steht. Dabei wird eine baumartige Objektrespresentation des gesamten *XML*-Dokuments erzeugt, die wie jede andere baumartige Datenstruktur auch, manipuliert werden kann. Auf diese Art und Weise wird das `<REQUEST>` Element entfernt und durch das Element `<RETURN>` ersetzt, dem die Rückgabewerte, in unserem Fall der gelesene Sensorwert, hinzugefügt werden. Das *roboXL* Responsedokument **Response.xml** sieht wie folgt aus:

```
<?xml version="1.0" encoding="UTF-8"?>
<EXOS xmlns="ExOSNamespace"
      xmlns:exos="ExOSNamespace">
<ROBOXL>
  <RESPONSE>
    <SENSORREQUEST>
      <PARAM name="id">1</PARAM>
      <PARAM name="type">int</PARAM>
      <PARAM name="KIND">light</PARAM>
      <PARAM name="MODE">pct</PARAM>

      <RETURN>
        <RETURNVALUE>36</RETURNVALUE>
      </RETURN>
    </SENSORREQUEST>
  </RESPONSE>
</ROBOXL>
</EXOS>
```

Der Lichtsensor hat eine Helligkeit von 36% gemessen. Daraus folgt, dass sich der Roboter momentan auf der schwarzen Linie befindet.

Kapitel 3

Bewertung und Rückblick

Die *XML*-Programmiersprache *roboXL* wurde in erster Linie entworfen, um einen Vorteil aus den vielen verschiedenen *XML*-Tools zu ziehen, die heutzutage verfügbar sind. *roboXL* Dokumente sind demnach leicht zu erstellen und zu bearbeiten. Das *XML*-Format ermöglicht einen einfachen Datenaustausch. Die Menge der Sprachelemente von *roboXL* ist leicht überschaubar und bietet dennoch die meisten, auch in anderen Programmiersprachen gängigen Programmierkonstrukte.
Durch Anlehnung an die Programmiersprache *Java* beim Entwurf von *roboXL* erreichten wir, dass sich *roboXL* Kontrollprogramme leicht in *Java* Code übersetzen lassen. Wie wir gesehen haben, lässt sich der *roboXL* Compiler einfach als *XSLT* Stylesheet implementieren. Komplexe Programmanalysen lassen sich mit *XSLT* jedoch nicht durchführen. Bestimmte Programmierfehler in *roboXL* Kontrollprogrammen (z.B. in Bezug auf Typsicherheit) werden deshalb erst beim Compilieren des erzeugten *Java* Programms durch den *Java* Compiler aufgedeckt.
Bei der Implementierung des *roboXL* Compilers gibt es devicespezifische Teile, die bei jedem Wechsel des Devices eine erneute Implementierung des Compilers nötig machen würden. Um dies zu vereinfachen, wurde ein generischer Compiler implementiert und die devicespezifischen Teile in einer Devicespezifikation definiert. Ein generisches Kernel *Java* Programm wird um eben diese Devicespezifikation und den vom Compiler aus einem *roboXL* Kontrollprogramm erzeugten *Java* Code erweitert. Auf diese Weise wird das *Java* Programm generiert, das letztendlich auf das Device geladen wird. Für die Implementierung des Retargeting Generators, der die Devicespezifikation verarbeitet, wurde ebenfalls *XSLT* eingesetzt.
Die klare Aufteilung der Codefragmente des generischen Kernels, der devicespezifischen Erweiterungen und des eigentlichen Programmcodes in verschiedene Komponenten führte zu einem gut überschaubaren und durchführbaren Architekturentwurf. Die Kombination von *XSLT* und *AOP* erwies sich als gleichermaßen einfach wie elegant. Das Vorhandensein von Technologien wie *XSLT* und *Inject/J* erlaubte es, sich in erster Linie auf die Trennung der wiederverwendbaren und spezifischen Teile der Implementierung zu konzentrieren und weniger auf den Transformationsprozess selbst. Das Retargeting, d.h. die Anpassung des Transformationsvorgangs der *roboXL* Dokumente (Kontroll- und Monitoringprogramme)

an ein bestimmtes Device, lässt sich demnach mit möglichst geringem Arbeitsaufwand durchführen.

Eine der Herausforderungen dieser Arbeit lag darin, die vielen unterschiedlichen Technologien, die verwendet wurden, zu einem Gesamtsystem zusammenzusetzen. Die verschiedenen Technologien benötigen natürlich auch eine Reihe an zusätzlicher Software und Tools, die installiert werden muss. An dieser Stelle ein kurzer Überblick:
Der gesamte Verarbeitungsprozess wurde mit dem J2SDK 1.4.2_04 implementiert. Das J2SDK stellt mit *Xalan* bereits einen *XSLT* Processor zur Verfügung. Um *XML Schema* in der Programmiersprache *Java* verarbeiten zu können, benötigt man die *Java API for XML Processing (JAXP)*, die im Java Web Services Developer Pack 1.3 [5] enthalten ist und den benötigten *Apache Xerces Parser* enthält, der den im J2SDK standardmäßig enthaltenen *Crimson Parser* ersetzt. Bei der Verarbeitung der *XML*-Dokumente kommen die *Simple API for XML (SAX)* und die *Document Object Model (DOM) API* zum Einsatz. Die Verwendung dieser APIs ist in [5] und [6] beschrieben.
Alle Tests wurden mit dem Lego Mindstorms RCX durchgeführt [16]. Die ursprüngliche Lego Firmware des Lego Mindstorms RCX wurde mit der *leJOS* Firmware ersetzt, die eine *Java Virtual Machine* implementiert. Die Programmierung des RCX erfolgt mit Hilfe der *leJOS Java API*. [7]. Zum automatischen Compilieren und Downloaden der aus den *roboXL* Kontrollprogrammen generierten *Java* Programme verwenden wir das Framework **Integrating Smart Devices into Java Applications** von Marc Jansen [14]. Über dieses Framework wird auch der Datenaustausch mit dem RCX bei der Verarbeitung eines *roboXL* Monitoringprogramms realisiert.
Die Web Applikation, die die in Kapitel 2 vorgestellte Systemarchitektur implementiert, wurde mit dem *Apache Axis Framework 1.2 RC2* [1] entwickelt und auf einem *Apache Tomcat Server 5.5.4* [2] eingerichtet. Genauergesagt handelt es sich bei der Web Applikation um einen *Axis* Web Service. Sowohl die Verarbeitung von *roboXL* Kontrollprogrammen als auch *roboXL* Monitoringprogrammen wird von dem Web Service serverseitig ausgeführt. Zur Übermittlung von Dateien bietet *Axis* eine API für Attachments an. Die *roboXL* Dokumente werden als Attachments an die Aufrufe der Methoden des Web Services angehängt. Zur Interaktion mit dem Web Service steht eine grafische Client-Applikation in Form eines *Java* Applets zur Verfügung.

Im Verlauf des Implementierungsprozesses sind vor allem drei Punkte zu erwähnen. Zum einen die Tatsache, dass der RCX keine Garbage Collection unterstützt. Dies hatte zur Folge, dass die gesamte Implementierung darauf ausgelegt ist, eine Reallokation von Objekten zu vermeiden. Zum anderen wurden Probleme verursacht durch einen Bug im *Inject/J* Compiler, die jedoch durch Einfügen zusätzlichen *Java* Codes in die *Inject/J* Skripte (Aspekte) behoben werden konnten. Ein Problem, zu dem keine Lösung gefunden werden konnte, ist die Synchronisierung der Sensorzugriffe in *leJOS*. Das Problem tritt nur in einem ganz bestimmten Fall auf und zwar wenn mehrere Threads versuchen, denselben Sensor in unterschiedlichen Modi zu verwenden. Ein Beispiel wäre ein Lichtsensor, mit dem man die Helligkeit sowohl in Form von Rohdaten als auch in Prozent messen möchte. Die

Sensorzugriffe können dabei Teil von Threads sein, die in *roboXL* Kontrollprogrammen auftreten, aber auch durch den Thread veranlasst werden, in dem das Kommunikationsprotokoll implementiert ist und der durch ein *roboXL* Monitoringprogramm definierte Sensorzugriffe verarbeitet. Das Problem begründet sich darin, dass die Sensoren in *leJOS* über statische Instanzen der Klasse *Sensor* angesprochen werden, die Teil der *leJOS Java API* ist. Das auch in *Java* zur Synchronisation verwendete *synchronized*-Konstrukt zeigt für Methodenaufrüfe über statische Variablen keine Wirkung, wie aus der *leJOS* Mailingliste zu entnehmen ist.

Die Sprache *roboXL* ist darauf ausgelegt, Sensoren und Motoren eines Devices auf einfache Weise konfigurieren und nutzen zu können. Haupteinsatzgebiet von *roboXL* ist demnach die Programmierung von Sensoren und Motoren in allen möglichen Varianten. Grundlegende Programmierkenntnisse sind für eine erfolgreiche Benutzung von *roboXL* jedoch notwendig. Um die Benutzung von *roboXL* weiter zu vereinfachen, wäre es im Rahmen einer weiterführenden Arbeit denkbar, eine visuelle Programmierumgebung zu entwerfen. Der Benutzer könnte so seine Programme per "Drag and Drop" zusammenstellen und das entsprechende *roboXL* Dokument würde automatisch erzeugt werden. Dies würde *roboXL* auch für Benutzer interessant machen, die über keine bzw. nur über sehr geringe Programmiererfahrung verfügen. Sieht man von den bereits angesprochenen Vorteilen, die *XML* bietet einmal ab, so sei natürlich die Frage erlaubt, ob es für einen mit *Java* vertrauten Benutzer nicht einfacher wäre, seine Programme direkt in *Java* (z.B. im Falle des RCX mit *leJOS*) zu erstellen. Prinzipiell schon, allerdings gibt es mehrere Argumente, die den Einsatz von *roboXL* rechtfertigen. Ein Vorteil unseres Systems besteht darin, dass Download und Start der Programme vollautomatisch durchgeführt werden. Dazu gehört auch das Compilieren der erzeugten *Java* Programme. Zudem besteht die Möglichkeit die Programmierung eines Devices über das Netzwerk zu ändern. Programmieraufwand wird eingespart, indem die Teile, die sich in jedem Programm wiederholen, anfangs einmal in einer eigenen Komponente (generischer Kernel) gekapselt und später automatisch hinzugefügt werden. Die Syntax der Kontroll- und Monitoringprogramme ist, abgesehen von geringfügigen Unterschieden bei der Verwendung devicespezifischer Elemente wie `<SENSOR>` und `<MOTOR>`, auch für verschiedene Devices vollkommen gleich. Nach dem Retargeting lassen sich verschiedene Devices allein mit *XML* programmieren. Programmiert man die Devices hingegen direkt in *Java*, muss man die Programme immer an die aktuell verwendete *Java* API anpassen.

Ein großes Manko bleibt trotz allem das Fehlen eines weiteren Devices. In Betracht gezogen hatten wir den Leonardo von Phantom 2 [19]. Es gibt zwar eine *Java* API für den Leonardo, jedoch keine *Java Virtual Machine*. Somit können keine Programme in Form von *Java* Bytecode auf den Leonardo geladen werden, was eine Grundvoraussetzung für den Einsatz unseres Systems ist. Ein vielversprechendes Device ist der JCX [21], den wir von Anfang an als weiteres Device neben dem RCX verwenden wollten. Allerdings befindet sich der JCX immer noch im Entwicklungsstadium und ist zum gegenwärtigen Zeitpunkt noch nicht erhältlich. Für den JCX soll eine *Java* API zur Verfügung stehen,

die vergleichbar mit *leJOS* ist. Eine Vorabversion dieser *Java* API steht bereits auf der JCX Webseite [21] zum Download zur Verfügung. Möchte man *roboXL* für den JCX verwenden, so sind folgende Schritte auszuführen:

1. Erstellen eines devicespezifischen *XML Schemas* für den JCX (analog zu **RCX.xsd**).

2. Erstellen einer Devicespezifikation für den JCX (analog zu **RCX.xml**). Bei der Implementierung der Methoden *controlMotor()* und *readSensor()* sowie des Kommunikationsprotokolls kann dabei die Implementierung für den RCX als Vorlage verwendet werden. Es muss lediglich eine Anpassung an die Methoden der Klassen *Sensor* und *Motor* der JCX *Java* API vorgenommen werden.

3. Anlegen einer neuen Mapping-Datei (analog zu **Mapping.xml**), die während der Verarbeitung von *roboXL* Monitoringprogrammen benötigt wird.

Danach können sowohl der RCX als auch der JCX mit *roboXL* programmiert werden. Der JCX wird wesentlich leistungsfähiger sein als der RCX und es ist davon auszugehen, dass in Zukunft weitere vergleichbare Devices auf den Markt kommen werden, an die sich unser System durch Retargeting anpassen lässt. Auch ist es denkbar, *roboXL* als Grundlage für eine *XML*-Sprache zu verwenden, die den Ansprüchen weitaus komplexerer Devices gerecht wird. Zum Beispiel Steuerungsdevices in der Automobilindustrie, die mittels einer *Java* API Komponenten steuern, die vom Grundprinzip Sensoren und Motoren ähneln.

Kapitel 4

Anhang

4.1 Vollständge Beschreibung der definierten *XML*-Sprachen

Dieses Kapitel widmet sich der vollständigen Beschreibung der *XML*-basierten Programmiersprache *roboXL* und der *XML*-basierten Devicespezifikationssprache. Beide Sprachen sind im *XML Schema* **ExOS.xsd** definiert. Es sei darauf hingewiesen, dass *roboXL* noch devicespezifisch erweitert werden muss, denn nur die deviceunabhängigen Sprachkonstrukte sind in **ExOS.xsd** definiert. In Kapitel 2.1 haben wir gesehen, wie diese Erweiterung im Falle des RCX aussieht (**RCX.xsd**). Auf die devicespezifische Erweiterung von **ExOS.xsd** wird in diesem Kapitel deshalb nicht weiter eingegangen.
Sowohl Instanzdokumente der Sprache *roboXL* als auch Instanzdokumente der Devicespezifikationssprache haben das Element `<EXOS>` als Wurzelelement. Als Kindelement von `<EXOS>` wählt man nun `<ROBOXL>`, wenn man ein *roboXL* Dokument erstellen möchte, d.h. ein *roboXL* Kontroll- oder Monitoringprogramm, oder man wählt `<SPECIFICATION>`, um eine Devicespezifikation mit der *XML*-basierten Devicespezifikationssprache zu erstellen. Auf den folgenden Seiten werden die dazugehörigen Sprachkonstrukte vorgestellt.

```
<xsd:element name="EXOS">
<xsd:complexType>
    <xsd:choice>
      <xsd:element name="ROBOXL">
      ...
      </xsd:element>
      <xsd:element name="SPECIFICATION">
      ...
      </xsd:element>
    </xsd:choice>
</xsd:complexType>
</xsd:element>
```

4.1.1 Die Sprache *roboXL*

Dieses Kapitel widmet sich der genauen Beschreibung der *XML*-basierten Programmiersprache *roboXL*. *roboXL* definiert eine *XML*-Syntax zur Erstellung von *roboXL* Kontollprogrammen (<PROGRAM>), zur Erstellung von *roboXL* Monitoringprogrammen (<MONITORING>) und zur Erstellung von *roboXL* Request- bzw. Responsedokumenten (<RESPONSE>).

```
<xsd:element name="ROBOXL">
   <xsd:complexType>
      <xsd:choice>
         <xsd:element name="PROGRAM">
         ...
         </xsd:element>
         <xsd:element name="MONITORING">
         ...
         </xsd:element>
         <xsd:element name="RESPONSE">
         ...
         </xsd:element>
      </xsd:choice>
   </xsd:complexType>
</xsd:element>
```

Elemente in *roboXL* Kontrollprogrammen

Dieser Abschnitt befasst sich mit der Syntax von *roboXL* Kontrollprogrammen. Sieht man einmal von der schlecht leserlichen *XML*-Syntax von *roboXL* Kontrollprogrammen ab, so sind diese der Programmiersprache *Java* sehr ähnlich. *roboXL* stellt Konstrukte bereit für:

- **Variablendeklaration**

```
<xsd:element name="VAR">
   <xsd:complexType>
      <xsd:attribute name="name" type="xsd:string"
                     use="required"/>
      <xsd:attribute name="type" use="required">
         <xsd:simpleType>
            <xsd:restriction base="xsd:string">
               <xsd:enumeration value="bool"/>
               <xsd:enumeration value="int"/>
               <xsd:enumeration value="float"/>
```

```
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:attribute>
        <xsd:attribute name="id" use="optional">
          <xsd:simpleType>
            <xsd:restriction base="xsd:integer">
            <xsd:minInclusive value="1"/>
            <xsd:maxInclusive value="10"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:attribute>
      </xsd:complexType>
    </xsd:element>
```

<VAR> dient allein der Deklaration einer Variablen. Es werden lediglich Name und Typ einer Variablen festgelegt. Variablen können vom Typ bool, int oder float sein. Optional kann der Variablen eine **id** zugeordnet werden, über die sie zur Laufzeit des Kontrollprogramms mit Hilfe eines *roboXL* Monitoringprogramms abgefragt werden kann (s. Seite 70).

```
<xsd:element name="PROGRAM">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="VAR" minOccurs="0" maxOccurs="10"/>
      ...
    </xsd:sequence>
  </xsd:complexType>
  <xsd:key name="Var">
    <xsd:selector xpath="exos:VAR"/>
    <xsd:field xpath="@name"/>
  </xsd:key>
  ...
```

Es können maximal zehn Variablen deklariert werden. Das Attribut name wird als Key definiert, um zu gewährleisten, dass jeder Variablenname im gesamten Kontrollprogramm eindeutig ist.

- **Wertzuweisung an eine Variable**

```
<xsd:element name="SETVALUE">
   <xsd:complexType>
      <xsd:choice>
         <xsd:element ref="EXPRESSION"/>
         <xsd:element ref="VALUE"/>
         <xsd:element ref="SENSOR"/>
         <xsd:element ref="GETVALUE"/>
      </xsd:choice>
   <xsd:attribute name="name" type="xsd:string" use="required"/>
   </xsd:complexType>
</xsd:element>
```

Über `<SETVALUE>` kann einer Variablen ein Wert zugewiesen werden. Der Bezug zur Variablen wird über das Attribut `name` hergestellt. Dabei wird anhand des oben definierten Keys durch eine Keyreferenz überprüft, ob eine Variable mit diesem Namen existiert.

```
<xsd:element name="PROGRAM">
  ...
  <xsd:keyref name="SetVar" refer="Var">
    <xsd:selector xpath=".//exos:SETVALUE"/>
    <xsd:field xpath="@name"/>
  </xsd:keyref>
  ...
```

Einer Variablen kann der Wert eines Ausdrucks (`<EXPRESSION>`), ein einfacher Wert mit gleichem Typ wie die Variable (`<VALUE>`), der Wert eines Sensors (`<SENSOR>`) oder der Wert einer anderen Variablen (`<GETVALUE>`) zugewiesen werden.

- **Auslesen einer Variable**

```
<xsd:element name="GETVALUE">
   <xsd:complexType>
      <xsd:attribute name="name" type="xsd:string"
                     use="required"/>
   </xsd:complexType>
</xsd:element>
```

Das Auslesen einer Variable erfolgt mit Hilfe des Elements `<GETVALUE>`. Der Name der auszulesenden Variable wird als Attribut angegeben. Wiederum wird mit Hilfe des oben definierte Keys überprüft, ob eine Variable mit diesem Namen existiert.

```
<xsd:element name="PROGRAM">
  ...
  <xsd:keyref name="GetVar" refer="Var">
    <xsd:selector xpath=".//exos:GETVALUE"/>
    <xsd:field xpath="@name"/>
  </xsd:keyref>
  ...
```

- **Definition eines einfachen Wertes**

```
<xsd:element name="VALUE">
   <xsd:complexType>
      <xsd:choice>
         <xsd:element name="BOOL">
            <xsd:simpleType>
               <xsd:restriction base="xsd:string">
                  <xsd:enumeration value="false"/>
                  <xsd:enumeration value="true"/>
               </xsd:restriction>
            </xsd:simpleType>
         </xsd:element>
         <xsd:element name="INT" type="xsd:int"/>
         <xsd:element name="FLOAT" type="xsd:float"/>
      </xsd:choice>
   </xsd:complexType>
</xsd:element>
```

Die erlaubten Werte umfassen die Werte der primitiven Datentypen boolean, int und float in der Programmiersprache *Java*.

- **Anhalten des Programmflusses für eine bestimmte Zeitspanne**

```
<xsd:element name="SLEEP">
   <xsd:complexType>
      <xsd:simpleContent>
         <xsd:extension base="xsd:integer">
            <xsd:attribute name="unit" use="required">
               <xsd:simpleType>
                  <xsd:restriction base="xsd:string">
                     <xsd:enumeration value="msec"/>
                     <xsd:enumeration value="sec"/>
                     <xsd:enumeration value="min"/>
                     <xsd:enumeration value="hr"/>
                  </xsd:restriction>
               </xsd:simpleType>
            </xsd:attribute>
         </xsd:extension>
      </xsd:simpleContent>
   </xsd:complexType>
</xsd:element>
```

  Durch <SLEEP> wird der aktuelle Thread für die angegebene Zeitspanne schlafen gelegt. Die Zeiteinheit kann über das Attribut unit spezifiziert werden. Eine kurze Erklärung, um die Definition des Elements <SLEEP> nachvollziehen zu können: Der einfache Inhalt des Elements <SLEEP> ist vom Typ xsd:integer (<xsd:simpleContent>) und wird um ein Attribut erweitert. Da das Element <SLEEP> ein Attribut besitzt, benötigt man <xsd:complexType>.

- **Ansprechen eines Motors**
  Ein Motor wird mit <MOTOR> angesprochen. <MOTOR> wird in der devicespezifischen Erweiterung von **ExOS.xsd** definiert wird (s. dazu **RCX.xsd** in Kapitel 2.1.1).

- **Ansprechen eines Sensors**
  Ein Sensor wird mit <SENSOR> angesprochen. <SENSOR> wird in der devicespezifischen Erweiterung von **ExOS.xsd** definiert wird (s. dazu **RCX.xsd** in Kapitel 2.1.1).

- **Ausdrücke**

```
<xsd:element name="EXPRESSION">
   <xsd:complexType>
      <xsd:choice minOccurs="2" maxOccurs="2">
         <xsd:element ref="SENSOR"/>
         <xsd:element ref="VALUE"/>
         <xsd:element ref="GETVALUE"/>
         <xsd:element ref="EXPRESSION"/>
      </xsd:choice>
      <xsd:attribute name="op" use="required">
         <xsd:simpleType>
            <xsd:restriction base="xsd:string">
               <xsd:enumeration value="less"/>
               <xsd:enumeration value="greater"/>
               <xsd:enumeration value="equal"/>
               <xsd:enumeration value="unequal"/>
               <xsd:enumeration value="div"/>
               <xsd:enumeration value="mul"/>
               <xsd:enumeration value="add"/>
               <xsd:enumeration value="sub"/>
               <xsd:enumeration value="and"/>
               <xsd:enumeration value="or"/>
            </xsd:restriction>
         </xsd:simpleType>
      </xsd:attribute>
      <xsd:attribute name="type" use="required">
         <xsd:simpleType>
            <xsd:restriction base="xsd:string">
               <xsd:enumeration value="bool"/>
               <xsd:enumeration value="int"/>
               <xsd:enumeration value="float"/>
            </xsd:restriction>
         </xsd:simpleType>
      </xsd:attribute>
   </xsd:complexType>
</xsd:element>
```

Mit Hilfe `<EXPRESSION>` können bool'sche und arithmetische Ausdrücke definiert werden. Ein Ausdruck besteht immer aus einer linken und einer rechten Seite, die über einen Operator zueinander in Beziehung gesetzt werden. Als Beispiel: 3<4 (linke Seite: 3, rechte Seite: 4, Operator: <). Als linke bzw. rechte Seite kommen Sensorwerte

(<SENSOR>), einfache Werte (<VALUE>), Werte von Variablen (<GETVALUE>) und wiederum Ausdrücke (<EXPRESSION>) in Frage. Aufgrund der Tatsache, dass innerhalb von Ausdrücken wiederum andere Asudrücke stehen können, lassen sich verschachtelte Ausdrücke bilden. Der Operator wird über das Attribut op angegeben. Das Attribut type spezifiziert den Typ des Wertes, den man erhält, wenn der Ausdruck vollständig ausgewertet ist.

- **Bedingtes Warten**

```
<xsd:element name="WAITFOR">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="EXPRESSION"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

Das Element <WAITFOR> wird verwendet, um den aktuellen Thread solange anzuhalten, bis der Wert des mit <EXPRESSION> definierten Ausdrucks "true" ist. Dies hat zur Folge, dass das Attribut type eines in <WAITFOR> definierten Ausdrucks immer den Wert "bool" haben muss.

- **Schleifen**

```
<xsd:element name="LOOP">
  <xsd:complexType>
    <xsd:choice minOccurs="0" maxOccurs="unbounded">
                <xsd:element ref="MOTOR"/>
                <xsd:element ref="SLEEP"/>
                <xsd:element ref="LOOP"/>
                <xsd:element ref="SWITCH"/>
                <xsd:element ref="WAITFOR"/>
                <xsd:element ref="PARALLEL"/>
                <xsd:element ref="SETVALUE"/>
    </xsd:choice>
    <xsd:attribute name="counter" type="xsd:nonNegativeInteger"
                   use="required"/>
  </xsd:complexType>
</xsd:element>
```

Das Element <LOOP> dient der wiederholten Ausführung von Anweisungen. Mit Hilfe des Attributs counter wird die Anzahl der Wiederholungen angegeben. Setzt man

counter auf den Wert 0, so wird eine Endlosschleife ausgeführt. Die Definition der meisten Elemente, die innerhalb von <LOOP> auftreten können, haben wir bereits gesehen. Die Elemente <PARALLEL> und <SWITCH> sind wie folgt definiert:

- **Parallele Tasks**

```
<xsd:element name="PARALLEL">
  <xsd:complexType>
    <xsd:sequence minOccurs="1" maxOccurs="unbounded">
      <xsd:element ref="TASK"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:element name="TASK">
  <xsd:complexType>
    <xsd:choice minOccurs="0" maxOccurs="unbounded">
        <xsd:element ref="MOTOR"/>
        <xsd:element ref="SLEEP"/>
        <xsd:element ref="LOOP"/>
        <xsd:element ref="SWITCH"/>
        <xsd:element ref="WAITFOR"/>
        <xsd:element ref="SETVALUE"/>
    </xsd:choice>
  </xsd:complexType>
</xsd:element>
```

Das Element <PARALLEL> dient, wie der Name schon sagt, der parallelen Ausführung von Anweisungen. Ein <PARALLEL> Element hat mindestens ein <TASK> Element als Kindelement. Jeder dieser Tasks (Threads) wird parallel zum Hauptprogramm und allen anderen Tasks ausgeführt. Innerhalb eines <TASK> Elements kann jedoch kein weiteres <PARALLEL> Element auftreten. Die Definition eines Threads innerhalb eines anderen Threads wird nicht unterstützt.

- **Fallunterscheidung**

```
<xsd:element name="SWITCH">
   <xsd:complexType>
      <xsd:sequence>
         <xsd:choice>
            <xsd:element ref="EXPRESSION"/>
            <xsd:element ref="SENSOR"/>
            <xsd:element ref="GETVALUE"/>
         </xsd:choice>
         <xsd:element ref="CASE" minOccurs="1"
                                 maxOccurs="unbounded"/>
      </xsd:sequence>
   </xsd:complexType>
</xsd:element>

<xsd:element name="CASE">
   <xsd:complexType>
      <xsd:sequence>
         <xsd:element ref="VALUE"/>
         <xsd:choice minOccurs="0" maxOccurs="unbounded">
            <xsd:element ref="MOTOR"/>
            <xsd:element ref="SLEEP"/>
            <xsd:element ref="LOOP"/>
            <xsd:element ref="SWITCH"/>
            <xsd:element ref="WAITFOR"/>
            <xsd:element ref="PARALLEL"/>
            <xsd:element ref="SETVALUE"/>
            <xsd:element ref="BREAK"/>
         </xsd:choice>
      </xsd:sequence>
   </xsd:complexType>
</xsd:element>

<xsd:element name="BREAK"/>
```

Zur Definition von Fallunterscheidungen wird `<SWITCH>` verwendet. Das Argument eines `<SWITCH>` Elements kann der Wert eines Ausdrucks (`<EXPRESSION>`), eines Sensors (`<SENSOR>`) oder einer Variable (`<GETVALUE>`) sein. Mit `<CASE>` können nun unterschiedliche Fälle beschrieben werden. Der Wert des Argumentes wird für jeden Fall mit dem dort durch `<VALUE>` definierten Wert verglichen. Stimmen die Werte überein, wird in den jeweiligen Fall verzweigt. Jeder Fall (`<CASE>`) entspricht nach

der Verarbeitung eines *roboXL* Kontrollprogramms einer *if*-Anweisung in *Java*. Es wird also kein *case*-Konstrukt erzeugt, was zur Folge hat, dass jeder Fall mit Übereinstimmung ausgeführt wird, nicht nur der erste. Das *case*-Konstrukt existiert in *leJOS* nicht. Aus diesem Grund wird die Fallunterscheidung mit *if*-Anweisungen realisiert. Man beachte, dass innerhalb eines `<CASE>` Elements ein Element `<BREAK>` erlaubt ist, das dazu verwendet werden kann, eine das `<SWITCH>` Element umgebende Schleife zu beenden. Gibt es keine solche Schleife, so führt die Verwendung des `<BREAK>` Elements zu einen Compilierfehler. Es sei ausdrücklich darauf hingewiesen, dass das `<BREAK>` nicht dazu dient, die Fallunterscheidung zu verlassen.

Alle Elemente, die in einem *roboXL* Kontrollprogramm vorkommen können, sind jetzt beschrieben und wir kommen noch einmal auf die eigentliche Definition des Elements `<PROGRAM>` zurück, die wir am Anfang dieses Kapitels noch ausgespart haben.

```
<xsd:element name="EXOS">
  <xsd:complexType>
      <xsd:choice>
        <xsd:element name="ROBOXL">
          <xsd:complexType>
            <xsd:choice>
              <xsd:element name="PROGRAM">
                <xsd:complexType>
                  <xsd:sequence>
                    <xsd:element ref="VAR" minOccurs="0" maxOccurs="10"/>
                    <xsd:choice minOccurs="0" maxOccurs="unbounded">
                      <xsd:element ref="MOTOR"/>
                      <xsd:element ref="SLEEP"/>
                      <xsd:element ref="LOOP"/>
                      <xsd:element ref="SWITCH"/>
                      <xsd:element ref="WAITFOR"/>
                      <xsd:element ref="PARALLEL"/>
                      <xsd:element ref="SETVALUE"/>
                    </xsd:choice>
                  </xsd:sequence>
                </xsd:complexType>
                <xsd:key name="Var">
                  <xsd:selector xpath="exos:VAR"/>
                  <xsd:field xpath="@name"/>
                </xsd:key>
                <xsd:keyref name="SetVar" refer="Var">
                  <xsd:selector xpath=".//exos:SETVALUE"/>
                  <xsd:field xpath="@name"/>
                </xsd:keyref>
```

```
            <xsd:keyref name="GetVar" refer="Var">
              <xsd:selector xpath=".//exos:GETVALUE"/>
              <xsd:field xpath="@name"/>
            </xsd:keyref>
          </xsd:element>
          <xsd:element name="MONITORING">
          ...
          </xsd:element>


          <xsd:element name="RESPONSE">
          ...
          </xsd:element>
        </xsd:choice>
      </xsd:complexType>
    </xsd:element>
  </xsd:choice>
</xsd:complexType>
</xsd:element>
```

Elemente in *roboXL* Monitoringprogrammen

Dieser Abschnitt befasst sich mit der Syntax von *roboXL* Monitoringprogrammen, d.h. *roboXL* Dokumenten, die den Zugriff auf ein Kontrollprogramm zur Laufzeit definieren.

- **Zugriff auf Sensoren**
  Um auf Sensoren zuzugreifen wird das Element `<SENSOR>` verwendet. Dieses Element ist identisch mit dem Element `<SENSOR>`, das bereits in *roboXL* Kontrollprogrammen verwendet wird. `<SENSOR>` wird in der devicespezifischen Erweiterung von **ExOS.xsd** definiert (s. dazu **RCX.xsd** in Kapitel 2.1.1). Das Element erlaubt es, einen Sensor zu konfigurieren und seinen Wert abzufragen.

- **Konfiguration eines Motors**
  Um einen Motor zu konfigurieren wird das Element `<MOTOR>` verwendet. Dieses Element ist identisch mit dem Element `<MOTOR>`, das bereits in *roboXL* Kontrollprogrammen verwendet wird. `<MOTOR>` wird in der devicespezifischen Erweiterung von **ExOS.xsd** definiert (s. dazu **RCX.xsd** in Kapitel 2.1.1). Ein Motor kann (neu)konfiguriert werden in Bezug auf die Motorenstärke und Bewegungsrichtung.

- **Aktuelle Konfiguration eines Motors abfragen**

```
<xsd:element name="GETMOTOR" type="Motor"/>

<xsd:complexType name="Motor">
  <xsd:attribute name="id" type="MotorId" use="required"/>
</xsd:complexType>
```

Für den Fall, dass man die aktuelle Konfiguration eines Motors benötigt (Motorenstärke, Bewegungsrichtung), steht das Element <GETMOTOR> zur Verfügung. Der complexType Motor legt fest, dass jeder Motor über eine id angesprochen wird. Um die aktuelle Konfiguration eines Motors abzufragen, muss seine id angegeben werden. Der Typ des Attributs id kann durch Definition des Typs MotorId an die Anforderungen des jeweiligen Devices angepasst werden. Für den RCX wurde MotorId in der devicespezifischen Erweiterung von **ExOS.xsd** definiert (s. dazu **RCX.xsd** in Kapitel 2.1.1).

- **Aktuellen Wert einer Variablen abfragen**

```
<xsd:element name="GETVAR">
  <xsd:complexType>
    <xsd:attribute name="id" use="required">
      <xsd:simpleType>
        <xsd:restriction base="xsd:integer">
          <xsd:minInclusive value="1"/>
          <xsd:maxInclusive value="10"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:attribute>
  </xsd:complexType>
</xsd:element>
```

Zum Abfragen des aktuellen Wertes einer Variablen wird das Element <GETVAR> verwendet. Zur Erinnerung: Um eine Variable zur Laufzeit eines *roboXL* Kontrollprogramms durch ein *roboXL* Monitoringprogramm abfragen zu können, muss der Variablen bei der Deklaration im Kontrollprogramm eine id zugewiesen werden, über die diese beim Monitoring eindeutig identifiziert werden kann. Das Attribut id des Elements <GETVAR> entspricht genau dieser id.

- **Monitoring für eine bestimmte Zeitspanne anhalten**
  Das Element <SLEEP> erlaubt es, den Vorgang des Monitorings für eine bestimmte Zeitspanne zu unterbrechen. <SLEEP> entspricht dabei dem Element <SLEEP>, das auch in *roboXL* Kontrollprogrammen verwendet wird. Um die Verwendung dieses

Elements in Monitoringprogrammen nachvollziehen zu können, ein kurzes Beispiel: Angenommen man möchte ein Monitoringprogramm schreiben, das einen beliebigen Motor 5 Sekunden lang laufen lässt. Genau an dieser Stelle ist das `<SLEEP>` Element notwendig. Man benutzt das `<MOTOR>` Element um den Motor zu starten. Danach benutzt man `<SLEEP>` um die Verarbeitung des darauf folgenden `<MOTOR>` Elements, das den Motor wieder stoppt, um genau 5 Sekunden zu verzögern.

Nun, da alle Elemente beschrieben sind, die in einem *roboXL* Monitoringprogramm auftreten können, kommen wir zur vollständigen Definition des Elements `<MONITORING>`:

```
<xsd:element name="EXOS">
  <xsd:complexType>
      <xsd:choice>
        <xsd:element name="ROBOXL">
          <xsd:complexType>
            <xsd:choice>
              <xsd:element name="PROGRAM">
              ...
              </xsd:element>
              <xsd:element name="MONITORING">
                <xsd:complexType>
                  <xsd:sequence>
                    <xsd:element ref="MAPPING"/>
                    <xsd:choice minOccurs="0" maxOccurs="unbounded">
                      <xsd:element ref="SENSOR"/>
                      <xsd:element ref="MOTOR"/>
                      <xsd:element ref="GETMOTOR"/>
                      <xsd:element ref="GETVAR"/>
                      <xsd:element ref="SLEEP"/>
                    </xsd:choice>
                  </xsd:sequence>
                </xsd:complexType>
              </xsd:element>
              <xsd:element name="RESPONSE">
              ...
              </xsd:element>
            </xsd:choice>
          </xsd:complexType>
        </xsd:element>
      </xsd:choice>
    </xsd:complexType>
  </xsd:element>
```

Auf die Verwendung des Elements `<MAPPING>` wurde bereits in Kapitel 2.2 eingegangen. Wie wir dort gesehen haben dient es dazu, ein Mapping zu definieren zwischen den Inhalten devicespezifischer Elemente und den Bytewerten, die zum Device gesendet werden. Nur so kann das *XSLT* Stylesheet, das die Verarbeitung der *roboXL* Monitoringprogramme vornimmt, generisch implemeniert werden. Der Vollständigkeit halber hier die Definition des Elements `<MAPPING>` im *XML Schema* **ExOS.xsd**:

```
<xsd:element name="MAPPING">
  <xsd:complexType>
    <xsd:sequence minOccurs="0" maxOccurs="unbounded">
      <xsd:element ref="PAIR"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:element name="PAIR">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="ITEM" type="xsd:string"/>
      <xsd:element name="BYTEVALUE" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

Elemente in *roboXL* Request- und Responsedokumenten

Dieser Abschnitt befasst sich mit den Elementen in *roboXL* Request- und Responsedokumenten, die wir in Kapitel 2.2 gesehen haben. Im Gegensatz zu den Elementen in *roboXL* Kontroll- bzw. Monitoringprogrammen, die in *XML*-Dokumenten verwendet werden, die von einem Benutzer des Systems selbst erstellt werden, werden die hier erwähnten Elemente nur in *XML*-Dokumenten verwendet, die bei der Verarbeitung von *roboXL* Monitoringprogrammen automatisch generiert werden.
Im *XML Schema* **ExOS.xsd** sind nur die Elemente definiert, die in *roboXL* Requestdokumenten auftreten. Das liegt daran, dass ein *roboXL* Responsedokument durch Manipulation der *DOM*-Struktur aus einem *roboXL* Requestdokument erzeugt wird. Die Validierung von *roboXL* Responsedokumenten ist von daher überflüssig, da das *roboXL* Requestdokument bereits validiert wurde. Aus diesem Grund müssen die Elemente der *roboXL* Responsedokumente nicht im *XML Schema* **ExOS.xsd** definiert werden. Genaugenommen sind dies neben den Elementen, die auch in *roboXL* Requestdokumenten auftreten, nur die Elemente `<RETURN>` und `<RETURNVALUE>`.

Alle Elemente in *roboXL* Requestdokumenten werden aus Elementen in *roboXL* Monitoringprogrammen erzeugt:

- `<SENSORREQUEST>`

```
<xsd:element name="SENSORREQUEST" type="Request"/>
```

  Das Element `<SENSORREQUEST>` wird aus dem Element `<SENSOR>` erzeugt.

- `<MOTORREQUEST>`

```
<xsd:element name="MOTORREQUEST" type="Request"/>
```

  Das Element `<MOTORREQUEST>` wird aus dem Element `<GETMOTOR>` erzeugt.

- `<MOTORSETTING>`

```
<xsd:element name="MOTORSETTING" type="Request"/>
```

  Das Element `<MOTORSETTING>` wird aus dem Element `<MOTOR>` erzeugt.

- `<VARREQUEST>`

```
<xsd:element name="VARREQUEST" type="Request"/>
```

  Das Element `<VARREQUEST>` wird aus dem Element `<GETVAR>` erzeugt.

- `<SLEEPSTATEMENT>`

```
<xsd:element name="SLEEPSTATEMENT" type="Request"/>
```

  Das Element `<SLEEPSTATEMENT>` wird aus dem Element `<SLEEP>` erzeugt.

Wie man sieht, liegt allen Elementen in *roboXL* Requestdokumenten ein complexType Request zugrunde.

```
<xsd:complexType name="Request">
  <xsd:sequence>
    <xsd:element ref="PARAM" minOccurs="0" maxOccurs="unbounded"/>
    <xsd:element ref="REQUEST"/>
  </xsd:sequence>
</xsd:complexType>
```

Wird bei der Definition eines Elements der complexType Request verwendet, so ist festgelegt, dass dieses Element beliebig viele Kindelemente `<PARAM>` haben kann, gefolgt von einem Element `<REQUEST>`.

```
<xsd:element name="REQUEST">
  <xsd:complexType>
    <xsd:sequence minOccurs="0" maxOccurs="unbounded">
      <xsd:element ref="BYTECODE" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

In Kapitel 2.2 wurde beschrieben, wie das `<REQUEST>` Element zum Speichern der Folge von Bytewerten benutzt wird, die aus den Elementen des *roboXL* Monitoringprogramms erzeugt wurden. Dabei haben wir auch gesehen, wie die gespeicherte Folge von Bytewerten durch die Werte ersetzt wird, die das Kommunikationsprotokoll zurückliefert. Das Kommunikationsprotokoll liefert jedoch nur für den Fall der Abfrage eines Sensors, der Abfrage der aktuellen Konfiguration eines Motors oder der Abfrage einer Variablen Werte zurück. Wird hingegen die Konfiguration eines Motors geändert (`<MOTOR>` bzw. `<MOTORSETTING>`) oder soll das Monitoring für eine bestimmte Zeitspanne angehalten werden (`<SLEEP>` bzw. `<SLEEPSTATEMENT>`), so gibt es keine Rückgabewerte. In diesem Fall wird nach Senden der Folge von Bytewerten bei durch Manipulation der *DOM*-Struktur das gesamte Element (`<MOTORSETTING>` bzw. `<SLEEPSTATEMENT>`) entfernt und erscheint damit nicht mehr im *roboXL* Responsedokument.

```
<xsd:element name="PARAM">
  <xsd:complexType>
    <xsd:simpleContent>
        <xsd:extension base="xsd:string">
            <xsd:attribute name="name" type="xsd:string" use="required"/>
        </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
</xsd:element>
```

Parameter werden nun gerade für diejenigen Elemente erzeugt, in denen auch Rückgabewerte des Kommunikationsprotokolls gespeichert werden. Nur so ist es für den Verfasser eines *roboXL* Monitoringprogramms möglich, die zurückgelieferten Resultate denen im Monitoringprogramm definierten Abfragen zuzuordnen.
Eine kurze Erklärung, um die Definition des Elements `<PARAM>` nachvollziehen zu können: Der einfache Inhalt des Elements `<PARAM>` ist vom Typ xsd:string (`<xsd:simpleContent>`) und wird um ein Attribut erweitert. Da das Element `<PARAM>` ein Attribut besitzt, benötigt man `<xsd:complexType>`.

Zum Abschluss noch die vollständige Definition des Elements `<RESPONSE>`:

```
<xsd:element name="EXOS">
  <xsd:complexType>
    <xsd:choice>
      <xsd:element name="ROBOXL">
        <xsd:complexType>
          <xsd:choice>
            <xsd:element name="PROGRAM">
            ...
            </xsd:element>
            <xsd:element name="MONITORING">
            ...
            </xsd:element>
            <xsd:element name="RESPONSE">
              <xsd:complexType>
                <xsd:choice minOccurs="0" maxOccurs="unbounded">
                  <xsd:element ref="SENSORREQUEST"/>
                  <xsd:element ref="MOTORREQUEST"/>
                  <xsd:element ref="MOTORSETTING"/>
                  <xsd:element ref="VARREQUEST"/>
                  <xsd:element ref="SLEEPSTATEMENT"/>
                </xsd:choice>
              </xsd:complexType>
            </xsd:element>
          </xsd:choice>
        </xsd:complexType>
      </xsd:element>
    </xsd:choice>
  </xsd:complexType>
</xsd:element>
```

4.1.2 Devicespezifikationssprache

In den Kapiteln 2.1 und 2.2 wurden bereits alle Elemente der *XML*-basierten Devicespezifikationssprache verwendet. Die Inhalte einer Devicespezifikation werden über die Kindelemente des Elements `<SPECIFICATION>` angegeben:

```
<xsd:element name="SPECIFICATION">
  <xsd:complexType>
    <xsd:all>
      <xsd:element ref="IMPORTS"/>
      <xsd:element ref="VARIABLES"/>
      <xsd:element ref="INIT"/>
      <xsd:element ref="CONTROLMOTOR"/>
      <xsd:element ref="READSENSOR"/>
      <xsd:element ref="METHODS" minOccurs="0"/>
      <xsd:element ref="PROTOCOL"/>
    </xsd:all>
  </xsd:complexType>
</xsd:element>
```

Durch `<xsd:all>` erreicht man, dass die dort angegebenen Elmente alle als Kindelemente von `<SPECIFICATION>` vorkommen müssen, die Reihenfolge in der sie auftreten jedoch beliebig sein darf.

- **Importanweisungen**

  ```
  <xsd:element name="IMPORTS" type="xsd:string"/>
  ```

  Mit `<IMPORTS>` werden die Importanweisungen für die *Java* Klassen angegeben, die für die Implementierung der spezifischen Teile des Kernels benötigt werden.

- **Variablen**

  ```
  <xsd:element name="VARIABLES" type="xsd:string"/>
  ```

  Das Element `<VARIABLES>` dient dazu, neue Variablen anlegen zu können. Die Variablen werden immer als Membervariablen des Kernels angelegt.

- **Initialisierungen**

```
<xsd:element name="INIT" type="xsd:string"/>
```

Der Code für die Initialisierung von Variablen wird durch das Element <INIT> angegeben. Die hier angegebenen Initialisierungen sind später Teil des Konstruktors des Kernel *Java* Programms.

- **Implementierung der Methode *controlMotor()***

```
<xsd:element name="CONTROLMOTOR" type="Method"/>

<xsd:complexType name="Method">
  <xsd:all>
    <xsd:element name="MODIFIER" type="xsd:string"/>
    <xsd:element name="RETURN" type="xsd:string"/>
    <xsd:element name="ARGS"   type="xsd:string"/>
    <xsd:element name="CODE"   type="xsd:string"/>
  </xsd:all>
</xsd:complexType>
```

Zur Angabe der Implementierung der Methode *controlMotor()*, die das devicespezifische Ansteuern eines Motors implementiert, wird das Element <CONTROLMOTOR> verwendet. <CONTROLMOTOR> ist durch den complexType Method definiert. complexType Method definiert die einzelnen Bestandteile einer Methode: die Modifier (<MODIFIER>), den Rückgabewert (<RETURN>), die Argumente (<ARGS>) und den Rumpf der Methode (<CODE>).

- **Implementierung der Methode *readSensor()***

```
<xsd:element name="READSENSOR">
  <xsd:complexType>
    <xsd:complexContent>
      <xsd:restriction base="Method">
        <xsd:all>
          <xsd:element name="ARGS"   type="xsd:string"/>
          <xsd:element name="CODE"   type="xsd:string"/>
        </xsd:all>
```

```
      </xsd:restriction>
    </xsd:complexContent>
  </xsd:complexType>
</xsd:element>
```

Zur Angabe der Implementierung der Methode *readSensor()*, die das devicespezifische Ansteuern eines Sensors implementiert, wird das Element <READSENSOR> verwendet. Analog zu <CONTROLMOTOR> wird auch hier der complexType Method verwendet. Der Rückgabewert der Methode *readSensor()* ist jedoch immer vom Typ *Object*, um zu ermöglichen, dass ein Sensor jeden beliebigen Typ zurückgeben kann. Aus diesem Grund werden von complexType Method nur die Elemente <ARGS> und <CODE> geerbt. *XML Schema* bietet dazu eine spezielle Vererbungstechnik (<xsd:restriction>) an. Schränkt man einen complexType ein, so benötigt man in *XML Schema* <xsd:complexContent>.

- **Definition von Hilfsmethoden**

```
<xsd:element name="METHODS">
  <xsd:complexType>
    <xsd:sequence minOccurs="0" maxOccurs="unbounded">
      <xsd:element ref="METHOD"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:element name="METHOD">
  <xsd:complexType>
    <xsd:complexContent>
      <xsd:extension base="Method">
        <xsd:attribute name="name" type="xsd:string"
                       use="required"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
</xsd:element>
```

Das Element <METHODS>, besser gesagt dessen Kindelement <METHOD>, kann dazu verwendet werden eigene Methoden zu definieren. Die Verwendung dieses Elements ist fakultativ. Für jede definierte Methode muss zu den durch complexType Method definierten Bestandteilen einer Methode (Modifier, Rückgabewert, Argumente, Rumpf)

auch ein Name vergeben werden. Zu diesem Zweck wird der complexType Method um ein Attribut name erweitert.

- **Implementierung des Kommunikationsprotokolls**

```
<xsd:element name="PROTOCOL">
  <xsd:complexType>
    <xsd:all>
      <xsd:element ref="IMPORTS"/>
      <xsd:element ref="VARIABLES"/>
      <xsd:element ref="INIT"/>
      <xsd:element ref="PROCESSBYTECODE"/>
    </xsd:all>
  </xsd:complexType>
</xsd:element>

<xsd:element name="PROCESSBYTECODE" type="xsd:string"/>
```

Die Implementierung des Kommunikationsprotokolls wird durch das Element <PROTOCOL> angegeben. Wie wir in Kapitel 2.2 gesehen haben, ist das Protokoll in eine eigene *Java*-Klasse gekapselt. Deshalb können für das Protokoll eigene Importanweisungen, Variablen und Initialisierungen angegeben werden. Die Elemente <IMPORTS>, <VARIABLES> und <INIT> sind dabei dabei identisch mit den weiter oben in diesem Kapitel gesehenen Elementen. Die eigentliche Implementierung des Kommuniaktionsprotokolls wird durch das Element <PROCESSBYTECODE> angegeben.

Index

Literaturverzeichnis

[1] Apache Axis Framework. *Apache Web Services Project* `http://ws.apache.org/axis`, 2005.

[2] Apache Jakarta Tomcat. *Apache Jakarta Project* `http://jakarta.apache.org/tomcat`, 2005.

[3] Appel A.W., *Modern Compiler Implementation in Java*. Cambridge University Press, New York, Cambridge, 1998.

[4] Attali I., Courbis C., Degenne P., Fau A., Fillon J., Held C., Parigot D., and Pasquier C., Aspect and XML-oriented Semantic Framework Generator: SmartTools. In Proceedings of 2nd Workshop on Language Descriptions, Tools and Applications LDTA'02. Electronic Notes in Computer Science, Elsevier, 2002.

[5] Armstrong E., Ball J., Bodoff S., Carson D.B., Evans I., Fisher M., Fordin S., Green D., Haase K., Jendrock E., *The Java Web Services Tutorial 1.3*. `java.sun.com/webservices/docs/1.3/tutorial/doc/`, 2003.

[6] Armstrong E., Ball J., Bodoff S., Carson D.B., Evans I., Green D., Haase K., Jendrock E., *The J2EE 1.4 Tutorial*. `http://java.sun.com/j2ee/1.4/docs/tutorial-update2/doc/index.html`, 2004.

[7] Bagnall B. (webmaster). *lejOS - Java for the RCX*. Open source project at `http://lejos.sourceforge.net`, 2005.

[8] Bray T., Hollander D., Layman A., Namespaces in *XML*, W3C Recommodation, 14 January 1999. `http://www.w3.org/XML/Core/#Publications`.

[9] Cleaveland J.C., *Program Generators with XML and Java*. Prentice Hall, 2001.

[10] Costello R.L., *XML Schema Tutorial*. `http://www.xfront.com`, 2003.

[11] Costello R.L., *XSLT Tutorial*. `http://www.xfront.com`, 2003.

[12] Fuzellier M. (webmaster). *W3C - World Wide Web Consortium*. `http://www.w3.org`, 2005.

[13] Inject/J Team. *Inject/J - Source Code Transformation at your Fingertips*. `http://injectj.fzi.de`, 2005.

[14] Jansen Marc. *Integrating Smart Devices into Java Applications*. Dissertation (in Vorbereitung). Universität Duisburg-Essen, 2004.

[15] Kiczales G., Lamping J., Mendhekar A., Maeda C., Videira Lopes C., Loingtier J.-M. and Irwin J., Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Springer-Verlag LNCS 1241, 1997.

[16] Lego Company. Lego Mindstorms. `http://mindstorms.lego.com`, 2005.

[17] Makatchev M. and S. K. Tso. Human-Robot Interface Using Agents Communicating in an XML-Based Markup Language. In *Proceedings of the 2000 IEEE International Workshop on Robot and Human Interactive Communication ROMAN2000*, pages 270-275, Osaka, Japan, September 2000. `http://www.roboml.org`, 2003.

[18] Münz Stefan. *SelfHTML Dokumentation*. Kapitel "Einführung in XML". `http://de.selfhtml.org/xml/intro.htm`, 2005.

[19] Phantom II, LLC. Leonardo Interface. `http://www.phantom2.com`, 2005.

[20] Sato J., *Jin Sato's Lego Mindstorms, The Master's Technique*. No Starch Press, 2002.

[21] Systronix Corporation. JCX - Java Control System `http://jcx.systronix.com`, 2005.

[22] Wikipedia. *Die freie Enzyklopädie*. XML Schema. `http://de.wikipedia.org/wiki/XML_Schema`, 2005.

[23] Wikipedia. *Die freie Enzyklopädie*. XSLT. `http://de.wikipedia.org/wiki/XSLT`, 2005.

[24] Zeller A., *Aspektorientierte Programmierung*. Vorlesung "Softwaretechnik" (WS 2003/04). `http://www.st.cs.uni-sb.de/edu/se/aop.pdf`, 2003.