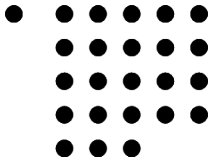

Fachhochschule Köln
University of Applied Sciences

Fakultät 07
Informations-, Medien- und Elektrotechnik
Institut für Nachrichtentechnik
Studiengang Master of Science Technische Informatik

Masterarbeit

Spezifikation einer Robotic Task Definition Language (RTDL)

Student:	**Marcus Teske**
Matrikelnummer:	**1104 1133**
Referent:	**Prof. Dr. phil. Gregor Büchel**
Koreferent:	**Prof. Dr.-Ing. Georg Hartung**

Abgabedatum: **17. Januar 2011**

Hiermit versichere ich, dass ich die Masterarbeit selbständig angefertigt und keine anderen als die angegebenen und bei Zitaten kenntlich gemachten Quellen und Hilfsmittel benutzt habe.

Marcus Teske

Inhaltsverzeichnis

Abbildungsverzeichnis

Abstract

Roboter finden in der heutigen Zeit bereits in vielen Bereichen ihre Einsatzgebiete. Zu diesen Bereichen gehören die Industrie, wo Roboter hauptsächlich für filigrane, zu monotone oder gefährliche Arbeiten Anwendung finden. Ein weiterer wichtiger Bereich ist die militärische Aufklärung und das Aufspüren von Personen in schwer zugänglichen Gebieten, wie nach einer Umweltkatastrophe. All diese Roboter haben gemeinsam, dass sie bestimmte Aufgaben ausführen, für welche sie entwickelt wurden.

Das Ziel dieser Arbeit ist die Definition einer allgemeinen Aufgaben-Beschreibungssprache, die nicht an einen einzelnen Roboter gebunden ist, sondern für alle möglichen Arten von Robotern einsetzbar ist. Dieses Ziel soll mit Hilfe der Robotic Task Definition Language erreicht werden.

Die Robotic Task Definition Language (RTDL) ist eine Sprache, die an der Fachhochschule Köln im Rahmen des Instituts-übergreifenden Projektes “Verteilte Mobile Applikationen (VMA)” entwickelt wird. Die Aufgabe der RTDL besteht darin, eine allgemein gültige Aufgabenbeschreibungssprache für unterschiedlichste Roboter zu definieren. Mit einer solchen Aufgabenbeschreibungssprache ist es möglich, die Steuerungssoftware verschiedener Roboter in einer einheitlichen Sprache zu beschreiben und in einem weiteren Schritt auch umzusetzen.

1 Einleitung

Die Spezifikation einer Robotic Task Definition Language (RTDL) schafft eine Abstraktionsebene in der Steuerungssoftware von Robotern. Innerhalb dieser Ebene werden Aufgaben (Tasks) in einer abstrakten Form als Hierarchie von primitiven Roboteraktionen und untergeordneten Tasks beschrieben.

Die Motivation für die Spezifikation einer RTDL liegt in dem Wunsch nach der Fähigkeit, Tasks unabhängig von einem bestimmten Roboter zu definierten. Erweitert man die Steuerungssoftware von verschiedenen Robotern um die Fähigkeit, eine Taskdefinition die in der RTDL formuliert worden ist, zu interpretieren, so schafft man eine Wiederverwendbarkeit von Taskdefinitionen, die es ermöglichen, definierte Tasks von einem konkreten Roboter auf andere Roboter zu übertragen.

Im Rahmen dieser Arbeit wird eine Sprache für die Beschreibung von Robotertasks entwickelt. Hierzu werden zunächst existierende Sprachen zur Taskbeschreibung analysiert. Aus den Ergebnissen dieser Analyse wird eine neue Sprache erzeugt. Zusätzlich wird eine einfache Simulationsumgebung für Roboter entwickelt, die unter anderem als Testumgebung für die RTDL dient.

Diese Arbeit beginnt mit einigen Grundlagenkapiteln in denen das Umfeld dieser-Arbeit eingeführt wird. In Kapitel 2 dieser Arbeit werden zunächst Roboter von einem allgemeinen Standpunkt aus betrachtet. Das darauf folgende Kapitel 3 liefert eine Übersicht, über bereits existierende Roboter der Fakultät für Informations-, Medien- und Elektrotechnik der Fachhochschule Köln, wo diese Arbeit entstanden ist. In Kapitel 4 wird die Joint Architecture for Unmanned Systems (JAUS) eingeführt. Hierbei handelt es sich um einen Standard für die Beschreibung von Roboterarchitekturen.

Nachdem die Grundlagen für diese Arbeit in den ersten Kapiteln gelegt wurden, wird ein Fokus auf Tasksprachen gelegt. In Kapitel 5 werden bereits existierende Sprachen zur Taskdefinition betrachtet. Diese Sprachen dienen als Grundlagen für Kapitel 6, wo die RTDL beschrieben wird. In Kapitel 7 wird eine Simulationsumgebung für die beispielhafte Implementierung und Testmöglichkeit der RTDL beschrieben. Zum Abschluss dieser Arbeit wird in Kapitel 8 ein Fazit gezogen.

2 Roboter

2.1 Einleitung

Der Begriff Roboter stammt von dem tschechischen Schriftsteller Karl Čapek, der diesen Begriff für sein 1921 erschienenes Bühnenstück "R.U.R. Rossum's universal robots" vom tschechischen Wort für Zwangsarbeit "robota" ableitete.

Da Roboter in teils sehr unterschiedlichen Bereichen eingesetzt werden, lassen sie sich auch in mehrere Kategorien einteilen. Zu oberst können sie in stationäre und mobile Roboter unterschieden werden.

2.2 Stationäre Roboter

Stationäre Roboter besitzen keine Möglichkeit, sich in ihrer Umgebung zu bewegen. Zu stationären Robotern lassen sich beispielsweise Industrieroboter zählen. Einem Industrieroboter werdenoftmalsWerkstückevorgesetzt,sei es manuell oderautomatisiert durch ein Förderband. Der Industrieroboter nutzt dann einen oder mehrere mechanische Arme, an denen bestimmte Werkzeuge wie beispielsweise Schweißgeräte oder Greifer befestigt sind, um das vor ihm liegende Werkstück zu bearbeiten.

Die wichtigsten Vorzüge eines stationären Roboters sind:

- Roboter ermüden nicht und liefern stets eine gleich bleibend hohe Qualität.
- Roboter können sehr präzise arbeiten, oftmals präziser als Menschen.
- Roboter können Tätigkeiten ausführen, die für Menschen zu gefährlich sind.

2.3 Mobile Roboter

Mobile Roboter besitzen die Fähigkeit, sich in ihrer Umgebung zu bewegen. Dieser Umstand stellt einen mobilen Roboter jedoch vor weitere Herausforderungen: So muss ein mobiler Roboter sich in seiner Umgebung orientieren können. Dies setzt voraus dass er seine Umgebung über Sensoren erfassen und intern abbilden kann. Des weiteren muss ein mobiler Roboter bei der Bewegung durch seine Umgebung auf Hindernisse reagieren, das heißt diese erfassen, erkennen und ausweichen können.

Bei mobilen Robotern ist eine weitere Unterteilung nach ferngesteuerten und autonomen Systemen sinnvoll.

2.3.1 Mobile ferngesteuerte Roboter

Ein mobiler ferngesteuerter Roboter besitzt die Fähigkeit, sich in seiner Umgebung zu bewegen und, je nach Art der zusätzlich installierten Werkzeuge, diese auch manipulieren zu können. Wie der Name jedoch bereits vermuten lässt, wird ein solcher Roboter dabei von einer entfernten Instanz über einen Kommunikationskanal ferngesteuert. Alle Entscheidungen werden von der entfernten Instanz getroffen, wobei es sich in den häufigsten Fällen um einen Menschen mit einer Fernsteuerung handelt. Diese Entscheidungen werden in Form von Anweisungen über den Kommunikationskanal an den Roboter übertragen, der diese Anweisungen dann als Aktionen umsetzt.

Bei ferngesteuerten Robotern benötigt die entfernte Instanz wichtige Informationen über die Umgebung, in welcher der ferngesteuerte Roboter eingesetzt wird. Es ist häufig so, dass die entfernte InstanzVideodaten von am Roboter angebrachten Kameras verwendet, um die aktuelle Situation des ferngesteuerten Roboters erfassen und Entscheidungen über weitere Aktionen treffen zu können.Alternativ kann sich die entfernte Instanz an einer Position befinden, von der aus sie die relevanten Teile der Umgebung des Roboters überblicken kann.

Ein Beispiel für einen mobilen ferngesteuerten Roboter findet man bei Firmen, die Abwasserrohre sanieren. Hierbei wird ein kleiner fahrender Roboter mit einer Kamera in ein Kanalrohr abgelassen. Der Mitarbeiter, der den Roboter steuert, kann mit Hilfe der Kamera an dem Roboter schadhafte Stellen in einem Abwasserrohr aufspüren und gegebenenfalls reparieren.

2.3.2 Mobile autonome Roboter

Ein mobiler autonomer Roboter, auch mobiles autonomes System genannt, ist ein Roboter, der in der Lage ist, selbstständig in seiner Umgebung zu navigieren und diese zu manipulieren. Er benötigt keine entfernte Instanz, welche ihn direkt fernsteuert. Auch wenn ein autonomer Roboter selbstständig agieren kann, ist es dennoch oftmals sinnvoll einen Kommunikationskanal zu einer entfernten Instanz zu verwenden. Ein solcher Kommunikationskanal wird häufig verwendet um Informationen

über bestimmte Situationen zu versenden. Des weiteren kann man auf diesem Wege eine Art Failsafe bereitstellen, über welchen der Roboter in bestimmten kritischen Situationen deaktiviert oder zwangsweise ferngesteuert werden kann.

Ein interessantes Einsatzgebiet für mobile autonome Roboter ist der kooperative Einsatz mehrerer Roboter zur Erfüllung einer gemeinsamen Mission. Ein Beispiel für kooperierende mobile autonome Systeme ist der RoboCup Wettbewerb [Rob11], bei dem mehrere Roboter als Team gegen eine gegnerisches Roboter-Team ein Fußballspiel bestreiten. Das ehrgeizige Ziel dieses Wettbewerbs ist es, bis 2050 die Roboter soweit zu entwickeln, dass diese in der Lage sind, den dann amtierenden Weltmeister nach den gültigen Fußballregeln zu besiegen.

Innerhalb dieser Arbeit wird der Fokus auf mobile autonome Roboter gelegt, da diese als einzige von einer Task-Beschreibungssprache profitieren.

2.4 Hardware

Zur Hardware eines Roboters zählen alle Komponenten, die physikalisch vorhanden sind und dem Roboter zur Ausübung seiner Fähigkeiten dienen. Man unterscheidet hierbei zwischen Sensoren, Aktoren und Kommunikationsgeräten.

2.4.1 Sensoren

Sensoren ermöglichen es einem Roboter, die Umwelt, in der er sich befindet, zu erfassen. Die Sensoren erzeugen entweder permanent oder auf Anfrage Sensordaten, die auf dem Roboter weiterverarbeitetwerden müssen.Beider Verarbeitungder Sensordatenwerdenrelevante Informationen aus diesen Daten in eine symbolische Form übersetzt, die einem internen Datenformat des Robotersentsprechen. DerRoboter nutzt diese Informationen dann, um bestimmte Gegebenheiten seiner Umgebung darstellen zu können und dienen der Planung von zukünftigen Aktionen oder als Feedback zu aktuell durchgeführten Aktionen.

Reelle Sensoren sind fast immer fehlerbehaftet, was dazu führt, dass Entscheidungen, die auf Basis solcher Daten getroffen werden, ebenfalls fehlerbehaftet oder gar völlig falsch sind. Bei reellen Robotern gehören diese Aspekte zum Umgang mit unsicherem Wissen. Hierzu gehört sowohl die Einschätzung der Abweichung von Sensordaten zum tatsächlichen Ereignis als auch die Wahrscheinlichkeit, dass ein Sensorwert

gänzlich verfälscht ist. Aus diesem Grund werden Entscheidungen eines Roboters oftmals nicht allein auf Basis eines einzigen Sensorwertes getroffen. Statt dessen werden mehrere Sensorwerte von unterschiedlichen Sensoren ausgewertet und deren Informationen zusammengefasst. Diesen Vorgang nennt man Merging (nach dem englischen Wort für zusammenfassen: to merge sth.). Auf diese Art lassen sich Fehler, die in bestimmten Situationen bei bestimmten Sensoren auftreten, kompensieren.

Üblicherweise verändern Sensoren die Umgebung des Roboters nicht, sondern dienen lediglich der Erfassung von Daten.

Im folgenden ist eine Liste von ausgewählten Sensoren und der Art ihrer erfassten Daten dargestellt:

- Ein **Kompass** kann die aktuelle Orientierung des Roboters anhand des magnetischen Nordpols der Erde bestimmen.
- Ein **Ultraschallsensor** kann das Vorhandensein und gegebenenfalls den Abstand eines Objektes in der Nähe des Roboters ermitteln.
- Ein **Radenkoder** kann die über Rollen oder Raupen zurückgelegte Strecke des Roboters über die Anzahl der Radumdrehungen ermitteln.
- Ein **RFID-Reader** kann RFID-Tags in seiner unmittelbaren Umgebung auslesen, auf deren Basis die aktuelle Position des Roboters ermittelt werden kann.
- Eine **Kamera** kann Umgebungsbilder erfassen, auf deren Basis eine Positionsbestimmung und Hinderniserkennung möglich ist.
- Ein auf den Boden gerichteter **optischer Mousesensor** kann die zurückgelegte Strecke eines Roboters ermitteln.
- Ein **GPS-Empfänger** kann durch Triangulation von Signalen mehrerer GPS-Satelliten zur Positionsbestimmung genutzt werden.

2.4.2 Aktoren

Aktoren sind Hardware-Geräte, die es dem Roboter ermöglichen sich in seiner Umgebung fort zu bewegen oder Objekte in der Umgebung zu manipulieren. Oftmals werden Aktoren unter sensorischer Überwachung betrieben, damit der Roboter eine Rückmeldung über die Manipulation durch die Aktoren erhält.

- **Fortbewegungsaktoren:** Hierunter fallen alle Aktoren, die es dem Roboter ermöglichen seine Position und Ausrichtung zu verändern. In den meisten Fällen handelt es sich hierbei um Räder oder Raupenketten zur Fortbewegung auf dem Boden, Schrauben für die Fortbewegung auf oder unter dem Wasser und Propeller oder Düsen für die Bewegung in der Luft.
- **Objektmanipulatoren:** Unter diesen Begriff fallen alle Aktoren die Objekte in der Umgebung manipulieren können. Diese Aktoren können vielfältiger Natur sein: Beispielsweise gibt es Greifarme, die Objekte greifen, verdrehen, ziehen, drücken und quetschen können. Aber auch Werkzeuge wie Nietenpistolen oder Schweißgeräte sind denkbar. Diese sind oftmals an einen beweglichen Arm gekoppelt. Beim RoboCup dienen beispielsweise die Beine der Roboter, welche primär für die Fortbewegung gedacht sind, zusätzlich als Abschusseinrichtung für den Fußball.

2.4.3 Kommunikationsgeräte

Wie bei der Vorstellung der unterschiedlichen Roboterarten bereits angeführt, besitzen Roboter oftmals einen Kommunikationskanal zu einer entfernten Instanz. Auf welche Art und Weise dieser Kanal zustande kommt ist hierbei abhängig von der verwendeten Technologie, die für die Kommunikation eingesetzt wird. Eine solche Technologie ist beispielsweise WLAN. WLAN steht für Wireless Local Area Network und ist in den IEEE Drafts 802.11 standardisiert. Auf WLAN wird dann häufig das IP-Protokoll (Internet Protocol) und das TCP- (Transmission Control Protocol) oder UDP-Protokoll (User Datagram Protokoll) verwendet, um die Daten vom Roboter zur entfernten Instanz und zurück zu übertragen. Es sind jedoch auch andere Technologien, wie zum Beispiel proprietäre (nicht-standardisierte) Funkgeräte, zur Realisierung eines Kommunikationskanal denkbar.

2.5 Software

Neben der Hardware spielt auch die Software bei einem Roboter eine entscheidende Rolle. Bei einem Roboter ist die Software ähnlich aufgebaut wie bei einem PC.

1. Auf der untersten Ebene befindet sich die Hardware.
2. Auf der Hardware läuft ein Betriebssystem

   (a) Der zentrale Teil eines Betriebssystems ist der Betriebssystemkern, auch Kernel genannt.

   (b) An den Kernel angebunden sind auf der untersten Ebene die Treiber, welche einen Zugriff auf die Hardware ermöglichen.

   (c) Auf der obersten Ebene des Betriebssystems befindet sich ein API (Application Programming Interface), mit dessen Hilfe Systemtools und Anwendungsprogramme mit dem Betriebssystem interagieren können.

3. Die Systemtools dienen der Verwaltung des Betriebssystems.

4. Anwendungsprogramme bilden dann die eigentlichen Dienste aus, die ein Roboter erbringen soll.

Abbildung 3 auf Seite 12 verdeutlicht den Zusammenhang von Hardware und Software nochmals an Hand des Roboters Surveyor.

Ein entscheidendes Anwendungsprogramm bei einem Roboter ist die Steuerungssoftware. Bei einem mobilen autonomen System legt sie die Fähigkeiten und das Verhalten des Roboters fest.

Die Steuerungssoftware nutzt die Daten der Sensoren um die Aktoren zur steuern. Des weiteren kann sie unter Verwendung der Kommunikationsgeräte mit anderen Robotern und Systemen in der Umgebung kommunizieren. All dies dient dem Roboter als Grundlage um bestimmte Aufgaben selbstständig erfüllen zu können. Oftmals sind die Fähigkeiten eines Roboters, also dessen Funktionalitäten, in Form von Funktionsaufrufen in der Steuerungssoftware implementiert.

3 Roboterplattformen

3.1 CokeRob

Der CokeRob [Cok10] gehört zur Kategorie der stationären Roboter. Er besteht aus einer Tisch-großen Fläche, die von einer externen Kamera überwacht wird, einem Steuerungsrechner und einem aus Lego-Steinen aufgebauten mobilen Roboter. Der mobile Roboter kann sich lediglich auf der kleinen Fläche frei und autonom bewegen und da die Fläche ihrerseits nicht mobil ist, ist das Gesamtsystem des CokeRobs eher als stationärer Roboter zu betrachten. Der einzige Sensor des CokeRob ist die Kamera. Die Daten der Kamera werden in dem externen Steuerungsrechner mit Bilderkennungssoftware ausgewertet und per drahtloser Kommunikationsschnittstelle an den mobilen Roboter gesendet.

Legt man nun eine Münze auf diese Fläche, erkennt der CokeRob diese und positioniert seinen Greifarm darüber. Dann lässt er den Arm herab und schiebt die Münze an eine definierte Position. Dort wird die Münze von der Kamera erfasst und identifiziert. Dann schiebt der CokeRob die Münze in einen Kassenbereich. Im nächsten Schritt betätigt der CokeRob einen Schalter, der eine Dose Cola freigibt.

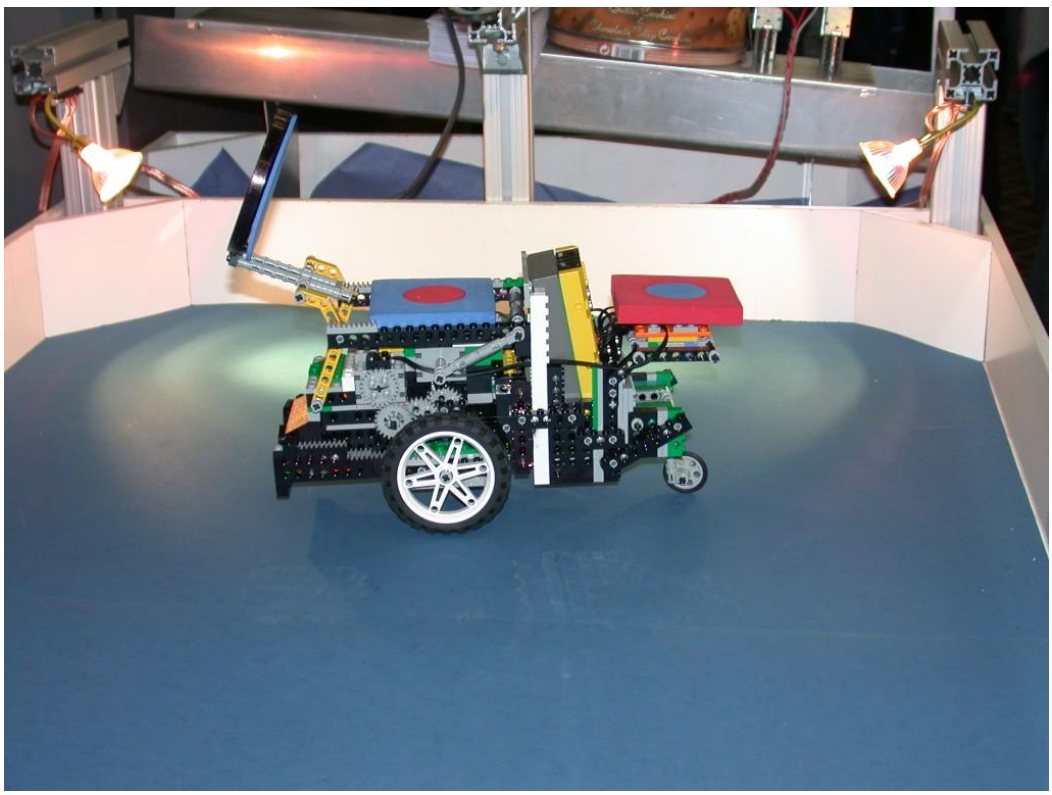

Abbildung 1: CokeRob - Quelle: [Cok10]

3.2 Surveyor

3.2.1 Übersicht

Der Surveyor ist derzeit die meist genutzte Entwicklungsplattform innerhalb der Fakultät 07 der FH Köln. Es handelt sich dabei um einen autonomen mobilen Roboter, dessen primäres Ziel die Wegfindung in einer Flurwelt ist. Der Aufbau des Surveyors wird im folgenden stichpunktartig beschrieben:

- Die zentrale Datenverarbeitungseinheit ist ein Blackfin Board.
- Aktoren
  - Zwei differential angesteuerte Räder vorne und zwei nachlaufende Räder hinten, verbunden durch eine Gummikette.
- Sensoren
  - RFID-Reader für die Erkennung der auf dem Boden verklebten RFID-Tags als Positionsmarker.
  - Kompass für die Bestimmung der Ausrichtung der Plattform.
  - Ultraschallsensor für die Hinderniserkennung und Kollisionsvermeidung.
  - Laseremitter gekoppelt mit einer Kamera zur Abstandsmessung.
- Kommunikationsschnittstellen
  - Zwei serielle Schnittstellen, wobei eine davon an einen WLAN-Adapter gekoppelt ist, wodurch eine drahtlose serielle Kommunikation ermöglicht wird.

Wie bereits in Kapitel 2.4.1 erwähnt, sind Sensoren oftmals fehlerbehaftet. Beim Surveyor führt das zu den folgenden Problemen:

- Der Surveyor kann seine Position nicht immer genau bestimmen, da er lediglich weiß, welches RFID-Tag er als letztes gelesen hat. Es kann also nur vermutet werden dass der Surveyor sich nahe des zuletzt gelesenen Tags befindet. Dieser Fehler ist nicht direkt auf einen Sensorenfehler zurückzuführen sondern ist durch das Design der Positionsbestimmung mit Hilfe von RFID-Tags bedingt.

- Der Kompass liefert eine maximale Genauigkeit von 1/10 Grad, jedoch wird der Messwert durch Stahlträger und andere metallische Körper in seiner Umgebung stark verfälscht. Dies führt dazu dass der Kompass in einigen Regionen die Ausrichtung sehr gut und anderen sehr unzuverlässig ermittelt.
- Dem Ultraschallsensor ist derzeit nicht im Einsatz. Deswegen liegen derzeit noch keine Erkenntnisse über mögliche Fehlerquellen vor.

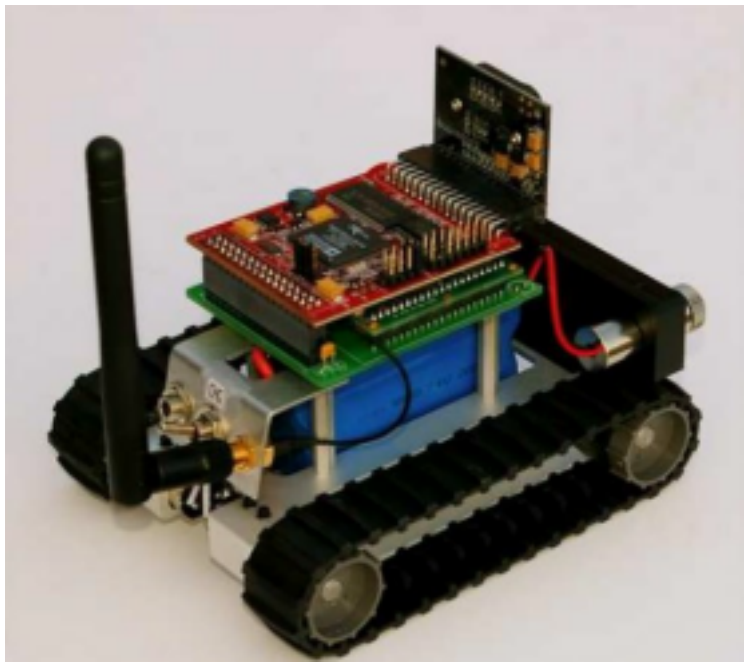

Abbildung 2: Surveyor - Quelle: [Erd09]

3.2.2 Surveyor Architektur

Die Surveyor Architektur besteht im Groben aus zwei Bestandteilen. Der erste Bestandteil ist der Surveyor Roboter bestehend aus der Hardware, wie sie in Kapitel 3.2.1 beschrieben ist, und der Steuerungssoftware. Als Betriebssystem wird uClinux verwendet, eine Linuxvariante für eingebettete Systeme. Die gesamte Software ist auf einem Flash-Speicher abgelegt. Innerhalb des uClinux existieren Treiber für die Zugriffe auf die Hardwaregeräte. Des weiteren bietet das Betriebssystem eine Eingabeaufforderung, die sogenannte Shell an, mit welcher über eine entfernte Telnet-Verbindung Kommandos auf dem Surveyor Roboter ausgeführt werden können. Auf dieser Shell laufen eine Reihe von Programmen, die vordefinierte Aktionen auf dem Roboter ausführen. Diese Programme sind vergleichbar mit vordefinierten Tasks,

die intern aus Funktionsaufrufen bestehen, die ihrerseits Kommandos über die Treiber an die Hardware absetzen. Der Surveyor besitzt unter anderem zwei serielle Schnittstellen, von der eine an einen WLAN-Adapter angeschlossen ist.

Die zweite Komponente ist der sogenannte Proxy. Der Proxy ist ein normaler PC der über eine WLAN-Schnittstelle mit dem Surveyor Roboter kommuniziert. Auf dem Proxy befindet sich zum einen die Entwicklungsumgebung für den Surveyor, mit dessen Hilfe ein verändertes uClinux-Image erzeugt und in den Flash-Speicher vom Surveyor geschrieben werden kann.

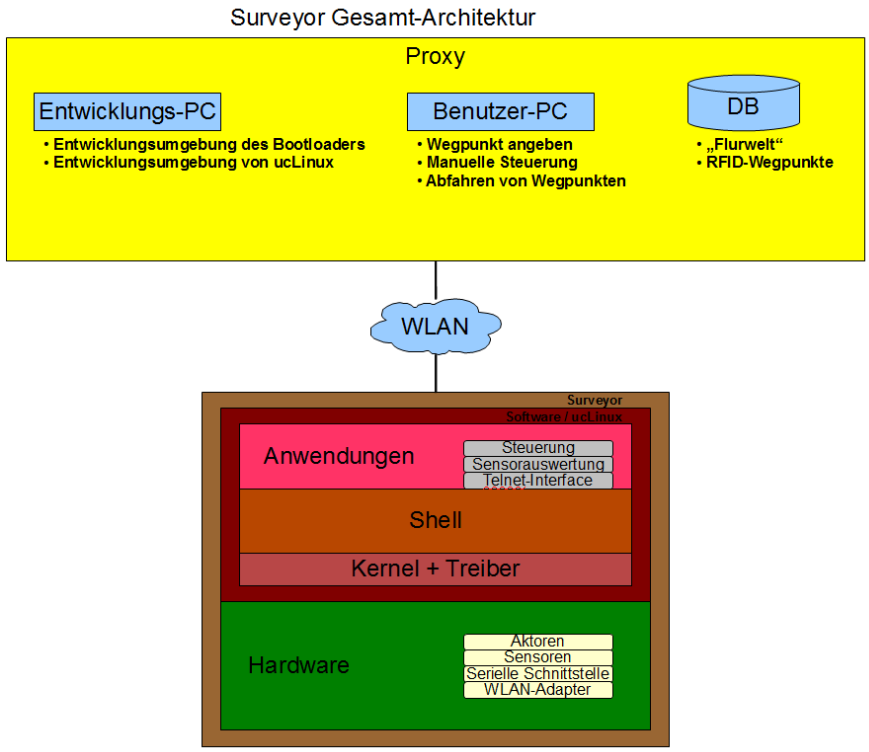

Abbildung 3: Surveyor Gesamtarchitektur

Zum anderen besteht der Proxy aus dem sogenannten Benutzer-PC. Der Benutzer-PC bietet eine Schnittstelle an, um auf dem Surveyor über den Kommunikationskanal eine Task auszuführen. Eine solche Task ist zum Beispiel das Anfahren eines bestimmten RFID-Tags auf einer Teststrecke. Der Benutzer-PC ist zudem an eine Flurwelt-Datenbank angeschlossen. In dieser Datenbank befinden sich die Positionsinformationen der verklebten RFID-Tags. Über den Kommunikationskanal erhält der Surveyor Zugriff auf diese Datenbank und ist in der Lage auf Basis dieser Daten einen Pfad von einem Start zu einem Endpunkt zu erzeugen und diesem zu folgen.

3.2.3 Capabilities des Surveyor Roboters (Services)

Im folgenden werden die wichtigsten Fähigkeiten (Capabilities, siehe Kapitel 6.5) des Surveyor Roboters aufgelistet, die durch die Kombination von Steuerungssoftware, Sensoren und Aktoren in Form von Funktionsaufrufen bereit gestellt werden.

Motorsteuerung (Aktoren zur Fortbewegung) Der Surveyor ist in der Lage sich auf einer ebenen Fläche fahrend fort zu bewegen. Hierzu sind die folgenden Funktionen implementiert:

- Geradeausfahrt: `motor_forward();`
- anhalten: `motor_stop();`
- nach rechts drehen (90°): `motor_turnright();`
- nach links drehen (90°): `motor_turnleft();`
- beliebig drehen: `motor_turn(int direction);`

RFID-Funktionen (Sensoren zur Positionsbestimmung)) Einen Großteil der Navigation erreicht der Surveyor durch das Auslesen von RFID-Tags, die auf dem Boden der Teststrecke verklebt sein müssen. RFID-Tags (Radio Frequency Identification) sind passive Transponder, die, wenn sie durch die Radiowellen eines RFID-Lesegeräts angeregt werden, eine Information aussenden, die dann von dem Lesegerät ausgelesen werden kann. Der Surveyor besitzt ein solches Lesegerät und stellt die folgenden Funktionen zum Umgang mit RFID-Tags zur Verfügung:

- Tag auswählen: `rfid_selecttag();`
- bei Tag authentifizieren: `rfid_authenticate();`
- Tag-Block 1 auslesen: `rfid_readblock1();`
- Tag auslesen: `rfid_gettag();`
- Block x in Tag t beschreiben: `rfid_writeblock(int x, rfidtag t);`

Kompass-Funktionen (Sensor zur Bestimmung der Ausrichtung) Der zweite wichtige Bestandteil der Navigationsfähigkeit des Surveyors, wird durch einen

Kompass erbracht. Der Kompass erlaubt es dem Surveyor seine Ausrichtung in Bezug auf eine bekannte Umgebung zu ermitteln. Der Surveyor unterstützt die folgenden Funktionen im Umgang mit dem Kompass:

- auf eine bestimmte Gradzahl drehen: `fahren_drehenzu(int gradzahl);`
- um eine bestimmte Gradzahl drehen: `fahren_drehenum(int gradzahl);`

Höhere Funktionen Der Surveyor verknüpft einige seiner Basis-Fähigkeiten und bietet diese gekapselt als eine höhere Funktion an. So bildet eine Kombination aus RFID-, Kompass- und Fahr-Funktionen gekoppelt mit einem Modell der Umgebung in Form einer Datenbank die Grundlage für autonome Fahrten innerhalb der Umgebung. Durch die Ermittlung der aktuellen Position durch ein gelesenes RFID-Tag und der Ausrichtung durch den Kompass, kann mit Hilfe des Umgebungsmodell in der Datenbank ein Vektor zu einem Ziel-Tag ermittelt werden. Dieser Vektor wird dann mit Hilfe der Fahrfunktionen abgefahren. Während dieser Fahrt ist es wahrscheinlich, dass der Surveyor andere RFID-Tags passiert. In dem Fall liest er die Tags aus und prüft, ob sie mit dem berechneten Vektor der autonomen Fahrt übereinstimmen und korrigiert seinen Kurs gegebenenfalls automatisch. Die autonome Fahrt kann durch Aufruf der folgenden Funktion gestartet werden:

- autonome Fahrt zu einer Koordinate: `fahren_drivetoxy(int x, int y);`

3.3 RobotTI

Man kann den RobotTI als den großen Bruder des Surveyor bezeichnen. Er ist durch seine Dimensionen in der Lage, ein Notebook als lokalen Verarbeitungsrechner an Bord zu transportieren. Hierdurch kann der RobotTI auf ein enormes Datenverarbeitungspotential zurückgreifen, wie sie höher integrierte Roboterplattformen wie der Surveyor nicht besitzen.

Ähnlich verhält es sich mit dem Aspekt der Software. Dadurch dass der RobotTI gebräuchliche PC- bzw. -Notebook-Hardware verwendet, können auch unmodifizierte gebräuchliche Betriebssysteme für den lokalen Verarbeitungsrechner verwendet werden.

4 JAUS

4.1 Einführung

Die Joint Architecture for Unmanned Systems (JAUS) war ein offener Standard der JAUS Working Group (WG), der eine gemeinsame Architektur für unbemannte Systeme definierte. Die letzte veröffentlichte Version des JAUS Standards datiert auf den 27.06.2007. Danach wurde die JAUS WG in die Society of Automotive Engineers (SAE) eingegliedert und die Arbeit in nicht-öffentlicher Form fortgesetzt.

Trotz dieser Entwicklung von JAUS liefern die alten Version dieses Standards sehr interessante Ansätze zum Entwurf und zur Beschreibung von unbemannten System beliebiger Art. Der Begriff "unbemanntes System" oder "unmanned System", wie er im Original heißt, umfasst im Grunde das gleiche, wie der Begriff des mobilen autonomen Systems, wie er in Kapitel 2.3.2 eingeführt worden ist. JAUS legt dabei jedoch einen großen Fokus auf die Architektur eines solchen unbemannten Systems und weniger auf dessen Einsatzgebiet oder Verhalten. Dieser Fokus ist bei JAUS jedoch gewollt, da sich der Standard auf eine möglichst große Bandbreite von unbemannten Systemen beziehen will und in keiner Weise Voraussetzungen oder Einschränkungen auf die Bereiche Einsatzgebiet und Verhalten auferlegen möchte.

Der JAUS-Standard besteht aus mehreren Dokumenten, die sich mit jeweils unterschiedlichen Aspekten des Standards befassen:

- JAUS Volume I: Domain Model v3.2 [JAU05]
- JAUS Volume II Part 1-3: Reference Architecture v3.3 [JAU07a], [JAU07b] und [JAU07a]
- JAUS Compliance Specification v1.1 [JAU06]

[Tes10] liefert eine ins Deutsche übersetzte Zusammenfassung des JAUS Standards, doch für eine ausführliche Betrachtung der Details sei auf die direkten Quellen zurückzugreifen.

Abbildung 4 zeigt den JAUS-Standard in einer Übersichtsgrafik. Für Details hierzu sei auf die folgenden Unterkapitel verwiesen.

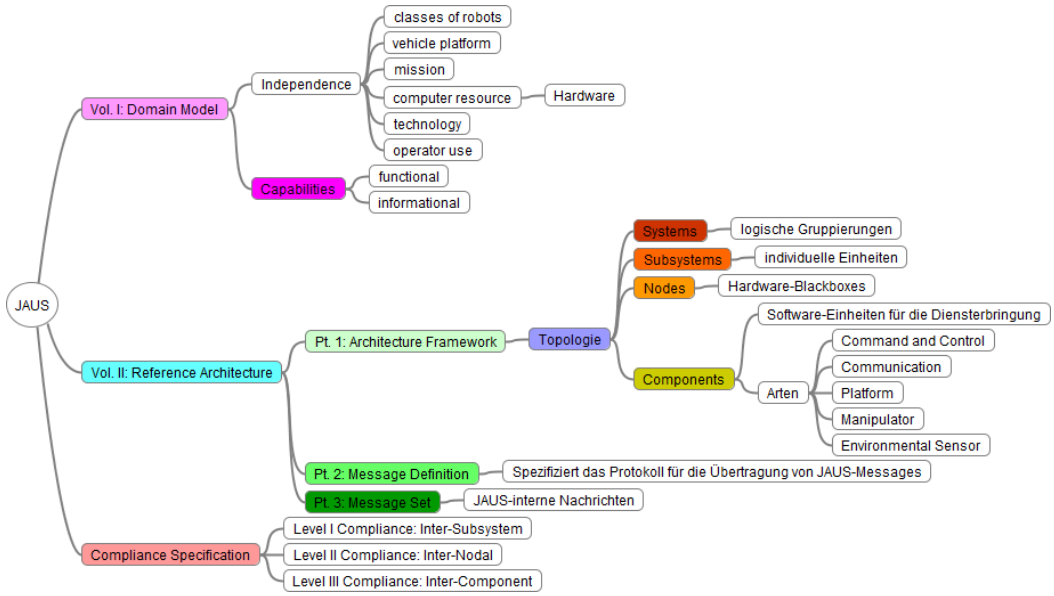

Abbildung 4: JAUS Übersicht

4.2 JAUS Volume I: Domain Model

Das Domain Model von JAUS [JAU05] vermittelt die generelle Idee, die sich hinter JAUS versteckt: Die Standardisierung einer Architektur für beliebige unbemannte Systeme, die zwar bestimmte Aspekte dringend vorschreibt, aber dennoch die Einsetzbarkeit des Standards in keiner Weise beschränkt.

Als erstes werden Anforderungen an die Architektur spezifiziert und mit einigen ausgewählten bereits bestehenden Standards verglichen. Im Verlauf dieses Vergleichs werden die Schwachstellen der anderen Standards aufgedeckt und in JAUS berücksichtigt. Als Ergebnis dieser Spezifikation werden die folgenden Stützpfeiler von JAUS festgehalten:

1. JAUS ist einsetzbar für alle Arten von Robotern und Fahrzeugplattformen.
2. JAUS erzeugt keine Beschränkungen für die Missionen, die ein unbemanntes System ausführen kann.
3. JAUS ist unabhängig von bestimmten Computer Ressourcen.
4. JAUS ist unabhängig von bestimmten eingesetzten Technologien.
5. JAUS erzeugt keine Einschränkungen für den Anwender bzw. Bediener eines unbemannten Systems.

Im JAUS Domain Model werden im Bereich des Modellierungsansatzes zusätzlich eine Reihe von Fähigkeiten (Capabilities) definiert, über die ein unbemanntes System verfügen kann. Diese Fähigkeiten sind sehr ausführlich spezifiziert. Eine Übersicht über diese Fähigkeiten findet sich in Anhang A.

4.3 JAUS Volume II: Reference Architecture

Die Referenzarchitektur von JAUS besteht aus drei Teilen:

- Part 1: Architecture Framework
- Part 2: Message Definition
- Part 3: Message Set

4.3.1 Architecture Framework

Der erste Teil, das Architecture Framework [JAU07a] stellt eine technische Spezifikation dar, nach welcher ein JAUS-konformes System entworfen und entwickelt werden sollte. Hierzu gehört die Einführung einer Topologie, die zwischen verschiedenen JAUS-Elementen besteht. Diese Topologie ist in der folgenden Grafik dargestellt und in den folgenden Unterkapiteln etwas näher beschrieben.

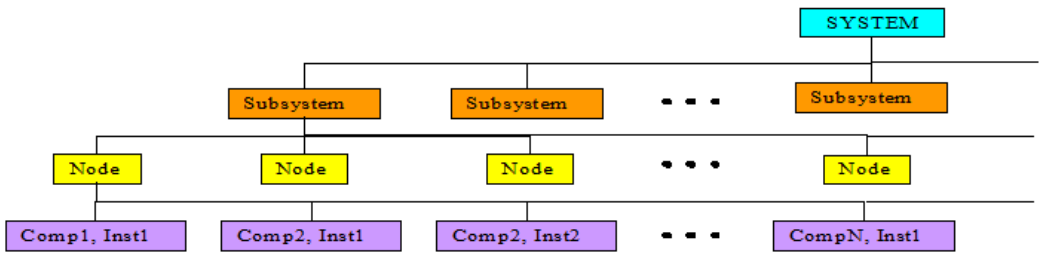

Abbildung 5: JAUS Topologie - Quelle: [JAU07a]

JAUS-Component Die JAUS-Component oder JAUS-Komponente stellt das kleinste Element bei JAUS dar. Komponenten sind definiert als eine Einheit, die eine Menge an Operationen mit einer logischen Gruppierung innerhalb des gegebenen Kontextes ausführen.

Ein wichtiges Merkmal einer JAUS-Komponente ist, dass Informationen zwischen ihnen lediglich über definierte JAUS- Nachrichten ausgetauscht werden dürfen.

Die Kommunikation zu einer Komponente, die nicht in JAUS definiert ist, darf über beliebige Nachrichten erfolgen.

Jede JAUS-Komponente besitzt eine Komponenten Schnittstelle (component interface), die als die Menge aller gültigen JAUS-Nachrichten dieser Komponente definiert ist.

Im Architecture Framework sind bereits viele JAUS-Komponenten für bestimmte Aufgaben vordefiniert. Für eine ausführliche Liste sei auf [JAU07a] verwiesen.

Eine Liste von ausgewählten JAUS-Komponenten, die bei der Beschreibung der meisten unbemannten Systeme vorhanden sind, befindet sich im Anhang B.

Node Eine Node ist definiert als eine Komposition aus allen Hardwaregeräten und Softwarekomponenten, die für eine oder mehrere wohldefinierte Verarbeitungsfähigkeiten benötigt werden. Es ist möglich, Nodes als Hardware Blackboxen zu betrachten, auf denen die benötigten Komponenten laufen.

Subsystem Ein Subsystem in JAUS ist eine unabhängige und unterscheidbare Einheit, die aus einer Menge von Komponenten besteht die auf einer oder mehreren Nodes laufen, um eine oder mehrere spezifizierte Aufgaben zu erfüllen.

System Ein System in JAUS ist eine Gruppierung von einem oder mehreren Subsystemen. Ein System wird häufig aus Subsystemen gewählt, bei denen die Kooperation untereinander einen Vorteil erbringt.

4.3.2 Message Definition

Der zweite Teil der Referenzarchitektur, die Message Definition [JAU07b] oder Nachrichten Definition, legt das JAUS- spezifische Protokoll für die Übermittlung von JAUS-Nachrichten fest.

Die Nachrichten Definition beschreibt den Aufbau der Nachrichtenheader der einzelnen definierten JAUS-Nachrichten und die Festlegungen, die für einzelne Felder im Nachrichtenheader gelten.

JAUS-Nachrichten sind einer der wichtigsten Bestandteile von JAUS, denn sie sorgen für einen wohl-definierten Nachrichtenaustausch zwischen JAUS-Komponenten und werden für das Auslösen von Aktionen und Ereignissen verwendet.

In JAUS werden Nachrichten in Nachrichtenklassen eingeteilt. Die Verwendung von Nachrichtenklassen dient der Ordnung von Nachrichten. Die Nachrichtenklassen werden durch den Command Code, also dem Typ einer Nachricht, spezifiziert und lauten wie folgt:

Message Class	Offset Range (0000h to FFFFh)
Command	0000h – 1FFFh
Query	2000h – 3FFFh
Inform	4000h – 5FFFh
Event Setup	6000h – 7FFFh (Deprecate v4.0)
Event Notification	8000h – 9FFFh (Deprecate v4.0)
Node Management	A000h – BFFFh
Reserved	C000h – CFFFh
Experimental Message	D000h – FFFFh

Abbildung 6: Einteilung von JAUS-Klassen nach Command Code - Quelle: [JAU07b]

1. **Command Class Messages** werden verwendet, um den Systemmodus zu verändern, zur Bestätigungskontrolle, um den Status einer Komponente oder eines Subsystems zu verändern oder um eine Aktion zu starten.

2. **Query Class Messages** werden verwendet, um Informationen von anderen Komponenten einzuholen.

3. **Inform Class Messages** sind das Gegenstück zu Query Class Messages. Sie erlauben es Komponenten, Daten wie z.B. Statusberichte, geographische Positionen und weitere Arten von Informationen untereinander auszutauschen.

4. **Event Setup Class Messages** werden verwendet, um die Parameter für eine Event Notification Nachricht vorzubereiten und um eine andere Komponente zu veranlassen auf ein Triggerevent zu warten. Eine Event Notice Nachricht wird immer dann erzeugt, wenn Parameter in einer Event Setup Nachricht gesetzt werden. Diese Nachrichtenklasse sollte ab JAUS v4.0 wegfallen.

5. **Event Notification Class Messages** kommunizieren das Auftreten eines Ereignisses wie z.B. Motorüberhitzung, Ölüberdruck usw. Diese Nachrichtenklasse sollte ab JAUS v4.0 wegfallen.

6. **Node Management Class Messages** werden nur von der Node Management Task verwendet. Diese Nachrichten werden für die Node-spezifische Kommunikation wie das Durchschleifen von Konfigurationsinformationen und die Komponentenregistrierung verwendet.

7. **Experimental Class Messages** werden als Grundlage für die Experimentierung mit neuen Nachrichten, die noch nicht in der JAUS Reference Architecture definiert sind, verwendet. Die Idee dabei ist, dass experimentelle Nachrichten in einer späteren Version der JAUS Reference Architecture in diese einfließen. Wenn eine Experimental Class Message in die JAUS Reference Architecture einfließt, wird ihr ein Command Code von den anderen Standardklassen zugewiesen. Die Anzahl dieser experimentellen Nachrichten sollen zwecks einer guten Interoperabilität möglichst gering gehalten werden.

Weitere Information zur Message Definition von JAUS befinden sich in [Tes10] und [JAU07b].

4.3.3 Message Set

Das Message Set von JAUS [JAU07c] beschreibt alle in JAUS definierten Nachrichten inklusive ihres Inhaltes. Der Fokus liegt dabei auf den Domänen-spezifischen Semantiken des Nachrichtenaustauschs.

Da die Menge, der in JAUS definierten Nachrichten bereits sehr groß ist, wird hier auf eine ausführliche Beschreibung aller Nachrichten verzichtet. Im Anhang C befindet sich eine Menge ausgewählter JAUS-Nachrichten. Für eine vollständige Beschreibung aller JAUS-Nachrichten sei auf [JAU07c] verwiesen.

4.3.4 Compliance Specification

Über die JAUS Compliance Specification [JAU06] kann mit Hilfe der Domain Model Spezifikation und der Referenzarchitektur bestimmt werden, ob ein System die Voraussetzungen von JAUS in so weit erfüllt, dass das System als JAUS-konform bezeichnet werden kann.

Die JAUS Compliance ist definiert als die Verifikation, dass der Nachrichtenverkehr zwischen JAUS-Komponenten der Reference Architecture Spezifikation ent-

spricht. JAUS ist eine nachrichtenbasierte Architektur, welche sich für eine Konformitätsprüfung besonders gut eignet. Es gibt drei Konformitäts-Ebenen, in denen Elemente geprüft werden können. Diese Ebenen lauten wie folgt:

- Level I: Inter-Subsystem Compliance
- Level II: Inter-Nodal Compliance
- Level III: Inter-Component Compliance

Die Erfüllung der JAUS Compliance auf einer der Ebenen ist unabhängig von der Compliance der anderen Ebenen und liefert diesbezüglich auch keinen Rückschluss auf deren Status.

Die JAUS Compliance unterscheidet zudem zwischen der Standards Compliance und der Applications Compliance. Der Unterschied zwischen diesen beiden Arten der Konformität ist, dass bei der Standards Compliance nur die im JAUS Standard definierten Nachrichten betrachtet werden, während bei der Applications Appliance auch benutzerdefinierte Nachrichten (user defined messages) herangezogen werden, die entweder die Anforderungen des JAUS Standards erweitern oder aber mit diesen in Konflikt stehen. Hierbei muss ein Element die folgenden Regeln einhalten:

1. Alle unterstützten eingehenden und ausgehenden Nachrichten des betreffenden Elements sind explizit aufgelistet.
2. Alle unterstützten ausgehenden Nachrichten dieses Elements geben ausdrücklich die Version der Referenzarchitektur an.
3. Alle unterstützten ausgehenden Nachrichten befolgen die JAUS Konventionen, Definitionen, Formate und Regeln für Nachrichten, so wie sie in der Referenzarchitektur definiert sind.
4. Alle festgelegten Toleranzen für Timing und Latenz werden eingehalten.
5. Alle unterstützten Inform-Nachrichten sollen auf die entsprechenden Query-Nachrichten antworten.
6. Alle unterstützten Event Notification Nachrichten sollen auf die entsprechenden Event Setup Nachrichten antworten.
7. Eine benutzerdefinierte Nachricht darf keine in JAUS definierte Nachricht umgehen oder duplizieren.

4.4 Surveyor als JAUS-System

Da der JAUS Standard gut geeignet ist, um die Architektur eines unbemannten Systems zu beschreiben und es sich bei dem Roboter Surveyor der Fachhochschule Köln um ein unbemanntes System handelt, wurde versucht dieses System mit Elementen aus JAUS zu beschreiben.

Ein Ansatz um den Surveyor mit Elementen aus JAUS zu beschreiben, ist dass man der Topologie von JAUS, wie in Kapitel 4.3.1 beschrieben folgt und dann die einzelnen Ebenen der Topologie denjenigen Elementen des Surveyors zuweist, die am besten auf die Definition der einzelnen Ebenen passen.

Auf der obersten Ebene hat man dann das System Surveyor, welches sich aus den beiden Subsystemen Surveyor und OCU zusammensetzt, die durch das Subsystem Network verbunden sind. Das Subsystem Network entspricht beim Surveyor dann dem WLAN Netzwerk, das der Roboter und der Proxy zur Kommunikation nutzen.

Das Subsystem Surveyor bildet hierbei den Roboter Surveyor selbst ab. Es besteht aus der Node Blackfin, die dem Blackfin Mainboard des Surveyors und der daran angeschlossenen Sensoren und Aktoren entspricht. Da der Surveyor Roboter nur aus dieser einen Node besteht, entfällt auch die Notwendigkeit eines Node Networks.

Das Subsystem OCU entspricht dem Proxy des Surveyors. Das Kürzel OCU steht für Operator Control Unit und bezeichnet in JAUS ein Element, dass ein unbemanntes System mit steuert. Steuern bedeutet in diesem Zusammenhang allerdings nicht fernsteuern, sondern eher, dass das unbemannte System Missionen, der JAUS-Begriff für Tasks, zugewiesen bekommt. Das Subsystem OCU besteht aus den beiden Nodes Bedien-PC und Entwickler-PC, die durch ein Node-Network miteinander verbunden sind.

Auf der Node-Ebene sind nun drei verschiedene Nodes identifiziert worden:

1. Das Blackfin-Board mit den angeschlossenen Sensoren und Aktoren,
2. der Bedien-PC
3. und der Entwickler-PC.

Innerhalb jeder dieser Nodes befinden sich die beiden Pflichtkomponenten Communicator und Node Manager.

Die Komponente Communicator bildet den Zugangspunkt zum Subsystem Network. Im Falle des Surveyor Roboters ist dies die Wireless LAN Schnittstelle mit welcher der Roboter mit dem Proxy kommuniziert. Die Communicator der beiden anderen Nodes sind entweder auch Wireless LAN Schnittstellen oder aber kabelgebundene Schnittstellen zu einem Wireless LAN Access Point. Für JAUS sind beide Alternativen gleichwertig, da die konkrete Implementierung des Subsystem Networks in JAUS nicht abgebildet wird.

Die andere Komponente ist der Node Manager. Der Node Manager dient den Nodes als Kommunikationsschnittstelle innerhalb eines Subsystems, dem Node Network. Im Falle des Surveyor Subsystems gibt es nur eine Node, weswegen das Node Network nicht verwendet wird. Im Falle der beiden Nodes im OCU Subsystem, ist das Node-Network physikalisch das selbe wie das Subsystem Network. JAUS unterscheidet diese beiden Netzwerke jedoch logisch voneinander, weswegen sie hier beide erwähnt werden.

Innerhalb der Node Blackfin gibt es nun mehrere Komponenten, die für die Umsetzung der Fähigkeiten des Surveyors zuständig sind:

Die Komponente System Commander, die auch auf der Node Bedien-PC vorhanden ist, bildet die Befehlsquelle des Systems. Beim Surveyor wird über eine Benutzerschnittstelle ein Wegpunkt angegeben und das System fährt autonom zu diesen Wegpunkt. Der System Commander ist die Benutzerschnittstelle, über welche dieser Wegpunkt von einem Bediener eingegeben werden kann.

Der Local Waypoint Driver nimmt den Wegpunkt (Desired Global Waypoint) und eine Geschwindigkeitsangabe (Desired Global Speed) von System Commander entgegen und berechnet einen Kurs zu diesem Wegpunkt. Hierzu bedient sich der Waypoint Driver der Daten des Global Pose Sensors.

Der Global Pose Sensor nutzt den RFID Reader als Sensor für die Bestimmung der aktuellen Position und den Kompass als Sensor für die Bestimmung der momentanen Ausrichtung der Plattform. Diese beiden Informationen bilden zusammen die sogenannte Pose, welche vom Global Pose Sensor an den Local Waypoint Driver geschickt wird.

Der Reflexive Driver erhält vom Local Waypoint Driver einen Bewegungsbefehl (Commanded Wrench Effort), welcher die Plattform zum nächsten Wegpunkt führen soll.

Der Reflexive Driver nimmt diese Information prüft mit dem Ultraschallsensor, ob sich ein Hindernis in der Fahrtrichtung befindet und passt den Bewegungsbefehl dementsprechend an, um die Sicherheit der Plattform nicht zu gefährden. Dieser abgeänderte Befehl (Modified Wrench Effort) wird dann an den Primitive Driver weitergegeben.

Der Primitive Driver bildet die eigentlich Motorsteuerung ab. Er erhält vom Reflexive Driver eine modifiziertes Steuerkommando und führt dieses aus.

Der Node Bedien-PC besitzt, wie bereits erwähnt, einen weitere Komponente vom Typ System Commander, über welche ein Benutzer dem Roboter einen Wegpunkt angeben kann. Des weiteren beinhaltet diese Node eine Komponente vom Typ World Model Vector Store, in welcher das System Zugriff auf ein Weltmodell der Testumgebung hat. Das Weltmodell besteht derzeit aus einem Koordinatensystem, dessen Punkte die jeweiligen RFID-Tags repräsentieren. Der Zugriff auf die Informationen auf die Daten in der Flurwelt-Datenbank geschieht dann über Vector Knowledge Store Objects, die über die entsprechenden Query- und Inform-Nachrichten angefragt und verschickt werden.

Der Entwickler-PC dient dem Erzeugen eines neuen uClinux Betriebssystemimages für den Surveyor Roboter. Diese Funktion ist hier als Nicht-JAUS-Komponente OS-Flasher abgebildet. Die Komponente überträgt das Image bei Bedarf über das Subsystem Network auf den Roboter.

Abbildung 7 fasst den Surveyor als JAUS-Architektur nochmals zusammen.

4.5 Bewertung

Der Joint Architecture for Unmanned Systems (JAUS) Standard ist vorrangig für die Beschreibung und den Entwurf der Architektur von unbemannten Systemen oder mobilen autonomen Robotern geeignet. JAUS hat zu diesem Zweck bereits viele vordefinierte Komponenten und Nachrichten für den Informationsaustausch zwischen diesen an Bord. Hierdurch ist es möglich, eine sehr große Auswahl an verschiedenen unbemannten Systemen mit den Möglichkeiten von JAUS in einer Art Baukastensystem zu spezifizieren. Hierbei legt JAUS einen Fokus auf die Kernbereiche der Architektur in dem die Verarbeitung von Sensoren- und Aktorendaten geschieht. Die Randbereiche der Architektur, an denen Sensoren und Aktoren an das System angebunden werden sind absichtlich nicht festgelegt um die Auswahl der Techno-

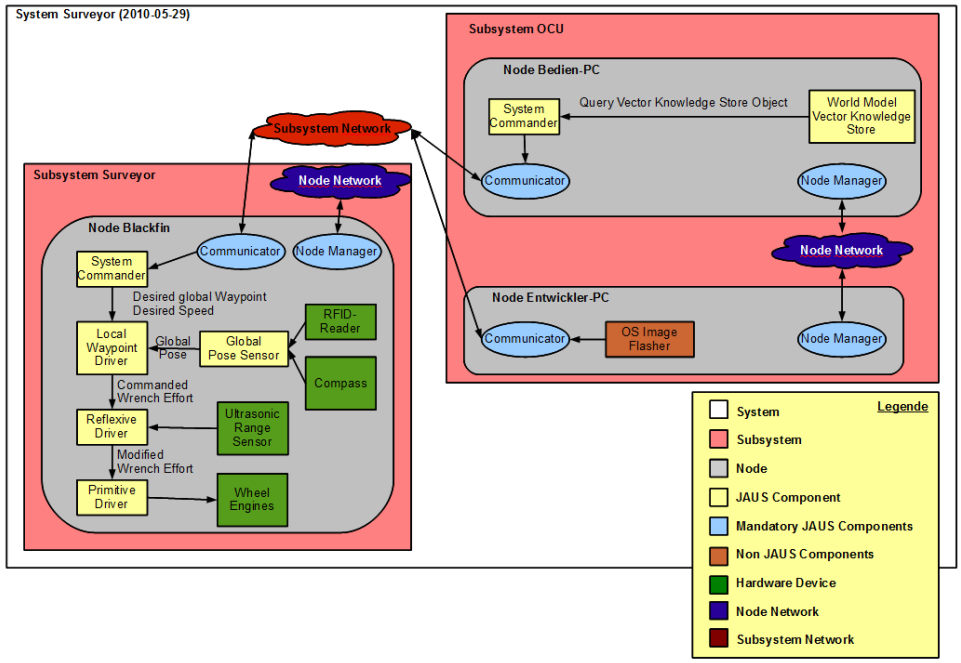

Abbildung 7: Surveyor als JAUS-System

logie der Sensoren und Aktoren nicht einzuschränken. Wenn man die Verwendung von JAUS für den Entwurf der Steuerungssoftware bei verschiedenen unbemannten Systemen stringent durchführt, kann man erreichen, dass auch teils sehr unterschiedliche Systeme mit oftmals auch sehr unterschiedlichen Einsatzgebieten, dennoch eine gemeinsame Plattform, bestehend aus standardisierten Softwarekomponenten, besitzen.

Allerdings gibt es auch Punkte die gegen eine Verwendung von JAUS sprechen. Dazu gehört zum einen die Entscheidung, den JAUS Standard nicht weiter in einer offenen Form zu entwickeln, wodurch es schwieriger wird an verbesserte Versionen des Standards zu gelangen. Es könnte sogar passieren, dass die Interessen der SAE dazu führen, dass die Plattformunabhängigkeit von JAUS zugunsten einiger weniger Plattformen aufgegeben wird. Des weiteren ist der JAUS-Standard in seiner letzten frei verfügbaren Version zwar schon recht umfangreich, aber einige wichtige Aspekte, wie die Planung von Missionen wurden bisher noch nicht spezifiziert. Es finden sich zudem einige Festlegungen in der untersuchten JAUS Standard Version, die als veraltet markiert sind und im damals nächsten geplanten Major-Release entfernt werden sollten. Dies zeigt, dass der JAUS Standard auch noch nicht frei von unnötigem Ballast ist.

Ein anderer Punkt, den man bei JAUS beachten muss, ist dass die Verwendung von JAUS-Komponenten einen gewissen Overhead mit sich bringt, den man sich auf hoch integrierten Systemen oftmals nicht leisten kann, da hierdurch mehr Ressourcen benötigt werden, als bei einer monolithischen Umsetzung der Steuerungssoftware.

Zusammenfassend kann man festhalten, dass es gute Gründe gibt, den JAUS Standard für die Spezifikation von Architekturen bei unbemannten Systemen zu verwenden, vor allem wenn man diese Architekturen einheitlich und erweiterbar halten möchte.

5 Existierende Taskbeschreibungssprachen

5.1 PDDL

Die Planning Domain Description Language, kurz PDDL, ist ein Standard für Planning Domains und Problembeschreibungen. Sie wurde 1998 von Drew McDermott entwickelt und mit Hilfe von Planungswettbewerben (International Planning Competitions) weiter entwickelt [McD11].

Der bekannteste Teil der PDDL ist STRIPS. Die Abkürzung STRIPS steht für Stanford Research Institute Problem Solver. Man bezeichnet hiermit sowohl den Solver, also eine Software zur Lösung eines Problems, als auch dessen formale Eingabesprache. Im folgenden wird die Eingabesprache kurz umrissen:

- In STRIPS werden Domänen (engl: domains) definiert. Eine Domäne hat folgende Bestandteile:
  - Prädikate (engl.: predicates): Ein Prädikat ist eine Funktion mit einem bool'schen Ergebniswert. Der Ergebniswert eines Prädikats ist wahr (true), wenn die abgefragte Bedingung erfüllt ist.
  - Aktionen (engl.: actions): Aktionen haben Vorbedingungen und Nachbedingungen. Eine Aktion kann ausgeführt werden wenn die Vorbedingungen erfüllt sind und erfüllt die Nachbedingungen nach Ausführung der Aktion.
    - Vor- und Nachbedingungen bestehen aus einer Reihe von Prädikaten.
- Planungsprobleme werden als Instanz einer Domäne beschrieben. Zu jeder Instanz gehören:
  - ein Initialzustand,
  - eine Menge deklarierter Objekte, auf welche sich die Prädikate beziehen,
  - und eine Menge von möglichen Endzuständen, für welche das beschriebene Problem gelöst ist.

Im folgenden ist ein sehr einfaches Beispiel für eine STRIPS-Instanz gegeben: In diesem Beispiel wird die Domäne "robotmovement" definiert. Hierzu wird der Text aus dem Beispiel in einer Textdatei "robotmovement.pddl" abgelegt.

```
1  ; PDDL / STRIPS Domain Beispiel
2  ; Definition der Domain robotmovement
3  ; Die Domain beschreibt die Moeglichkeiten der Manipulation der Objekte
       .
4
5  (define (domain robotmovement)
6
7  ; Deklaration der Praedikate
8  ; (room ?r)   - (room ?r) == true, wenn der Raum r existiert.
9  ; (at-room ?s)  - (at-room ?s) == true, wenn sich der Roboter im
10 ; Raum s aufhaelt.
11
12   (:predicates     (room ?r)
13              (at-room ?s)
14   )
15
16 ; Aktionen/Operationen definieren
17 ; Beschreibung: Der Roboter bewegt sich von Raum a nach Raum b.
18 ; Vorbedingung: Die Raeume a und b existieren und der Roboter befindet
19 ; sich in Raum A.
20 ; Effekt:      (at-room ?a) wird false
21 ;        (at-room ?b) wird true
22
23   (:action move
24     :parameters   (?a ?b)
25     :precondition (and
26              (and (room ?a) (room ?b))
27              (and (at-room ?a) not(at-room ?b))
28            )
29     :effect    (and (at-room ?b)
30              (not (at-room ?a))
31            )
32   )
33 )
```

Listing 1: robotmovement.pddl

Im ersten Schritt wird ein Problem namens “roomchange” in der Domäne “robotmovement” definiert. Die Beschreibung des Problems verwendet zur Lösung die Objekte ra und rb stellvertretend für die Räume a und b aus der Domänendefinition. Bei der Initialisierung des Startzustandes befindet sich der Roboter im Raum ra und nicht im Raum rb. Der Zielzustand ist dadurch definiert, dass der Roboter sich im Raum rb und nicht mehr im Raum ra befindet.

Hierdurch ist das Problem innerhalb der Domäne beschrieben. Um das Problem nun zu lösen, kann man die beiden Dateien an einen STRIPS-Solver übergeben, welcher dann versucht das gegebene Problem vom Initialzustand mit Hilfe der definierten Actions der zugeordneten Domänendefinition in den Endzustand zu überführen. Der Solver kann angewiesen werden den gesamten oder nur einen Teil des Lösungsweges auszugeben. Für dieses Beispiel sieht die Lösung des Solvers wie folgt aus:

```
1 (
2   (move ra rb)
3 )
```

Listing 2: roomchange_solved.txt

Die PDDL wird in Abbildung 8 noch einmal kurz zusammengefasst.

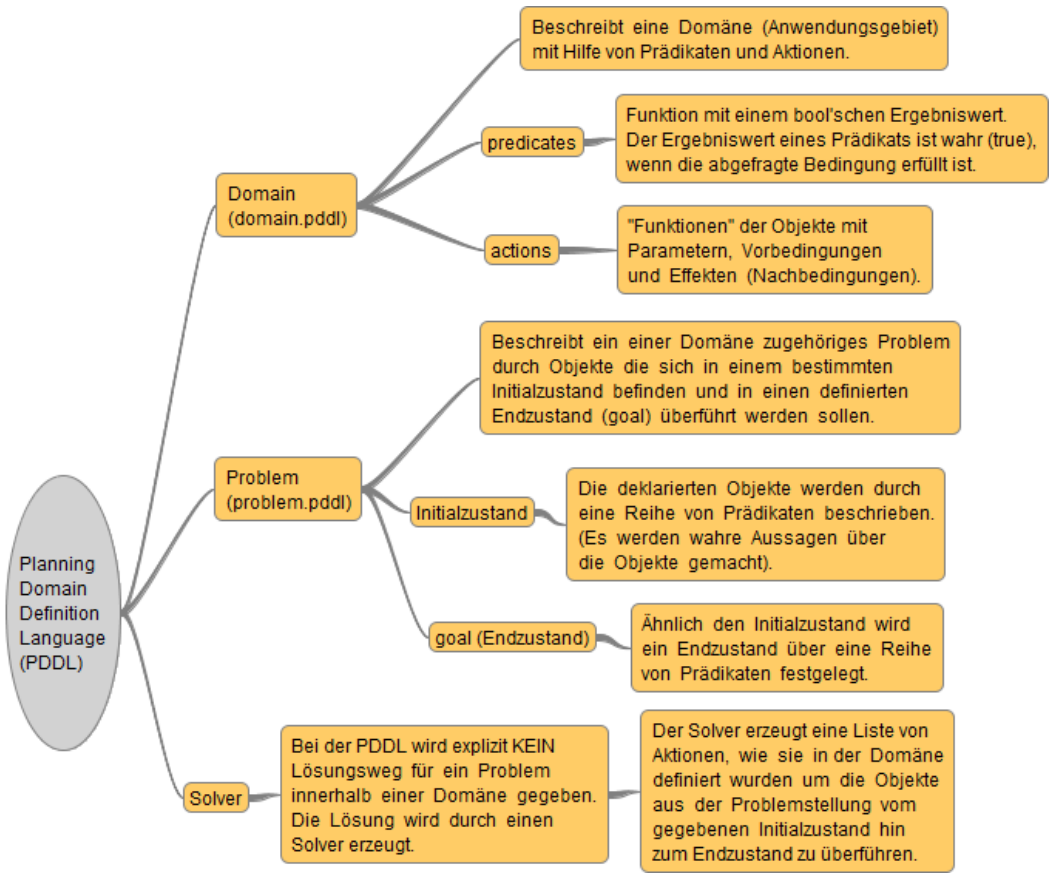

Abbildung 8: PDDL als Mindmap

5.2 Task Definition Language for Virtual Agents

5.2.1 Übersicht

Die Task Definition Language for Virtual Agents basiert auf SimHuman, einer Plattform für die Darstellung und Animation von virtuellen Agenten in Echtzeit. Virtuelle Agenten sind eine Möglichkeit in virtuellen Umgebungen Aktivitäten stattfinden zu lassen, die nicht direkt von einem oder mehreren Benutzern abhängig sind. Bei virtuellen Umgebungen spricht man auch häufig von virtueller Realität (VR = Virtual Reality). Ein virtueller Agent kann als autonome Entität in einer virtuellen Umgebung betrachtet werden, der nicht nur das Aussehen, sondern auch das Verhalten, von einem lebenden Organismus nachahmt.

Innerhalb einer virtuellen Umgebung ist ein virtueller Agent in der Lage, bestimmte Aktionen auszuführen. Werden mehrere Aktionen miteinander in einer hierarchischen oder prozedualen Struktur kombiniert, so lassen sich damit Aufgaben, die sogenannten Tasks, definieren. Durch einen geeigneten Aufbau der Tasks kann auf die Verwendung eines rechenintensiven Planers bei der Ausführung der Tasks verzichtet werden. Ein Planer wird dann lediglich auf einer höheren Ebene für den Wechsel zwischen unterschiedlichen Tasks eines virtuellen Agents benötigt.

Das Ziel der Task Definition Language ist es, Tasks in einer kontextunabhängigen Hochsprache definieren zu können, wodurch ein hoher Grad an Wiederverwendbarkeit erreicht werden kann.

5.2.2 Virtuelle Umgebung

Die virtuelle Umgebung, auch Welt oder World genannt, bildet ein übergeordnetes Objekt in dem die Menge aller Entitäten enthalten sind, die in der entsprechenden Welt existieren. Eine Entität (Entity) ist entweder ein Agent oder ein Objekt, wobei der Unterschied darin liegt, dass ein Agent die Möglichkeit hat, den aktuellen Zustand der Welt zu erfassen und Aktionen (Actions) in ihr auszuführen. Der aktuelle Zustand einer Welt setzt sich zusammen aus der Kombination der Zustände der in ihr enthaltenen Entitäten. Der Zustand einer Entität kann durch Aktionen eines Agenten verändert werden. Neben dem lebendigen Verhalten, das Agenten in einer virtuellen Umgebung zeigen können, kann es auch reaktive Objekte geben. Ein Beispiel hierfür ist ein fahrendes Auto.

In einer virtuellen Umgebung der Task Definition Language for Virtual Agents besitzen alle Entitäten eine eigene Geometrie. Es ist möglich, mehrere Entitäten in einer baumartigen Struktur miteinander zu verknüpfen, wobei eine geometrische Transformation einer Entität dieser Struktur sich auf die Entitäten in den Ebenen darunter auswirken kann.

Eine weitere Eigenschaft der virtuellen Umgebung und der Entitäten darin sind Attribute. Es wird hierbei zwischen globalen Attributen und lokalen Attributen unterschieden. Globale Attribute sind der Umgebung als ganzes zugeordnet und stehen allen Entitäten innerhalb dieser Umgebung zur Verfügung. Lokale Attribute hingegen sind innerhalb von Entitäten definiert und stehen nur über den gemeinsamen Aufruf mit dem zugehörigen Entitätsnamen zur Verfügung. Der allgemeine Aufbau eines Attributs lautet:

```
<name, type, value>
```

wobei der Name der eindeutige Name, Type der Datentyp und Value der eigentliche Wert des Attributs ist. Der Aufruf von Variablen wird in Kapitel 5.2.5 näher erläutert.

5.2.3 Die Task Definition Language

Die Task Definition Language verbindet einen Entscheidungsprozess auf oberster Ebene mit den Bewegungs- und Interaktions-Möglichkeiten eines virtuellen Agenten in einer virtuellen Umgebung. Sie ermöglicht es, Tasks als Kombinationen von parallel, sequentiell oder bedingt ausgeführten Aktionen zu beschreiben, ohne dabei alle Implementierungsdetails explizit kennen und formulieren zu müssen. Diese kontextunabhängige Art der Beschreibung bringt einen hohen Grad an Wiederverwendbarkeit mit sich. Definiert ist eine Task im Rahmen der Task Definition Language for Virtual Agents als “Eine Kombination von Aktionen eines virtuellen Agenten, die ein bestimmtes Ziel erfüllt”. Jede Task hat hierbei einen eindeutigen Namen und eine definierte Menge an Argumenten.

5.2.4 Aktionen

Eine Aktion (Action) ist ein Vorgang, ausgelöst oder durchgeführt durch einen virtuellen Agenten, der eine Änderung des Zustands der virtuellen Umgebung respek-

tive ihrer Entitäten bewirkt. Jede Aktion benötigt eine bestimmte Dauer für die Durchführung. Die Dauer wird in Zeitrahmen (time frames) gemessen. Eine Aktion hat entweder eine fest definierte Dauer oder sie ist variabel und zielorientiert, das heißt, die Aktion endet wenn ein bestimmter Zustand bei einer oder mehreren Entitäten erreicht worden ist.

Primitive Aktionen sind innerhalb eines Zeitrahmens ausführbar. Zu den primitiven Aktionen gehört zum Beispiel das Ändern einer geometrischen Eigenschaft, das Hinzufügen oder Entfernen einer Entität oder das Versenden von Nachrichten. Es liegt in der Verantwortlichkeit der virtuellen Umgebung die Aktionen letztendlich auszuführen und die resultierenden Effekte zu erzeugen. Komplexere Aktionen sind oftmals eine Kombination aus mehreren primitiven Aktionen.

Ein wichtiges Feature für die lebhafte Darstellung von Agenten in einer virtuellen Umgebung ist das sogenannte Keyframing. Hierbei handelt es sich um die Ausführung von vordefinierten Animationssequenzen.

Ein weiteres wichtiges Feature ist die Locomotion. Die Locomotion bezeichnet die Fähigkeit des Agenten sich in der virtuellen Umgebung zu bewegen. Wichtige Aktionen beinhalten:

- zu einer bestimmten Position bewegen,
- sich in eine bestimmte Richtung drehen,
- in eine bestimmte Richtung gehen
- oder einem bestimmten Pfad zu folgen.

Positionen und Orientierungen in bestimmte Richtungen sind jeweils als Vektoren gegeben. Die Locomotion Engine verwendet jedoch keine Körperteil-Bewegungen sondern ein Zustandsmodell mit dem beliebige Bewegungsabläufe möglich sind. Um die Interaktion des Agenten mit Objekten zu animieren, kann neben vordefinierten Animationen auch auf die inverse Kinematik zurückgegriffen werden. Hierbei handelt es sich um eine Reihe von Objekten die entlang einer Reihe von Vektoren bewegt werden.

Neben dem Aufruf einer einzelnen Aktion, ist es möglich eine vollständige Task mit entsprechenden Argumenten aufzurufen und damit auszuführen. Bei der Task

handelt es sich in dem Fall um eine Kombination aus Aktionen und Subtasks, die zusammen ein bestimmtes Ziel erfüllen sollen.

Manche Aktionen haben bei ihrer Ausführung keinen direkten Einfluss auf die Umgebung. Zu diesen Aktionen gehört das Kopieren von Daten zwischen einzelnen Variablen oder das Hinzufügen einer weiteren Annahme über die Umgebung im virtuellen Agenten.

5.2.5 Literals, Variablen und Funktionen

Funktionen können mit Argumenten aufgerufen werden. Die Argumente können dabei direkte Werte, Variablen eines festgelegten Datentyps oder wiederum Funktionen sein, wobei der Rückgabewert der Argumentfunktion dem Datentyp des Arguments entsprechen muss. Gültige Datentypen für Argumente sind:

- `bool`
- `integer`
- `float`
- `string`
- `entity`
- `list of entities`
- `vector`
- `list of vectors`
- `relation`

Bei Relations handelt es sich um zusammengesetzte Dateitypen bestehend aus einem eindeutigen Namen und einer Reihe von zusammenhängenden Daten von beliebigen Datenformaten, ähnlich einem Array in anderen Programmiersprachen. Ein Beispiel für eine Relation vom Typ Person kann wie folgt aussehen:

```
person('John', 28, 1.80, 'single')
```

Durch die Möglichkeit Funktionen wiederum Funktionen als Argumente übergeben zu können, ist ein verschachtelter Aufruf von mehreren Funktionen durch den Aufruf einer einzigen Funktion möglich.

In einer virtuellen Umgebung gibt es vier verschiedene Variablenmengen:

- Attribute des Agenten
- Argumente der Task
- Interne Variablen einer Task
- Attribute der Entität

Der Zugriff auf diese Variablen geschieht über folgenden Aufruf:

```
[<entity name>]<variable name>
```

Hierbei muss der ¡entity name¿ nur angegeben werden, wenn es sich um Attribute aus einer anderen als der Entität handelt, aus welcher die Abfrage gestartet wird.

Es gibt spezielle Funktionen um die räumliche Abhängigkeit zwischen zwei Entitäten zu bestimmen:

- `near(e1,e2)`
- `on(e1,e2)`
- `front_of(e1,e2)`
- `behind(e1,e2)`
- `left_of(e1,e2)`
- `right_of(e1,e2)`
- `above(e1,e2)`
- `below(e1,e2)`
- `intersect(e1,e2)`: Kollisionserkennung

Diese Funktionen bestimmen einen bool'schen Rückgabewert über den Vergleich von Position, Größe und Ausrichtung zweier Entitäten, die als Argumente übergeben werden.

Des weiteren gibt es Funktionen, die eine neue Position in einer vorher festgelegten Distanz relativ zu einer gegebenen Position erzeugen können. Diese lauten:

- `left`
- `right`
- `front`
- `behind`
- `above`
- `below`
- `on`

Zusätzlich sind die folgenden Logikfunktionen definiert:

- `and`
- `or`
- `not`

Die Funktion `exists(ent,f)` stellt eine weitere Besonderheit dar. Sie liefert den Wert true zurück, wenn in der Welt die Funktion f mit dem Argument ent und deren Argumenten, den Wert true zurückgibt.

5.2.6 Tasks definieren

Eine Task besteht aus drei Teilen:

1. Task Definition:
   `TASK name(type1, arg1, type2, arg2, , typen, argn)`

2. Dem String `#Variables` gefolgt von der Variablendeklaration: `type name = value`

3. Dem String `#Body` gefolgt von einem oder mehreren Kommandos. Dieser Block wird abgeschlossen durch den String `#end`. Mögliche Kommandos sind:

   - `<action>`: eine einzelne Aktion
   - `PAR(<block b1>, <block b2>)`: Gleichzeitige Ausführung von b1 und b2.
   - `DO(<block b>) UNTIL c`: Der Block b wird solange ausgeführt bis die Bedingung c erfüllt ist.
   - `IF <bool c> THEN(<block b1>) ELSE (<block b2>)`: Wenn die Bedingung c wahr ist, führe b1 aus, sonst führe b2 aus.

Sollte irgendein Kommando innerhalb einer Task scheitern, so scheitert ebenfalls die gesamte Task. Bei `PAR` dürfen die beiden Blöcke nicht zur gleichen Zeit auf die selben kritischen Ressourcen zugreifen, da die Aktionen ansonsten scheitern würden. Eine Besonderheit bei `DO UNTIL` ist, dass die Bedingung nach der Ausführung jedes einzelnen Schritts im Verarbeitungsblock ausgewertet wird und nicht erst wenn der ganze Block verarbeitet worden ist. Hierdurch ist ein Verlassen der `DO UNTIL` Schleifen quasi jederzeit möglich.

5.2.7 Beispiel

Ein Agent in einem Restaurant sucht einen freien Tisch an den er sich setzen kann.

```
1  TASK sit_on_free_table()
2  #Variables
3      entity Table ''
4      entity Chair ''
5      entity Human ''
6  #Body
7  DO (
8      task walk_around()
9  ) UNTIL exists( Table, and(
10     eq( [Table]class, 'table' ),
11       not( exists( Chair,
12         and( eq( [Chair]class, 'chair' ),
13           and( near( Chair, Table ),
14             exists( Human,
```

```
15              and( eq([Human]class, 'human'),
16                intersect( Human, Chair ) )
17              )
18            )
19          )
20        )
21      )
22   )
23 );
24 task go_and_sit(Table)
25 #end
```

Listing 3: sit_on_free_table.task

Der Agent läuft den Raum mittels eines bestimmten Pfades ab (`walk_around()`) bis er einen Tisch sieht an dem sich Stühle befinden, auf denen kein Mensch sitzt. Dabei impliziert diese Taskbescheibung, dass sich an jedem Tisch auch Stühle befinden. Hat er einen solchen Tisch gefunden, setzt er sich an diesen Tisch (`go_and_sit(Table)`).

6 Robotic Task Definition Language

6.1 Einführung

Die Robotic Task Definition Language, kurz RTDL, ist eine Sprache, mit welcher Tasks definiert und beschrieben werden können. Die Tasks, die mit Hilfe der RTDL beschrieben werden können, werden speziell für Roboter definiert, welche diese Taskbeschreibungen dann mit Hilfe eines Interpreters auswerten und ausführen können.

In diesem Kapitel werden zunächst der Begriff der Task und die Anforderungen an die RTDL festgelegt. Den Abschluss dieses Kapitels bildet die Definition der RTDL und ihrer Elemente.

6.2 Tasks und Anforderungen an die RTDL

Eine Task stellt eine spezifische Anweisung dar, die von einem Roboter ausgeführt werden soll. Da es sich bei einer Task um die Zuweisung einer Aufgabe an einen Roboter handelt, lassen sich bestimmte Parallelen zur Kommandierung einer Aufgabe zwischen Menschen finden.

Wenn es in einer zwischenmenschlichen Konstellation zu einer Situation kommt, in welcher ein Mensch einem anderen Menschen eine Anweisung gibt, welche der zweite Mensch ausführen soll, so geschieht dies oftmals über den Weg der natürlichen Sprache. Dies umfasst sowohl die mündliche, als auch die schriftliche Art der Aufgabenzuteilung. Im Kern der Anweisung steht immer ein Verb, das eine Tätigkeit beschreibt. Zudem ist es oftmals notwendig, noch weitere Informationen in die Anweisung zu packen, um diese präzise zu formulieren. Die Anweisung “gehe in die Küche” beschreibt die gewünschte Tätigkeit mit dem Verb “gehen” und ist parametriert mit der Entität “Küche”, die das Ziel der Bewegung ausmacht.

Damit eine Roboter eine bestimmte Task ausführen kann, müssen bestimmte Gegebenheiten geschaffen werden. Zum einen muss eine Tätigkeit für einen Roboter in einer geeigneten Art und Weise formuliert sein, damit der Roboter die notwendigen Informationen zur Durchführung einer Tätigkeit erhält. Diese Informationen erhält der Roboter durch eine Taskbeschreibung.

Eine Anforderung an eine solche Taskbeschreibungssprache ist, dass sie ihr Ein-

satzgebiet nicht selbst einschränkt. Dies beinhaltet insbesondere, dass eine Taskbeschreibungssprache für Roboter sowohl unabhängig von der konkreten Hardware eines Roboters, als auch unabhängig von der Plattform, also der Softwareumgebung, eines konkreten Roboters sein soll. Einschränkungen bezüglich der Möglichkeit der Durchführung einer Task dürfen sich nur aus den Kombination von Roboter und gestellten Aufgabe ergeben, jedoch nicht aus der Art der Beschreibung der Tätigkeit.

Eine weitere Anforderung an die RTDL ist es, dass die Elemente, aus denen eine Taskbeschreibung aufgebaut ist, eine Blockstrukturierung aufweisen. Hierdurch lassen sich sowohl die beschriebene Task selbst, als auch ihre Elemente, also die enthaltenen Aktionen, aus gleichartigen Bausteinen aufbauen, die sogar ineinander verschachtelt sein können. Ein weiterer Vorteil von Blöcken ist, dass sie sich sowohl seriell als auch parallel abarbeiten lassen, wodurch eine potentielle Effizienzsteigerung in der Ausführung einer Task erreicht werden kann.

Schließlich sollte die RTDL noch über eine XML-basierte (eXtensible Markup Language) Repräsentation einer Taskbeschreibung verfügen, weil dadurch eine Vielzahl von Verarbeitungsmöglichkeiten mit bekannten und ausgereiften Werkzeugen möglich ist. Zu diesen Werkzeugen gehört die Möglichkeit der Prüfung der Wohlgeformtheit (wellformed XML) und der Validität über eine Document Type Definition (DTD) (valid XML). Außerdem bietet die XML Transformation Stylesheet Language (XSLT) die Möglichkeit, eine Taskbeschreibung im XML-Format in ein beliebiges anderes Format zu transformieren.

6.3 Taskzuweisung an Roboter

Bei der Taskzuweisung wird ein Roboter vor einige Herausforderungen gestellt, die ein Mensch eher intuitiv zu lösen weiß. So ist nicht jeder Roboter in der Lage auf gesprochene Kommandos zu reagieren, da er hierfür über eine robuste Spracherkennung verfügen muss. Eine robuste Spracherkennung kann gesprochene Sätze in die Schriftform überführen, man spricht hierbei von Text-to-Speech.

In [Hol10] werden die Fähigkeit der robusten Spracherkennung und der Sprachsynthese genannt. Bei der Spracherkennung muss der Roboter gesprochene Sätze erkennen und analysieren können. Darüber hinaus muss der Roboter entscheiden, ob es sich bei einem gesprochenem Satz um eine Anweisung an ihn handelt oder nicht. Ebenso wichtig ist in diesem Zusammenhang eine gute Sprachsynthese, damit der Roboter qualifizierte Rückmeldungen zu einer Anweisung geben kann. Diese

Fähigkeiten sind für einen Roboter vor allem in einer Umgebung, in der er für oder mit Menschen arbeiten soll, sinnvoll. Dies gilt vor allem in Bezug auf die Effizienz, Flexibilität und Akzeptanz seiner kooperativen Arbeit mit Menschen, da die Zuweisung von Tasks an einen Roboter unmittelbar vor Ort und ohne Hilfsmittel vorgenommen werden kann.

Aber selbst wenn gesprochene Sätze in ihrer Schriftform vorliegen, sind weitere Hürden für den Roboter zu nehmen. Menschen sprechen in einer natürlichen Sprache. In [MT10] werden einige Schwierigkeiten, welche die Verwendung von natürlicher Sprache bei Maschinen mit sich bringt, erläutert. Zunächst ist die natürliche Sprache voller Mehrdeutigkeiten und selten präzise genug formuliert, um von einer Maschine direkt verarbeitet werden zu können. Zudem sprechen Menschen oftmals Sätze, die in einem bestimmten Kontext vorliegen. Bei der zwischenmenschlichen Kommunikation ist der Kontext der Kommunikationspartner oftmals der selbe. Ist dies nicht der Fall, so entstehen häufig große Missverständnisse in der Kommunikation, wie jeder Mensch sicher schon einmal leidvoll erfahren musste.

Für einen Roboter existiert ein solcher Kontext jedoch oftmals nicht. Somit müssen auch Anweisungen in der Schriftform oftmals weiter analysiert und verarbeitet werden, bevor ein Roboter diese als geeignete Anweisung ausführen kann. In seiner Arbeit beschreibt [Bud09] einen Küchenroboter, der Küchenrezepte, die in natürlicher Sprache geschrieben sind, mit Hilfe einer Satzanalyse in Anweisungen zerlegt und in Tätigkeiten umsetzt, welche das Rezept virtuell “nachkochen”. Bei der Domäne eines Küchenroboters ist der Kontext der natürlichsprachlichen Anweisungen von Rezepten jedoch vollständig bekannt, ein Luxus den ein generischer Roboter nicht hat.

Bei der Entwicklung der RTDL wurde aus diesem Grund auf die Verwendung der natürlichen Sprache zur Taskbeschreibung und -zuweisung verzichtet und stattdessen eine Funktionssyntax für Aktionen definiert, in welcher Tasks beschreiben werden können.

Aus [MT10] wurde die folgende Liste erstellt, welche eine Reihe von Fragestellungen bezüglich der Zuweisung von Tasks an Roboter bildet.

- Besitzt ein Roboter das gesamte Wissen, dass für die Erfüllung einer Aufgabe notwendig ist und wie erhält er dieses Wissen? Hierzu gehört:
  - Bestimmung der involvierten Objekte bei der Ausführung einer Aktion.

  - Korrekte Positionierung eines Roboters zu einem bestimmten Objekt.
  - Spezifische Parameter bei der Manipulation eines Objekts wie zum Beispiel die verwendete Kraft beim Greifen und der sichere Ablauf von Armbewegungen an einem Objekt.
  - Bestimmung des Zustands von Objekten.
  - Das Ziehen von Schlüssen aus bestimmten Gegebenheiten in der Umgebung des Roboters oder eines Objekts.

- Auf was begründet ein Roboter die symbolischen Repräsentationen in seinem Weltmodell?
- Wie soll ungenaues Wissen abgebildet und wie soll damit umgegangen werden.
- Was ist eine geeignete Repräsentation von Wissen, die sowohl ausdrucksstark ist, als auch ein schnelles Schließen (Reasoning) ermöglicht?
- Wie erhält der Roboter ein Bewusstsein über Aktionen und wie können Auswirkungen von Aktionen vorhergesagt werden?

All diese Punkte gilt es beim Einsatz von Robotern zu beachten, während sie für einen Menschen oftmals intuitiv erschlossen und einbezogen werden. Die hier genannten Punkte befassen sich vorwiegend mit der Fragestellung wie eine bestimmte Aktion ausgeführt werden soll und weniger mit der Frage, welche Aktionen eine Task ausmachen. Somit sind diese Punkte eher für die Entwicklung eines Roboters als für die Entwicklung einer Taskbeschreibungssprache von Belang.

6.4 Aufbau von RTDL Taskbeschreibungen

Im letzten Unterkapitel wurde auf die Schwierigkeit der Verwendung von natürlicher Sprache für die Taskzuweisung bei Robotern eingegangen. Diese Umstände haben dazu geführt, dass bei der RTDL ein anderer Ansatz für die Beschreibung von Tasks gewählt wurde.

Die verwendete Funktionssyntax verwendet bei der Definition einer Task einen Funktionsnamen und eine Menge von Parametern zu genaueren Spezifikation der Task. Der Funktionsname wird hierbei durch die englische Version jenes Verbs gebildet, welches den Kern einer Anweisung bildet. Angehängt an den Funktionsnamen wird

eine in runde Klammern gefasste und durch Kommata getrennte Liste von Parametern. Die einzelnen Parameter bestehen wiederum aus einem Entitätstypen und einem Parameternamen. Das folgende Beispiel zeigt die Anweisung "Gehe in die Küche" in der Funktionssyntax:

```
moveTo(kitchen);
```

Hierbei bezeichnet "moveTo" das Verb für Bewegung und "kitchen" das Ziel der Bewegung. Definiert ist die Funktion "moveTo" allgemein als:

```
moveTo(location loc);
```

Hierbei ist "location" der Entitätstyp des ersten und einzigen Parameters und "loc" die Referenz auf den übergebenen Wert dieses Parameters.

Zur Beschreibung einer Task ist neben der Definition der Signatur, also des Aufrufs der Task, auch deren Umsetzung von Bedeutung. Jede Task besteht intern aus einer Abfolge von Aktionen (actions), die von einem Roboter ausgeführt werden müssen, um das Ziel der Task zu erreichen. Anders als bei der PDDL, vgl. Kapitel 5.1, ist das Ziel einer RTDL Task nicht explizit beschrieben, sondern wird durch die Ausführung der Task erreicht. Das bedeutet: Eine Task gilt als erfolgreich ausgeführt, wenn alle Aktionen innerhalb der Taskbeschreibung erfolgreich ausgeführt werden konnten.

Eine Aktion, innerhalb einer Taskbeschreibung kann dabei entweder eine primitive Aktion sein, die von einem Roboter unmittelbar umgesetzt werden kann, oder aber eine bereits definierte Task. Aus diesem Grund wird eine Aktion genauso wie eine Task durch eine Signatur mit einem Funktionsnamen und einer Parameterliste beschrieben.

Da eine Task auch im Rahmen der Beschreibung einer anderen Task aufgerufen werden kann, lassen sich Taskhierarchien aufbauen, die jedoch auf der untersten Ebene immer auf primitiven Aktionen von Robotern basieren, denn nur diese sind von einem Roboter direkt ausführbar.

6.5 Capabilities

Die Capabilities eines Roboters sind die Fähigkeiten, die der Roboter besitzt, seine Umgebung zu erfassen, zu verändern und sich in ihr zu bewegen. Sie werden be-

stimmt über die Kombination seiner Sensoren, Aktoren und der verwendeten Steuerungssoftware.

Im Kontext der RTDL sind die Capabilities eines Roboters gegeben durch die Menge der primitiven Aktionen (actions), die ein Roboter über die Steuerungssoftware als Funktionen anbietet. So ist beispielsweise ein Roboter mit angetriebenen Rädern und ausreichenden Sensoren zur Kollisionserkennung physikalisch in der Lage, sich über bestimmte Terrains zu bewegen. Es wird jedoch eine Schnittstelle benötigt, um diese Fähigkeiten nutzen zu können. Diese Schnittstelle muss durch die Steuerungssoftware angeboten werden, beispielsweise in Form einer API (Application Programming Interface), welche diese Funktionalität als Software-Funktion anbietet.

Führt ein Roboter eine RTDL Task aus, die auf einer oder mehreren Aktionen basiert, welche nicht als Capability in der Steuerungssoftware angeboten werden und die auch nicht als definierte RTDL Task vorhanden ist, so scheitert diese Task. Ist diese Task nun Teil einer höherrangigen Taskdefinition so scheitern alle höherrangigen Tasks ebenfalls.

6.6 Definition der Robotic Task Definition Language

6.6.1 Funktionssyntax - Form

In diesem Unterkapitel wird der Aufbau der RTDL festgelegt. Ein RTDL-Dokument ist eine Textdatei, in welcher Tasks in Form einer Funktionssyntax spezifiziert werden. Innerhalb einer RTDL-Taskbeschreibung dürfen ausschließlich Zeichen aus dem ASCII-Zeichensatz (American Standard Code for Information Interchange) [Wik10] verwendet werden.

In einer Programmiersprache sind Kommentare ein weit verbreitetes und probates Mittel, um ein Programm für einen Menschen einfacher lesbar zu gestalten. Die RTDL definiert ebenfalls Kommentare in der gleichen Form wie in der Programmiersprache Java:

```
/*
 * Ich bin ein Kommentar und ich erstrecke mich über eine
 * oder mehrere Zeilen.
 */
```

Die RTDL ist flexibel im Umgang mit Leerzeichen, Tabulatoren und Zeilenumbrüchen. RTDL-Elemente müssen durch Abstandshalter voneinander getrennt werden. Der Standard-Abstandshalter ist das Leerzeichen. Es ist jedoch auch möglich anstatt eines einzelnen Leerzeichens beliebig viele Leerzeichen, Tabulatoren oder Zeilenumbrüche oder eine Kombination aus diesen zu erzeugen.

6.6.2 Funktionssyntax - Aufbau

Ein RTDL-Dokument beginnt immer mit der folgenden obligatorischen Zeile:

```
rtdl <Versionsnummer>
```

Hiermit wird angezeigt dass es sich um ein RTDL-Dokument in der Version `<Versionsnummer>` handelt. Da die RTDL hier erstmalig definiert wird, ist hier bislang nur der Wert 1.0 gültig. Somit sieht die erste Zeile wie folgt aus:

```
rtdl 1.0
```

In einem RTDL-Dokument dürfen Kommentare an beliebigen Stellen verwendet werden, außer in der ersten Zeile vor der RTDL-Versionsdeklaration.

Nach der ersten Zeile folgt eine Taskdefinition, die wie folgt beginnt:

```
task <Taskname>(<Argtyp1> <Argname1>, ..., <ArgtypN> <ArgnameN>) {
 <Body>
}
```

Das Schlüsselwort `task` leitet die Taskdefinition ein und wird gefolgt von einem Tasknamen. An den Tasknamen wird unmittelbar eine in Klammern gefasste und durch Kommata getrennten Liste von Argumenten angehängt, die ihrerseits aus dem Entitätstypen und einem eindeutigen Namen des Arguments bestehen. Der in geschweifte Klammern gefasste Body der Task bildet die Implementierung der Task und wird im folgenden näher beschrieben.

Als erstes wird eine optionale Menge von benannten Enititäten, die für die Ausführung der Task benötigt werden, erzeugt. Diese Menge wird definiert durch mehrere Strings mit dem folgenden Aufbau:

```
entity <Entitätstyp> <Entitätsname>;
```

Das Schlüsselwort `entity` zeigt an, dass eine benannte Entität definiert werden soll. Der Entitätstyp und der Entitätsname sind selbsterklärend. Abgeschlossen werden diese Definitionen durch ein Semikolon.

Eine Entität oder Entity bezeichnet im Sinne der RTDL ein irgendwie geartetes Objekt, mit dem ein Roboter auf eine beliebige Art interagieren kann. Jede Entität ist genauer durch einen Entitätstypen spezifiziert, der angibt um welche Art von Objekt es sich handelt. Oftmals sind mit bestimmten Entitätstypen bestimmte Eigenschaften und Interaktionsmöglichkeiten verknüpft. Wenn ein Roboter einen bestimmten Entitätstypen kennt, so besitzt er häufig eine Menge von möglichen Interaktionen, die er auf diesem Entitätstypen ausführen kann.

Im letzten und wichtigsten Abschnitt einer RTDL-Taskbeschreibung wird die eigentliche Task durch eine Menge von Operationen beschrieben. Zu diesen Operationen gehört der Aufruf von primitiven Aktionen und bereits definierten Tasks. Ein Aufruf einer Aktion oder Task mit ihren entsprechenden Argumenten hat folgenden Aufbau:

```
<Taskname>(<Argname1>, ..., <ArgnameN>);
```

Der Aufruf einer Task oder Aktion besteht aus dem Aufruf des Namens der Task oder Aktion gefolgt durch eine Liste von Argumenten, abgeschlossen durch ein Semikolon.

6.6.3 Funktionssyntax - Erweiterte Ausführungskonstrukte

Neben der seriellen Ausführung von Tasks oder Aktionen, ermöglicht die RTDL, die Ausführung durch Konstrukte, wie man sie in anderen Programmiersprachen vorfindet.

Eine While-Schleife ist ein solches Konstrukt, bei dem eine Liste von Tasks oder Aktionen so lange ausgeführt wird, solange eine boolsche Laufzeitbedingung wahr ist. Ist `<Bedingung>` nicht erfüllt, so wird die Ausführung nach dem While-Block fortgesetzt. In der RTDL hat eine While-Schleife den folgenden Aufbau:

```
while(<Bedingung>)
do
```

```
<Liste von Tasks oder Aktionen>
done
```

Ein weiteres Konstrukt ist die bedingte Ausführung durch if. Hierbei wird eine Bedingung geprüft. Eine Liste von Tasks oder Aktionen wird nur dann ausgeführt, wenn eine bestimmte boolsche Bedingung erfüllt ist, ansonsten wird der If-Block übersprungen. Innerhalb der RTDL hat ein If-Konstrukt den folgenden Ausbau:

```
if(<Bedingung>)
<Liste von Tasks oder Aktionen>
endif
```

In bestimmten Situationen ist ein gleichzeitiges Ausführen von mehreren Tasks oder Aktionen notwendig. Diese Möglichkeit wird durch das Par-Konstrukt gegeben, bei dem zwei Blöcke von Anweisungen zeitgleich ausgeführt werden. Der Aufbau des Par-Konstrukts ist wie folgt:

```
par
<Liste von Tasks oder Aktionen>
to
<Liste von Tasks und Aktionen>
endpar
```

6.6.4 Funktionssyntax - Abfangen des Scheiterns einer Task

Generell gilt, dass wenn eine Task oder Aktion innerhalb einer Taskdefinition bei deren Ausführung scheitert, so scheitert die gesamte definierte Task. Dieses Verhalten wurde in [SV03] ähnlich definiert. Diese Einschränkung gilt nicht für Bedingungen im Kopf von While- oder If-Blöcken. Jedoch bedeutet dies für die Ausführung einer Task, dass es eine oftmals hohe Wahrscheinlichkeit gibt, dass diese Task während ihrer Ausführung scheitert und damit eine ganze Hierarchie von Tasks und Aktionen scheitert. Um diese Problematik etwas abzuschwächen, gibt es verschiedene Lösungsansätze.

Es ist beispielsweise möglich, dass die Ausführung einer Task zu einem bestimmten Zeitpunkt scheitert, kurz darauf jedoch möglicherweise erfolgreich ausgeführt werden könnte. Um sich diesen Umstand nutzbar zu machen, kann man den Aufruf von

bestimmten Tasks eine Schleife packen, die eine bestimmte Anzahl an Versuchen hintereinander versucht, diese Task auszuführen. Ein solches Konstrukt hat den folgenden Aufbau:

```
retry(int retries)
<Liste von Tasks oder Aktionen>
endtry
```

Darüber hinaus kann man auch die Anzahl der Ausführungsversuche für alle Tasks und Aktionen innerhalb einer Taskdefinition global festlegen. Dies kann man beispielsweise über die folgende Anweisung innerhalb des Bodys einer Taskdefinition festlegen:

```
retrycount(int tries);
```

Innerhalb einer Taskdefinition kann es neben den notwendigen Ausführung auch die Möglichkeit der optionalen Ausführung von Tasks und Aktionen geben. Wenn eine solche optionale Task oder Aktion scheitert, hat dieses keinen Einfluss auf die Ausführung der Task als ganze. Die optionale Ausführung einer Task oder Aktion erreicht man durch das Schlüsselwort `optional` vor der Ausführung:

```
optional <taskname>(<argument1>, ..., <argumentN>);
```

6.7 Robotic Task Definition Language XML-Syntax

Neben der Funktionssyntax der Robotic Task Definition Language, wie sie in Kapitel 6.6 definiert wurde, wurde eine äquivalente XML-Repräsentation definiert. Der Aufbau einer Taskbeschreibung in der XML-Syntax ist im Grunde der selbe wie bei der Funktionssyntax:

Die erste Zeile hat stets den folgenden Aufbau:

```
<?xml version="1.0" encoding="UTF-8"?>
```

Diese Zeile wird gefolgt von der Deklaration des XML-Stylesheets, mit welchem die XML-Syntax in die Funktionssyntax transformiert werden kann. Das Stylesheet ist als Quellcode in Listing 22 auf Seite 144 integriert.

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

Nach der Deklaration des XML-Stylesheets kann man optional noch die Document Type Definition (DTD) einbinden. Die DTD müsste sich bei diesem Beispiel im selben Verzeichnis wie die Taskbeschreibung befinden und kann eingebunden werden durch die folgende Zeile:

```
<!DOCTYPE rtdl SYSTEM "rtdl.dtd">
```

Das eigentliche RTDL Dokument ist in das folgende Wurzeltag gefasst:

```
<rtdl version="1.0">
</rtdl>
```

Innerhalb des Wurzeltags und auch innerhalb der weiteren Tags können Kommentare mit dem folgenden Tag eingeleitet werden:

```
<comment>This is a comment.</comment>
```

Die Definition einer Task wird umschlossen von:

```
<task name="exampleTask">
</task>
```

Innerhalb der Taskdefinition werden als erstes die Argumente der Task als Var-Tags mit ihren Entitätstypen aufgelistet:

```
<taskargs>
  <var type="FirstArg">a1</var>
  <var type="SecondArg">a2</var>
 </taskargs>
```

Die globale Anzahl der Ausführungsversuche (Standardwert: 1) jeder Aktion wird durch das folgende optionale Tag überschrieben:

```
<retrycount>1</retrycount>
```

Als nächstes folgt die Aufzählung der Entitäten, die während der Taskbeschreibung verwendet werden, in Form von Var-Tags:

```
<entities>
   <var type="Entity">entity1</var>
   <var type="Entity">entity2</var>
   <var type="Entity">entity3</var>
 </entities>
```

Der interne Ablauf der Task wird durch ein Block-Tag eingeleitet:

```
<block>
</block>
```

Innerhalb eines Block-Elements können nun die verschiedenen Elemente zur Beschreibung des Ablaufs einer Task in beliebiger Reihenfolge und Anzahl verwendet werden. Zu diesen Elementen gehören:

Die Aktion mit ihrem Namen und ihren zugehörigen Parametern. Die Parameter einer Aktion werden in Arg-Tags gefasst. Arg-Tags sind vergleichbar mit Var-Tags, jedoch wird hierbei kein Entitätstyp angegeben.

```
<action>
  <name>Task1</name>
  <args>
    <arg>entity1</arg>
    <arg>entity2</arg>
    <arg>entity3</arg>
  </args>
</action>
```

Die bedingte Ausführung eines Blocks durch ein If-Konstrukt:

```
<if>
 <cond>condition</cond>
  <block>
  </block>
 </if>
```

Die Ausführung eines Blocks in einer While-Schleife:

```
<while>
<cond>condition</cond>
  <block>
  </block>
</while>
```

Die parallele Ausführung zweier Blöcke block und block2:

```
<par>
  <block>
  </block>
  <block2>
  </block2>
</par>
```

Die Ausführung eines Blocks mit mehreren Wiederholungsversuchen:

```
<retry count="5">
   <block>
  </block>
</retry>
```

Die optionale Ausführung einer Aktion, bei der ein Scheitern keine Auswirkung auf den Erfolg der definierten Task hat:

```
<action optional="true">
</action>
```

Durch die Verwendung des Block-Tags ist eine Verschachtelung von beliebiger Tiefe bei der Beschreibung einer Task möglich.

Die Struktur einer RTDL Taskbeschreibung ist zusätzlich in Form einer Document Type Definition (DTD) festgehalten. Diese hat den folgenden Aufbau:

```
1 <!-- RTDL v1.0 Document Type Definition -->
2
```

```
3  <!ELEMENT rtdl (comment*, task, comment*)>
4  <!ATTLIST rtdl
5    version CDATA #REQUIRED
6  >
7  <!ELEMENT comment (#PCDATA)>
8  <!ELEMENT task (comment*,
9                  taskargs,
10                 comment*,
11                 retrycount?,
12                 comment*,
13                 entities,
14                 comment*,
15                 block,
16                 comment*
17 )>
18 <!ATTLIST task
19   name CDATA #REQUIRED
20 >
21 <!ELEMENT taskargs (var | comment)*>
22 <!ELEMENT var (#PCDATA)>
23 <!ATTLIST var
24   type CDATA #REQUIRED
25 >
26 <!ELEMENT retrycount (#PCDATA)>
27 <!ELEMENT entities (var | comment)*>
28 <!ELEMENT block (action | par | if | while | retry | comment)*>
29 <!ELEMENT action (comment*, name, comment*, args?, comment*)>
30 <!ATTLIST action
31   optional CDATA #IMPLIED
32 >
33 <!ELEMENT name (#PCDATA)>
34 <!ELEMENT args (arg | comment)*>
35 <!ELEMENT arg (#PCDATA)>
36 <!ELEMENT par (block, block2)>
37 <!ELEMENT if (cond, block)>
38 <!ELEMENT cond (#PCDATA)>
39 <!ELEMENT while (cond, block)>
40 <!ELEMENT retry (block)>
41 <!ATTLIST retry
42   count CDATA #REQUIRED
43 >
```

Listing 4: rtdl.dtd

Die Einbinden einer DTD in Verbindung mit einem XML-Parser erlaubt die Validierung einer in der XML-Syntax formulierten RTDL Taskbeschreibung.

6.8 Beispiel für eine Task: fetchWater(Glass g, Tap t)

Das folgende Beispiel zeigt wie eine Task `fetchWater(Glass g, Tap t)` in der Funktionssyntax aufgebaut sein könnte.

Bei dem Beispiel soll ein Roboter einem Benutzer ein Glas gefüllt mit Wasser holen, wobei er das Wasser aus einem Wasserhahn besorgt. In einer natürlichsprachlichen Formulierung würde man sagen: "Fetch me a glass of water from the tap!". Aus dem Verb "fetch" leitet sich der Name der Task ab. Der Zusatz "water" im Tasknamen gibt der Task einen Namen mit mehr Aussagekraft. Die Substantive "glass" und "tap" werden zu den Argumenten der Task.

Eine Besonderheit bei dieser Task ist, dass sie zwei Argumente verwendet. Das erste Argument "Glass g" bezeichnet den Gegenstand, mit dem der Roboter das Wasser transportieren soll. Das zweite Argument "Tap t" bezeichnet die Quelle, an welcher der Roboter das Wasser erhält. Der Ablauf dieser Task wird im Folgenden beschrieben.

Der Roboter befindet sich anfänglich an einem bestimmten Ausgangsort und muss sich als erstes ein Glas besorgen. Dazu fährt er zu dem Glas hin und greift es. Dieses Glas wird dann zu einem Wasserhahn transportiert und mit Wasser gefüllt. Danach fährt der Roboter wieder zu seinem Ausgangsort zurück, wo er das Glas absetzt.

Im folgenden wird dieses Beispiel in Form einer RTDL-Taskbeschreibung in der Funktionssyntax beschrieben. Die Umsetzung beschränkt sich hierbei jedoch auf die oberste Ebene der Taskdefinition und lässt die Definition der aufgerufenen Tasks und Aktionen weg.

```
1 rtdl 1.0
2
3 /* In dieser Task soll ein Roboter mit einem Greifarm und der
      Möglichkeit
4  * sich in seiner Umgebung zu bewegen ein Glas Wasser besorgen und am
5  * Ausgangspunkt abstellen.
6  */
7 task fetchWater(Glass g, Tap t) {
8 /* Wegpunkt für die Ausgangsposition. */
9   entity Waypoint start;
```

```
10
11 /* Merke die Ausgangsposition */
12   setCurrentPosition(start);
13 /* Lokalisiere das Glas und fahre zu dessen Position */
14   move(g);
15 /* Greife das Glas */
16   grip(g);
17 /* Lokalisiere den Wasserhahn und fahre zu dessen Position */
18   move(t);
19 /* Stelle das Glas ab */
20   ungrip(g);
21 /* Benutze den Wasserhahn mit dem Glas, um es zu fuellen */
22   fill(Tap1, Glass1);
23 /* Greife das Glas */
24   grip(g);
25 /* Fahre zum Ausgangspunkt zurueck */
26   move(start);
27 /* Stelle das Glas ab */
28   ungrip(g);
29 }
```

Listing 5: fetchWater.rtdl

Um diese Task ausführen zu können, muss ein Roboter die folgenden Tasks oder Aktionen ausführen können:

- `setCurrentPosition(Waypoint w);`
  Der Roboter merkt sich die aktuelle Position in Form eines Wegpunktes `w`.
- `move(entity e);`
  Der Roboter lokalisiert die Position einer Entität `e` und bewegt sich zu dieser hin. Wenn `e` nicht gefunden oder nicht erreicht werden kann, scheitert diese Task.
- `grip(entity e);`
  Der Roboter greift die greifbare Entität `e`. Wenn `e` nicht gegriffen werden kann, scheitert diese Task.
- `ungrip(entity e);`
  Der Roboter setzt die Entität `e`, die er in einem seiner Greifarme hält, an der aktuellen Position ab. Wenn `e` nicht abgesetzt werden kann, scheitert diese Task.

- `fill(source s, container c);`
  Der Roboter befindet sich an einer `WorldPosition`, an der sich zusätzlich ein WorldObject abgeleitet von `source` und ein `WorldObject` abgeleitet von `container` befinden. Der Roboter lässt dann beide `WorldObjects` so interagieren, dass das von `container` abgeleitete `WorldObject` mit dem Ausstoß des von `source` abgeleiteten `WorldObjects` gefüllt wird. In diesem Beispiel ist `t` abgeleitet von `source` und `g` abgeleitet von `container`.

- `wait(int timeframes);`
  Der Robooter wartet so viele vordefinierte Zeitrahmen ab, wie der Task übergeben wurden. Diese Task kann nicht scheitern.

- `check(entity e, condition c);`
  Der Roboter nutzt seine Sensoren um den Wahrheitswert der Bedingung `c` einer Entität `e` zu prüfen. Ist die Bedingung nicht erfüllt, so liefert check ein `false` zurück.

Dieses Beispiel, jedoch in der XML-Syntax definiert, befindet sich in den Quelltexten im Anhang in Kapitel 20.

7 Roboter Simulationsumgebung

7.1 Einleitung

Die Roboter-Simulationsumgebung bildet den praktischen Teil dieser Masterarbeit. Sie besteht aus einer einfachen Robotersimulation und einem RTDL-Parser. Die Simulationsumgebung hat den Zweck, das Verhalten eines virtuelles Roboters zu verdeutlichen, der eine in der RTDL spezifizierte Task ausführt. Die eigentliche Simulation ist auf Textnachrichten beschränkt, welcher der virtuelle Roboter ausgibt anstatt reale Aktionen auszuführen. Durch den Einsatz einer Simulationsumgebung ist es möglich, sowohl während der Entwicklung, als auch nach Fertigstellung der RTDL-Spezifikation die Auswirkungen von Tasks zu verdeutlichen.

7.2 Simulationsumgebung

Im folgenden wird der Aufbau der Simulationsumgebung beschrieben. Sie setzt sich zusammen aus einem Weltmodell und einem Roboter der als die Summe seiner Sensor-, Aktor- und Kommunikationskomponenten gesehen wird.

7.2.1 Weltmodell

Das verwendete Weltmodell repräsentiert einen dreidimensionalen Raum, welcher aus diskreten Orten, den sogenannten WorldLocations, besteht. Innerhalb des Weltmodells befinden sich Weltobjekte, die sogenannten WorldObjects. Der virtuelle Roboter basiert ebenfalls auf einem WorldObject.

Die grobmaschige Einteilung der Welt in Form eines dreidimensionalen Gitters wurde gewählt, da die Simulationsumgebung lediglich als Hilfsmittel für die Entwicklung der RTDL gedacht ist. Zwar würde eine realistischere Simulationsumgebung gerade bei komplexen Tasks einen größeren Nutzen bieten, sie wäre allerdings auch mit einem erheblich größeren Entwicklungsaufwand verbunden, was letztendlich sogar die Entwicklung der RTDL in den Schatten stellen würde.

Im folgenden wird das verwendete Weltmodell anhand von Abbildung 9 erläutert:

Das Weltmodell der Simulationsumgebung besteht aus diskreten Punkten, die einen dreidimensionalen Raum aufspannen. Zur besseren Übersicht wird in Abbildung 9

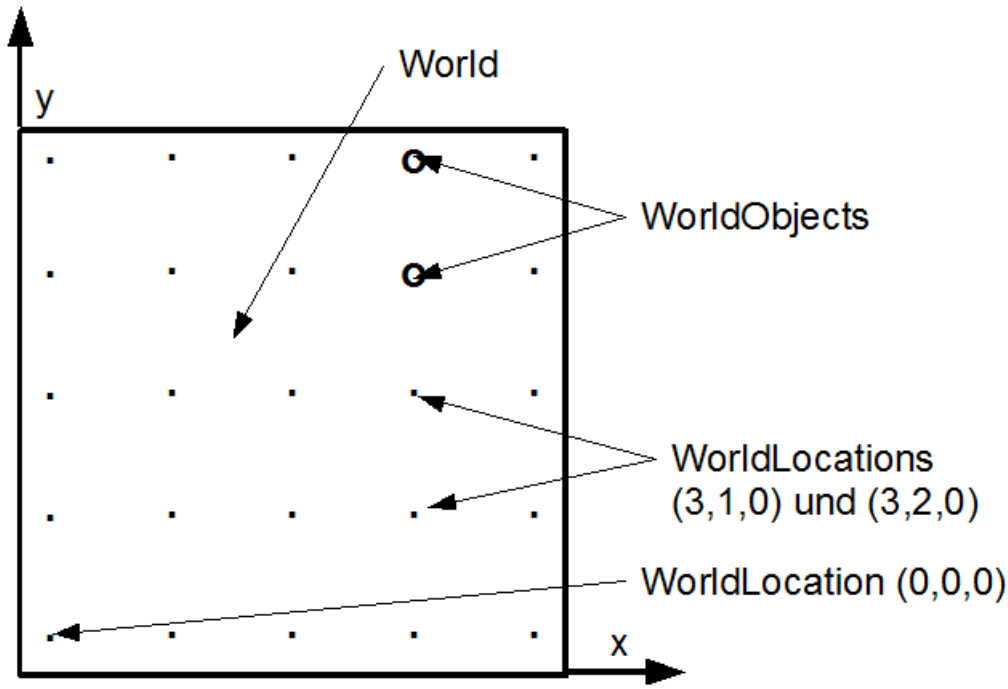

Abbildung 9: RobotSim Weltmodell

lediglich eine zweidimensionale Fläche zur Erläuterung verwendet.

Jeder Punkt in der Abbildung entspricht einer WorldLocation, also einem eindeutigen Ort innerhalb der Welt. Jede WorldLocation besitzt eine eindeutige dreidimensionale Koordinate. An jeder WorldLocation können sich beliebig viele WorldObjects befinden. In der Abbildung sind diese als Kreise dargestellt. Es besteht eine Beziehung mit einer 1:n Kardinalität zwischen den Orten und den Objekten. Das bedeutet, dass ein WorldObject sich lediglich an genau einer WorldLocation befinden kann, jede WorldLocation jedoch 0 bis n WorldObjects beinhalten kann.

Ein WorldObject entspricht einer Entität in der simulierten Welt. Jedes WorldObject wird durch einen eindeutigen Namen referenziert und besitzt einen definierten Typen, entsprechend dem Entitätstypen in der RTDL.

7.2.2 Sensoren, Aktoren und Kommunikationsgeräte

Im Rahmen der Simulationsumgebung werden Fehler in Sensordaten weder betrachtet noch modelliert. Der Grund hierfür ist, dass die Simulationsumgebung bewusst einfach gehalten werden soll.

In der hier verwendeten Simulationsumgebung werden Aktionen der Aktoren als

Meldungen dargestellt, die anzeigen, ob die gewünschte Aktion erfolgreich durchgeführt wurde oder nicht.

Diese Simulationsumgebung definiert in der aktuellen Version keine Kommunikationsgeräte, da dies für die Entwicklung der RTDL keine Relevanz besitzt.

7.3 Implementierung der Simulationsumgebung

Die Roboter-Simulationsumgebung ist in der Programmiersprache Java implementiert. Bei der Entwicklung wurde Wert auf eine einfache Erweiterbarkeit gelegt. In ihrer Basisversion verfügt ein simulierter Roboter über die Möglichkeiten sich in einer Welt zwischen den einzelnen WorldLocations zu bewegen. Ferner wurden Wände implementiert, die einem Roboter das beschreiten bestimmter WorldLocations verbieten. Die Simulationsumgebung verfügt derzeit über keine Benutzerschnittstelle. Wenn ein bestimmter Aufbau der Welt erzeugt werden soll, so müssen die einzelnen WorldObjects von Hand mit Hilfe von definierten Java-Methoden an die entsprechenden WorldLocations gesetzt werden. Dies geschieht im Quelltext der Simulationsumgebung. Hiernach muss das Programm neu kompiliert werden, was jedoch nur wenige Sekunden dauert. Die Entscheidung auf eine Benutzeroberfläche zu verzichten, ist begründet in dem Aufwand diese ständig mit der Simulationsumgebung anzupassen, sofern man diese erweitert.

Um nun die Benutzerfreundlichkeit etwas zu erhöhen wurde eine Funktion in das Programm übernommen, die es erlaubt, den Aufbau einer definierten Welt in eine XML-Datei zu sichern, die später auch wieder geladen werden kann.

Für die Implementierung der Basisversion der Roboter-Simulationsumgebung wurden folgende Klassen erzeugt:

- WorldPosition.java
- WorldLocation.java
- WorldObject.java
- World.java
- Wall.java
- Robot.java

- RobotSim.java
- MyXmlParser.java
- Main.java

Die wichtigsten Implementierungsdetails der einzelnen Klassen werden in den folgenden Unterkapiteln erläutert.

7.3.1 WorldPosition

Eine WorldPosition ist eine einfache Klasse, um die Position innerhalb einer Welt durch eine X-, Y- und Z-Koordinate zu bestimmen. Die Koordinatenwerte sind als Datentyp long implementiert, wodurch die Welt lediglich aus diskreten Punkten besteht.

Neben den obligatorischen Getter- und Setter-Methoden sind die folgenden Methoden erwähnenswert:

- `public long getOffsetX(WorldPosition wp)`
  Diese Methode erhält als Argument ein WorldPosition-Objekt und gibt den Abstand der X-Koordinaten der aktuellen zur übergebenen WorldPosition zurück. Wenn die übergebene WorldPosition einen höheren oder den gleichen X-Koordinatenwert besitzt, wie die aktuelle WorldPosition, so ist der Rückgabewert positiv beziehungsweise Null, ansonsten ist er negativ.

- `public long getOffsetY(WorldPosition wp)`
  analog zu `public long getOffsetX(WorldPosition wp)`

- `public long getOffsetZ(WorldPosition wp)`
  analog zu `public long getOffsetX(WorldPosition wp)`

- `public boolean equals(WorldPosition wp)`
  Diese Methode gibt `true` zurück, wenn alle drei Koordinatenwerte der aktuellen WorldPosition mit denen der übergebenen WorldPosition übereinstimmen, ansonsten wird `false` zurückgegeben.

7.3.2 WorldObject

Die Klasse WorldObject bildet die Basisklasse für alle Typen von Weltobjekten, die innerhalb dieser Simulationsumgebung definiert sind. Jedes WorldObject besitzt einen eindeutigen Namen und eine Reihe von Prädikaten. In der Basisversion sind folgende Prädikate vordefiniert:

- `protected boolean canMove = false`
  Dieses Prädikat legt fest, ob ein WorldObject in der Lage ist sich selbstständig innerhalb der Welt zu bewegen.

- `protected boolean grippeable = false`
  Dieses Prädikat legt fest, ob ein WorldObject greifbar ist.

- `protected boolean passable = true`
  Dieses Prädikat legt fest, ob ein WorldObject durch ein sich bewegendes Objekt passierbar ist (`true`), oder ob es sich um ein Hindernis handelt (`false`).

In der Klasse WorldObject können weitere Prädikate definiert werden, sollte dies für die Implementierung eines neuen WorldObject Typs notwendig sein. Ferner werden relevante Prädikate in Ableitungen von WorldObjects überschrieben.

Neben den obligatorischen Getter- und Setter-Methoden gibt es weitere relevante Methoden:

- `public String getObjectType()`
  Diese Methode liefert den Typ eines abgeleiteten WorldObjects als String zurück.

Darüber hinaus können in dieser Klasse Platzhalter-Methoden für jene Methoden definiert werden, die in abgeleiteten WorldObjects implementiert wurden. Ein Beispiel hierfür ist die Methode `public boolean move(World w, int x, int y, int z)`. Durch den Aufruf dieser Methode eines abgeleiteten WorldObject soll sich dieses in der Welt w an die WorldLocation bewegen, die durch die übergebenen X-, Y- und Z-Koordinaten spezifiziert ist. Da der Standardwert für das Prädikat `canMove` allerdings `false` ist, muss im abgeleiteten WorldObject sowohl dieses Prädikat auf `true`

gesetzt als auch diese Methode überschreiben werden , so dass eine Bewegung innerhalb der Welt möglich wird. Für alle anderen abgeleiteten WorldObjects wird die Methode der Oberklasse WorldObject aufgerufen, die lediglich eine Fehlermeldung ausgibt, dass sich das aktuelle Objekt nicht innerhalb der Welt bewegen kann.

7.3.3 Wall

Die Klasse Wall ist eine einfache Ableitung der Klasse WorldObject und als Unterklasse von WorldObject definiert. Der Konstruktor `Wall(String objectName)` erzeugt ein Objekt der Klasse Wall, setzt den Objektnamen entsprechend dem Argument und setzt das Prädikat passable auf `false`, wodurch eine Wall zu einem Hindernis wird.

7.3.4 Waypoint

Die Klasse Waypoint ist eine weitere einfache Ableitung der Klasse WorldObject. Der Konstruktor `Waypoint(String objectName)` erzeugt ein Objekt der Klasse Waypoint und setzt den Objektnamen entsprechend dem Argument. Die Klasse Waypoint repräsentiert einen Wegpunkt in der Welt. Ein Wegpunkt bezeichnet einen benannten Ort.

7.3.5 Robot

Im Vergleich zur Klasse Wall ist die Klasse Robot eine ungleich kompliziertere Ableitung der Oberklasse WorldObject. In der Basisversion implementiert die Klasse Robot einen Roboter, der in der Lage ist, sich innerhalb der Welt frei zu bewegen, sofern er nicht auf Hindernisse stößt.

Ein Roboter verfügt wie jedes WorldObject über einen eindeutigen Namen. Zusätzlich ist eine Blickrichtung implementiert, die an die einfache Welt aus diskreten Punkten angepasst ist. So kann ein Roboter entweder nach Norden, Süden, Osten, Westen, Oben oder Unten schauen. Die Änderung der Blickrichtung ist jederzeit über den Aufruf entsprechender Methoden möglich.

In der Basisversion der Simulationsumgebung ist die wichtigste Fähigkeit des Roboters sich in der Welt zu bewegen. Hierzu wurde die Methode public boolean `move(World w, int x, int y, int z)` der Oberklasse WorldObjects entsprechend

angepasst. Diese Methode verwendet einen sehr einfachen Algorithmus, um von der aktuellen Position zu einer Zielposition zu gelangen. Alle weiteren Methoden in der Basisversion dienen als Hilfsmethoden für die Methode `public boolean move(World w, int x, int y, int z)`.

Der Algorithmus bestimmt zunächst die Differenz der X-Koordinaten der aktuellen und der Zielposition. Dann bewegt sich der Roboter so lange entlang der X-Achse bis die Koordinaten übereinstimmen oder er auf ein unpassierbares Hindernis gestoßen ist. Das gleiche wiederholt der Roboter dann entlang der Y- und dann entlang der Z-Achse. In einem Raum ohne Hindernisse erreicht der Roboter stets sein Ziel, sofern das Ziel innerhalb der definierten Welt liegt. Beim Vorhandensein von Hindernissen zeigt der einfache Algorithmus sehr schnell seine Schwächen, da er in vielen Szenarien den Zielpunkt oftmals nicht erreicht, obwohl das über einen Umweg möglich gewesen wäre. Die offene Architektur der Simulationsumgebung erlaubt es jedoch, diesen Algorithmus jederzeit durch einen besseren zu ersetzen.

7.3.6 WorldLocation

Eine WorldLocation bezeichnet einen eindeutigen Ort innerhalb der Welt. Die Position einer WorldLocation wird durch ein WorldPosition Objekt festgelegt. Darüber hinaus verfügt eine WorldLocation über eine HashMap in welcher die WorldObjects, die sich an diesem Ort befinden, über ihren eindeutigen Namen referenziert sind.

Neben den obligatorischen Getter- und Setter-Methoden sind die folgenden Methoden erwähnenswert:

- `public void removeWorldObject(String name)`
  Diese Methode entfernt das WorldObject mit dem eindeutigen Namen `name`.

- `public boolean isPassable()`
  Diese Methode liefert `false` zurück, wenn sich an diesem Ort mindestens ein WorldObject befindet, dass nicht passierbar ist, ansonsten gibt sie `true` zurück.

7.3.7 World

ie Klasse World repräsentiert die Welt der Simulationsumgebung. Der Konstruktor `World(int x, int y, int z)` erzeugt ein Java-Objekt vom Typ World. Die Argu-

mente `int x`, `int y` und `int z` bestimmen hierbei die Ausdehnung der Welt entlang der jeweilige Koordinatenachsen. Jedes Argument muss mindestens den Wert 1 besitzen, da die Welt sonst nicht erzeugt werden kann.

Innerhalb einer Welt befinden sich alle WorldLocations in Form eines dreidimensionalen Arrays. Über die Koordinaten der Position ist dann ein direkter Zugriff auf das entsprechende Objekt im dreidimensionalen Array aller WorldObjects möglich.

Ferner bietet die Klasse World einen vereinfachten Zugriff auf alle WorldObjects in allen WorldLocations innerhalb der Welt über deren eindeutigen Namen. Dies wird durch die Verwendung einer HashMap, die zu dem eindeutigen Namen eines WorldObjects dessen Position innerhalb der Welt ablegt, erreicht.

Im folgenden sind die wichtigsten Methoden der Klasse World aufgeführt:

- `public WorldObject getWorldObjectByName(String name)`
  Gibt das WorldObject, das durch den übergebenen Namen referenziert wird, zurück.

- `public String getWorldObjectType(String name)`
  Gibt den Typ des WorldObjects, das durch den übergebenen Namen referenziert wird, zurück. Der Typ bezeichnet hierbei den Klassennamen der von WorldObject abgeleiteten Klasse.

- `public boolean removeWorldObjectByName(String name)`
  Entfernt das WorldObject, das durch den übergebenen Namen referenziert wird, aus der Welt.

- `public boolean contains(String name)`
  Prüft, ob ein WorldObject mit dem Namen `name` in der Welt existiert.

- `public String[] getWorldObjectNames()`
  Gibt die Namen aller in der Welt befindlichen WorldObjects als String-Array zurück.

7.3.8 RobotSim

Die Klasse RobotSim bildet die eigentlich Simulationsumgebung ab. Diese Klasse ist hauptsächlich ein Wrapper um ein Java-Objekt vom Typ World mit Methoden,

die den Zugriff auf die Welt vereinfachen.

- `public boolean createWorldObject(String type, String name, int x, int y, int z)`
  Diese Methode erzeugt ein Objekt der Klasse `type` und dem Namen `name` an der Position in der Welt, die durch die Koordinaten `x`, `y` und `z` spezifiziert wird. Der Typ `type` ist hierbei eine definierte Unterklasse der Klasse WorldObject.

- `public boolean createRobot(String RobotName, int x, int y, int z)`
  Diese Methode entspricht einem Aufruf der Methode
  `public boolean createWorldObject(String type, String name, int x, int y, int z)` mit Type `Robot`.

- `public WorldObject getObject(String name)`
  Diese Methode gibt Zugriff auf das WorldObject, dass durch `name` referenziert wird.

- `public boolean removeObject(String name)`
  Diese Methode entfernt das WorldObject, das durch `name` referenziert wird, aus der Welt.

- `public boolean place(String name, int x, int y, int z)`
  Diese Methode platziert das WorldObject, das durch `name` referenziert wird, direkt an die Position, die durch `x`, `y` und `z` gegeben ist.

- `public boolean move(String name, String destination)`
  Diese Methode ruft die Methode `public boolean move(World w, int x, int y, int z)` des WorldObjects, das durch name referenziert wird, mit den Argumenten `x`, `y` und `z` auf. Die Argumente `x`, `y` und `z` ergeben sich aus den Koordinaten, an denen sich das WorldObject mit dem Namen aus der Variablen `destination` befindet.

- `public String toXmlString()`
  Diese Methode erzeugt auf der Konsole die Ausgabe eines aktuellen Abbildes der Welt mit ihren Dimensionen und allen WorldLocations inklusive aller in ihnen enthaltenen WorldObjects im XML-Format.

7.3.9 MyXmlParser

Diese Klasse ermöglicht es den Zustand einer Welt, wie er mit der Methode `public String toXmlString()` erzeugt wurde, aus einer Textdatei zu laden und die Welt entsprechend zu rekonstruieren. Der Konstruktor `MyXmlParser(String filename)` bekommt hierzu als `filename` den Pfad zu der Datei übergeben, in welcher der Zustand der Welt als XML-Beschreibung abgelegt worden ist. Das Parsen der Datei wird durch einen angepassten `XMLStreamreader` erledigt.

7.3.10 Main

Die Klasse Main bildet den Eingabeteil der Benutzerschnittstelle der Roboter-Simulationsumgebung ab. In ihr wird ein RobotSim- und ein MyXmlParser-Objekt definiert. In der ausführbaren Methode `public static void main(String[] args)` kann dann entweder die Simulationsumgebung durch den Aufruf von Methoden der Klasse RobotSim manuell erzeugt und gesichert werden oder alternativ durch Aufruf des MyXmlParsers aus einer bereits bestehenden Umgebungsbeschreibung rekonstruiert werden.

Manuelle Erzeugung der Simulationsumgebung Das folgende Beispiel der Methode `public static void main(String[] args)` erzeugt ein neues RobotSim Objekt mit den Dimensionen x=10, y=10, z=1. Innerhalb dieser Simulation wird ein WorldObject mit eben jenen Dimensionen automatisch erzeugt. Danach wird ein Roboter namens “Robot1” an der Position [0, 0, 0] erzeugt, der sich in einem weiteren Schritt zur Position [5, 5, 0] bewegen soll.

Anschließend soll der aktuelle Zustand der Umgebung als XML-Beschreibung auf die Konsole ausgegeben werden.

```
public static void main(String[] args) {
  // Erzeugung der Simulationsumgebung
  rs = new RobotSim(10, 10, 1);

  // Erzeugung eines Roboters
  rs.createRobot("Robot1", 0, 0, 0);

  // Bewegung des Roboters zu Position [5, 5, 0]
```

```
  rs.move("Robot1", 5, 5, 0);

  // Sicherung des aktuellen Zustands der Welt.
  echo(rs.toXmlString());
}
```

Der gesicherte Zustand der Welt sieht dann wie folgt aus:

```
<world>
  <sizex>10</sizex>
  <sizey>10</sizey>
  <sizez>1</sizez>
  <objects>
    <object>
      <name>Robot1</name>
      <type>Robot</type>
      <x>5</x>
      <y>5</y>
      <z>0</z>
    </object>
  </objects>
</world>
```

Rekonstruktion der Simulationsumgebung Das folgende Beispiel der Methode `public static void main(String[] args)` lädt eine bereits definierte Simulationsumgebung aus der lokalen Datei ‘‘`C:\RobotSim.xml`’’ auf einem Windows-System:

```
public static void main(String[] args) {
  // Beispiel Import
  try {
    MyXmlParser parser = new MyXmlParser("C:\\RobotSim.xml");
    rs = parser.parse(rs);
  } catch (Exception e) {
    System.err.println("ERROR beim Parsen!");
  }
```

```
// Manipulation der rekonstruierten Umgebung
// ...
}
```

Hierbei wird ein neuer MyXmlParser mit der Datei `C:\RobotSim.xml` erzeugt und gestartet. Zu beachten ist, dass alle Blackslashes "\" im Windows-Pfad durch ein weiteres Backslash-Zeichen escapet werden müssen.

7.4 Erweiterung der Simulationsumgebung

Die Roboter-Simulationsumgebung wurde im Hinblick auf einfache Erweiterbarkeit entworfen. Um das Beispiel aus Kapitel 6.8 umsetzen zu können, muss die Simulationsumgebung um mehrere Bestandteile erweitert werden. In diesem Kapitel werden die Erweiterungen an Hand dieses Beispiels erläutert.

Die Task `fetchWater(DrinkingGlass glass, Tap tap)` besteht aus den folgenden Aktionen:

1. `moveTo(posGlass)`
2. `grip(glass)`
3. `moveTo(posTap)`
4. `ungrip(glass)`
5. `fill(tap, glass)`
6. `grip(glass)`
7. `moveTo(posStart)`

Hierbei bezeichnet `posGlass` die Position von einem Glas `glass` in der Welt, `posTap` die Position eines Wasserhahns `tap` und `posStart` die Startposition des Roboters. Da die Task `moveTo(position p)` bereits in der Basisversion der Simulationsumgebung implementiert ist, ist hier keine Erweiterung notwendig.

Im folgenden werden die hierzu notwendigen Erweiterungen beschrieben. Zunächst werden die neuen Weltobjekt Typen definiert und dann in die Simulationsumgebung integriert.

7.4.1 DrinkingGlass

Die Klasse DrinkingGlass repräsentiert ein Wasserglas in der Simulationsumgebung. Der Konstruktor `DrinkingGlass(String objectName)` setzt beim Instanzieren den Objektnamen und einige Prädikate:

- `super.isContainer = true;`
  Dieses Prädikat gibt an, ob es sich bei dem WorldObject um einen Container handelt, der etwas beinhalten kann.
- `super.isGrippeable = true;`
  Dieses Prädikat gibt an, ob das WorldObject greifbar ist.

Neben dem Objektnamen und den Prädikaten wird zusätzlich eine neue interne Variable `this.contentType = new String()` deklariert. Der `contentType` gibt an, mit welcher Art von Inhalt ein Container-artiges WorldObject gefüllt ist.

Die Klasse `DrinkingGlass` definiert zusätzlich zwei neue Methoden:

- `public boolean fillContainer(World w, String source)`
  Diese Methode füllt das DrinkingGlass mit dem Erzeugnis einer Quelle `source`.
- `public boolean emptyContainer()`
  Diese Methode leert das DrinkingGlass.

Diese Klasse wurde direkt von der Oberklasse WorldObject abgeleitet. Die nächsten beiden Kapitel geben ein Beispiel, wie ein Weltobjekt indirekt von der Oberklasse WorldObject abgeleitet definiert werden kann

7.4.2 Source

Die Klasse Source repräsentiert eine Quelle, die einen gewissen Ausstoß produziert. Der Konstruktor `Source(String objectName, String sourceType)` setzt zum einen den Objektnamen der Source und definiert zusätzlich das Prädikat `isSource`, das angibt, ob es sich bei dem WorldObject um eine Source handelt und die Variable `sourceType`, welche die Art des Quellenausstoßes bezeichnet.

7.4.3 Tap

Die Klasse Tap repräsentiert einen Wasserhahn in der Simulationsumgebung. Der Tap ist abgeleitet von der Oberklasse Source, die ihrerseits von der Oberklasse WorldObject abgeleitet ist. Der Konstruktor `public Tap(String objectName)` setzt den Objektnamen und den Quellentyp hartkodiert auf `water`.

7.4.4 Robot

Um mit den Objekten der neu definierten Klassen interagieren zu können, muss der Roboter erweitert werden.

Im ersten Schritt erhält der Roboter einen Arm, der ein greifbaren WorldObject fassen kann:

```
private WorldObject arm;
```

Des weiteren erhält der Roboter neue Methoden um greifbare WorldObjects greifen und wieder abstellen zu können:

- `public boolean grip(World w, String name)`
  Diese Methode lässt den Roboter ein Object name greifen, wenn dieses das Prädikat `isGrippeable` erfüllt und wenn der Roboter sich an der selben Position wie das benannte Objekt befindet.
- `public boolean ungrip(World w, String name)`
  Mit dieser Methode stellt der Roboter das mit `name` benannte Objekt an seiner aktuellen Position ab, sofern es sich beim Methodenaufruf in seinem Greifarm befindet.

7.4.5 WorldObject

Da in der Definition der Klassen `DrinkingGlass`, `Source` und `Tap` neue Prädikate und Methoden definiert worden sind, müssen diese in die Klasse `WorldObject` übernommen werden.

Die zusätzlichen Prädikate werden einfach zu den bereits existierenden Prädikaten hinzugefügt:

```
/* Die bereits definierten Prädikate */
protected boolean canMove = false;
protected boolean isGrippeable = false;
protected boolean isPassable = true;
/* Die neu hinzugefügen Prädikate */
protected boolean isContainer = false;
protected boolean isSource = false;
```

Die zusätzlichen Methoden `grip(World w, String name)` und `ungrip(World w, String name)` werden als Dummy-Methoden implementiert, die lediglich eine Fehlermeldung ausgeben.

7.4.6 RobotSim

Um nun einen einheitlichen Zugriff auf die Erweiterungen der Roboter-Simulationsumgebung zu erreichen muss die RobotSim Klasse noch angepasst werden.

Zunächst muss die Methode `public boolean createWorldObject(String type, String name, int x, int y, int z)` angepasst werden, so dass auch WorldObjects von Typ `DrinkingGlass` und `Tap` instanziert werden können.

Danach müssen die folgenden Methoden in der RobotSim Klasse definiert werden:

- `public boolean grip(String agent, String entity)`
  Diese Methode lässt einen `agent`, zum Beispiel einen Roboter, ein WorldObject `entity` greifen, wenn `entity` greifbar ist und sich agent an der selben Position wie `entity` befindet.

- `public boolean ungrip(String agent, String entity)`
  Diese Methode lässt einen `agent`, zum Beispiel einen Roboter, ein WorldObject `entity` an seiner aktuellen Position absetzen, sofern sich `entity` zum Zeitpunkt des Methodenaufrufs in seinem Greifarm befindet.

- `public boolean fill(String agent, String source, String container)`
  Diese Methode lässt einen `agent`, zum Beispiel einen Roboter, einen Container `container` mit dem Ausstoß einer Quelle `source` füllen. Hierzu müssen sich `agent`, `source` und `container` an der selben Position befinden.

7.4.7 Testen der Erweiterungen

Zum Testen der Erweiterungen kann man das Beispiel `fetchWater(DrinkingGlass Glass1, Tap Tap1)` aus Kapitel 6.8 als Reihe von Anweisungen in die Klasse Main der Roboter-Simulationsumgebung eingeben.

Für dieses Beispiel wird eine Welt mit den Dimensionen [10, 10, 1] erzeugt. Danach werden verschiedene WorldObjects gesetzt:

- Ein `Roboter Robot1` an die Position [0, 0, 0].
- Ein `Tap Tap1` an die Position [3, 3, 0].
- Ein `DrinkingGlass Glass1` an die Position [6, 6, 0].

Danach lässt man `Robot1` zu `Glass1` fahren, es greifen und zu `Tap1` fahren. An `Tap1` angekommen, wird `Glass1` abgesetzt und von `Robot1` mit dem Ausstoß von `Tap1` gefüllt. Danach greift `Robot1` das Objekt `Glass1` erneut und fährt es zur Startposition [0, 0, 0] wo er `Glass1` absetzt.

Als Implementierung der Methode `public static void main(String[] args)` ergibt das den folgenden Quellcode:

```
public static void main(String[] args) {
  // Erzeugung der Simulationsumgebung
  rs = new RobotSim(10, 10, 1);

  // Erzeugen eines Roboters
  rs.createRobot("Robot1", 0, 0, 0);

  // Erzeugen eines Waypoints am Startpunkt [0, 0, 0]
  rs.createWorldObject("Waypoint", "Start", 0, 0, 0);

  // Erzeugen eines Wasserhahns an Position [3, 3, 0]
  rs.createWorldObject("Tap", "Tap1", 3, 3, 0);

  // Positionieren eines Wasserglases an Position [6, 6, 0]
  rs.createWorldObject("DrinkingGlass", "Glass1", 6, 6, 0);
```

```
  // Bewegung des Roboters zu Position [6, 6, 0]
  rs.move("Robot1", "Glass1");

  // Greifen von Glass1
  rs.grip("Robot1", "Glass1");

  // Zu Tap1 bewegen
  rs.move("Robot1", "Tap1");

  // Glass1 abstellen
  rs.ungrip("Robot1", "Glass1");

  // Glass1 mit Ausstoss von Tap1 fuellen
  rs.fill("Robot1", "Tap1", "Glass1");

  // Greifen von Glass1
  rs.grip("Robot1", "Glass1");

  // Zur Ausgangsposition fahren
  rs.move("Robot1", "Start");

  // Glass1 absetzen
  rs.ungrip("Robot1", "Glass1");
}
```

Nach dem Kompilieren und Ausführen der Simulation wurden folgende Ausgaben auf der Konsole ausgegeben:

```
Robot1 has moved to [6,6,0].
Robot1 has gripped Glass1.
Robot1 has moved to [3,3,0].
Robot1 has ungripped Glass1.
Robot1 fills Glass1 at Tap1.
Glass1 is now filled with water.
Robot1 has gripped Glass1.
Robot1 has moved to [0,0,0].
Robot1 has ungripped Glass1.
```

7.5 RTDL Parser

Nachdem die Roboter Simulationsumgebung durch die in Kapitel 7.4 beschriebenen Änderungen erweitert worden ist, ist sie in der Lage das Beispiel `fetchWater(Glass g, Tap t)` aus Kapitel 6.8 als eine Serie von Kommandos der Simulationsumgebung zu beschreiben und auszuführen, wie in Kapitel 7.4.7 gezeigt.

Der letzte noch verbleibende notwendige Schritt zu einer RTDL-fähigen Roboter Simulationsumgebung ist die Verschmelzung von RTDL und Simulationsumgebung. Dies soll durch die Integration eines RTDL Parsers in die Roboter Simulationsumgebung geschehen.

7.5.1 Klasse RtdlParser

Um den RTDL Parser zu implementieren, wurde eine neue Klasse RtdlParser erzeugt. Der Konstruktor `RtdlParser(String filename)` erzeugt ein neues Rtdl-Parser Objekt und erhält als Argument den Pfad zu einer RTDL Taskbeschreibung im XML-Format übergeben.

Intern besteht die Klasse `RtdlParser` aus einer angepassten Instanz eines `XMLStream-Readers`, ähnlich der Klasse `MyXmlParser` aus Kapitel 7.3.9. Die wichtigste Methode des RtdlParsers ist die Methode `public RobotSim parse(RobotSim rs, String robot)`. Diese Methode parst die im Konstruktor übergebene Taskbeschreibung und führt die Aktionen als jener Roboter, der durch das Argument `robot` übergeben wird, in der übergebenen Instanz der Klasse `RobotSim` aus. Im Anschluss an diesen Vorgang wird die modifizierte `RobotSim` Instanz zurückgegeben.

7.5.2 Parsen einer Taskbeschreibung

Der in den Klassen `RtdlParser` und `MyXMLParser` verwendete `XMLStreamReader` agiert ereignisgesteuert. Der Parser durchläuft ein XML-Dokument und erzeugt bei jedem Auftreten eines XML-Elements ein Event. Die für den RTDL Parser wichtigsten Events sind der Beginn `START_ELEMENT` und das Ende `END_ELEMENT` jener XML-Elemente.

Die Validität einer Taskbeschreibung muss derzeit noch durch den Benutzer festgestellt werden, da aus Zeitmangel auf die Implementierung eines XML Validierers in

den RTDL Parser verzichtet wurde.

Der Parser prüft zunächst, ob das Attribut `version` des rtdl Elements den Wert 1.0 hat, wenn nicht wird die Ausführung der Simulationsumgebung beendet.

Als zweites wird beim Betreten eines `task` Elements dessen Attribute `name` ausgewertet und zwischengespeichert.

Im dritten Schritt werden Aktionen, gegeben durch `action` Elemente, ausgewertet. Zunächst wird der Name einer Aktion aus dem `name` Element zwischengespeichert. Danach werden die Argumente der Aktion, gegeben durch `arg` Elemente, ausgewertet und in einem String Array zwischengespeichert. Beim Verlassen eines `action` Elements wird auf Basis des `action` Namens ein entsprechendes Kommando in der übergebenen `RobotSim` Klasse ausgeführt. Die Argumente der `action` werden als Argumente des `RobotSim` Kommandos mit übergeben.

Aus Zeitmangel gegen Ende der Arbeit wurde kein vollständiger RTDL Parser geschrieben, so dass es einige Einschränkungen bei der Verwendung zu beachten gibt:

1. Der RTDL kann keine Elemente verarbeiten, die bisher nicht in diesem Kapitel genannt wurden. Hierzu zählen:
   (a) If-Konstrukte
   (b) While-Schleifen
   (c) Par-Konstrukte
   (d) Retrycount Elemente
   (e) Retry-Blöcke
2. Der RTDL Parser kann keine Entitätsdefinitionen auswerten, weswegen alle Entitäten, die bei der Ausführung von Aktionen beteiligt sind, vor der Ausführung einer Taskbeschreibung in der RobotSim Klasse angelegt sein müssen.
3. Der RTDL Parser kann keine Taskargumente der Taskdefinition auswerten. Das bedeutet, dass Tasksbeschreibungen derzeit keine lokalen Entitätsnamen verwenden dürfen, sondern nur solche, die bereits gültig in der RobotSim Klasse definiert wurden, vgl. Punkt 2.
4. Der RTDL Parser führt keine Validierung von Taskbeschreibungen gegen die RTDL Document Type Definition durch.

5. Der RTDL Parser bricht die Ausführung einer Task nicht ab, wenn eine Aktion innerhalb der Task scheitert, unabhängig von `optional` und `retry` Statements.

6. Der RTDL Parser kann derzeit maximal fünf Argumente je Aktion bei der Ausführung einer Task zwischenspeichern.

8 Fazit

Im Rahmen dieser Masterarbeit wurde mit der Spezifikation der Robotic Task Definition Language (RTDL) der Grundstein für eine neue Sprache der Taskbeschreibung für mobile autonome Systeme geschaffen.

Die RTDL in ihrer jetzigen Version kann nur als Prototyp angesehen werden, da bisher nur eine kleine Implementierung in Form der Roboter-Simulationsumgebung existiert. Dennoch hat die RTDL das Potential durch Weiterentwicklung zu einem einfachen und dennoch effizienten Weg der plattformunabhängigen Beschreibung von Roboter-Tasks zu werden.

Das gleiche gilt für die Roboter-Simulationsumgebung, welche durch ihren einfachen aber strukturierten Aufbau die Simulation von komplexeren Szenarien der Roboter-Simulation in Zukunft ermöglichen könnte.

Aber auch andere interessante Aspekte, die nicht unmittelbar in die Entwicklung der RTDL mit eingeflossen sind, wurden in dieser Arbeit betrachtet. Hier ist zum Beispiel die Joint Architecture for Unmanned Systems (JAUS) zu nennen. JAUS ist ein Standard, der gut geeignet ist, um die Architektur der Steuerungssoftware von unbemannten Systemen beliebiger Art zu entwerfen und umzusetzen.

Auch wenn noch ein weiter Weg vor den Entwicklern von Robotern liegt, bis das Ziel von selbstständig agierenden und mit Menschen kooperierenden Robotern erreicht ist, so ist die Einführung einer plattformunabhängigen Task-Schicht dennoch einen weiterer wichtiger Schritt hin zu diesem Ziel.

A Modellierungsansatz

(Dies ist ein Auszug aus [Tes10] Eine ausführliche Beschreibung der Capabilities befindet sich in [JAU07a, S.17ff.]).

Der Modellierungsansatz von JAUS unterteilt sich in drei verschiedene Bereiche:

1. **Functional Capabilities (FC)**: Was der Kunde/Benutzer tut.
2. **Informational Capabilities (IC)**: Was der Kunde/Benutzer weiß.
3. **Device Groups (DG)**: Womit der Kunde/Benutzer interagiert, die Schnittstellen.

Die funktionalen (FC) und informellen Fähigkeiten (IC) können mit den Geräte-Gruppen (DG) gekoppelt sein, jedoch wird das JAUS Domain Model diese Schnittstellen nicht spezifizieren. Diese Schnittstellen sind offene Standards und werden entweder durch die Joint Technical Architecture (JTA) oder den Original Equipment Manufacturer (OEM) definiert.

A.1 Functional Capability (FC)

Funktionale Fähigkeiten sind eine Menge von Fähigkeiten mit ähnlichem funktionalem Zweck. Das Domain Model unterteilt die funktionalen Fähigkeiten in elf Kategorien, die festlegen, welche Funktionen das unbemannte System ausführen kann. Die funktionalen Fähigkeiten sind jedoch nicht auf die genannten elf Kategorien beschränkt.

1. **Command Capabilities** erstellen Pläne, passen Daten an, treffen Entscheidungen, weisen Aufgaben zu und behandeln unerwartete Ereignisse.
   (a) Vehicle Commander
   (b) Team Commander
   (c) Team-of-Teams Commander
2. **Maneuver Capabilities** kommunizieren mit allen externen maneuvering control devices und können eine Menge von Sensoren beinhalten. Die Maneuver Capability besitzt eine mehrschichtige Menge von intelligenten Komponenten, die verschiedene Stufen von Manöveranweisungen akzeptieren.

   (a) Primitive Maneuver Capability
   (b) Vector Maneuver Capability
   (c) Waypoint Maneuver Capability
   (d) Feature-Following Capability
   (e) Leader-Follower Capability

3. **Navigational Capabilities** sind verantwortlich für die Pfadplanung des unbemannten Systems zwischen zwei oder mehreren geographischen Orten.
   (a) Intelligent Navigation
   (b) Primitive Navigation
   (c) Semi-autonomous Navigation
   (d) Autonomous Navigation

4. **Communication Capabilities** sind verantwortlich für die Behandlung der Kommunikation zwischen dem unbemannten System und verschiedenen OCUs. OCUs sind Operator Control Units, Steuerungseinheiten für unbemannte Systeme.

5. **Payload Capabilities** sind für die Aufrechterhaltung der Kontrolle und der Statusüberwachung der Payloads zuständig.
   (a) Articulation
      i. Primitive Articulation
      ii. Repeating-path Articulation
      iii. Position and Attitude Articulation
   (b) Reconnaissance/Surveillance/Target Acquisition & Identification (RSTA-I)
      i. Primitive RSTA-I
      ii. Semi-autonomous RSTA-I
      iii. Autonomous RSTA-I
   (c) Weapon Payload
      i. Weapon Payload Primitive Capability
      ii. Area Engagement Capability
      iii. Target Engagement Capability

   (d) Environmental Sensing
   (e) Manipulation

6. **Safety Capabilities** sind verantwortlich für die Betriebssicherheit des unbemannten Systems.
   (a) Primitive Safety Capability
   (b) Intelligent Safety Capability
7. **Security Capabilities** verhindern den unautorisierten Zugriff auf das unbemannte System.
8. **Resource Management Capabilities** verwalten die wichtigsten Ressourcen des unbemannten Systems wie zum Beispiel Treibstoff und Energie.
9. **Maintenance Capabilities** sind verantwortlich für die Systemwartung und können mit externen Test-Equipments kommunizieren.
10. **Training Capabilities** bieten Trainingsmöglichkeiten für Anwender an.
11. **Automatic Configuration Capabilities** minimieren die notwendige Konfiguration des unbemannten Systems durch einen Anwender.

A.2 Informational Capability (IC)

Informale Fähigkeiten sind Gruppen von ähnlichen Informationstypen. Es handelt sich hierbei um globale Repositories mit Systemdaten. Sie sind nach ihrem Datentyp benannt und spezifizieren, welche Informationen das unbemannte System über sich selbst besitzt. Die ICs stellen einen kontrollierten Zugang zu diesen Informationen für das gesamte unbemannte System zur Verfügung.

1. **Vehicle Status Capabilities** sind verantwortlich für die Bereitstellung des Fahrzeugstatus an andere Capabilities. Dies umfasst die Parameter: Position, sichtbare und hörbare Daten, Kalender, u.a.
   (a) Platform position and attitude
   (b) Engine Data
2. **World Model Capabilities** unterhalten ein Repository an interessanten Weltobjekten einschließlich freundlicher und feindlicher Einheiten, Hindernisse, Treppenbereiche, Karten, Routen, Wegpunkte und Aktionen.

   (a) Topographic layouts
   (b) Building layouts and blueprints
   (c) Terrain feature information
   (d) Object of interest (last known and predicted location, last known and predicted path)

3. **Library Capabilities** verwalten Referenzmaterial, Prozeduren und Leistungsdaten.
   (a) Mission specific
      i. Procedures and tactics
      ii. Rules of Engagement
      iii. Mission planning (assembly areas, mobility areas, avoidance areas, potential target locations, etc.)
   (b) Platform specific
      i. Maintenance manuals
      ii. Online help
      iii. Training tutorials
      iv. Vehicle capability lists

4. **Log Capabilities** haben die Verantwortlichkeit der Aufzeichnung von Einsatz-, Wartungs- und Trainingsdaten.
   (a) System failures (vehicle and payload)
   (b) Operator commands
   (c) Events during mission execution
   (d) Collaborative information

5. **Time/Date Capabilities** bieten Datums- und Zeitdienste für das unbemannte System an.

A.3 Device Groups (DG)

Gerätegruppen sind Gruppierungen von Sensoren und/oder Aktoren die für ähnliche Funktionen verwendet werden. Spezifische Geräte werden nicht innerhalb von Gerätegruppen definiert.

A.4 Übersichtsgrafik

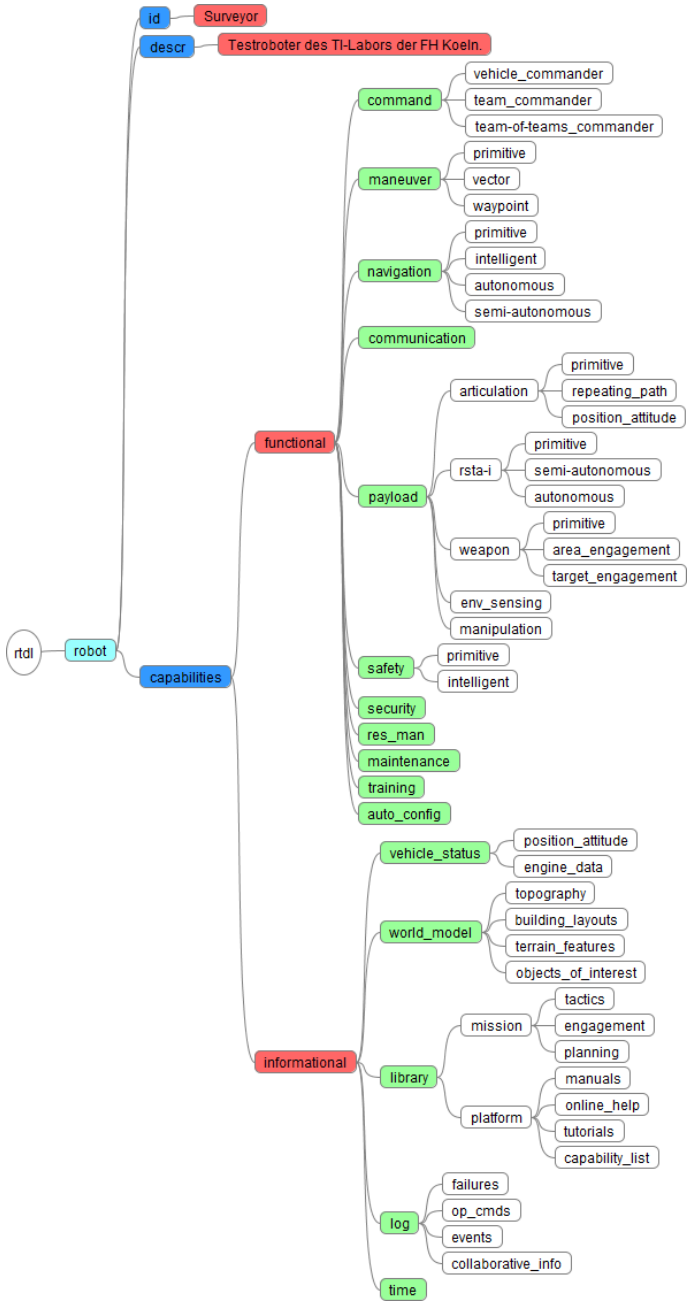

Abbildung 10: RTDL Capabilities

B Liste ausgewählter JAUS-Komponenten

B.1 Command and Control Klasse

B.1.1 System Commander (ID 40)

- Koordination der Subsysteme
- Kommandierung
- Statusabfrage
- Diese Funktion kann von einem Menschen oder einer Maschine ausgefüllt werden. Bei einem Menschen gehören hierzu alle Elemente einer Benutzerschnittstelle, die zur Erbringung der Funktionalitäten dieser Komponente notwendig sind.

B.1.2 Subsystem Commander (ID 32)

- Verantwortlich für Mission Planing
- Kommandierung, also das Übergeben von Anweisungen an das unbemannte System.
- Statusabfrage
- Diese Funktion kann von einem Menschen oder einer Maschine ausgefüllt werden. Bei einem Menschen gehören hierzu alle Elemente einer Benutzerschnittstelle, die zur Erbringung der Funktionalitäten dieser Komponente notwendig sind.

B.2 Communication

B.2.1 Communicator (ID 35)

- Single point of entry für jedes Subsystem
- Beinhaltet alle Data Links zu anderen Subsystemen

B.3 Platform

B.3.1 Global Pose Sensor (ID 38)

- Bestimmung der globalen Pose (Position und Ausrichtung) der Plattform
- Werte werden angegeben in Form von Longitude (Längengrad), Latitude (Breitengrad) und Elevation (Erhöhung).

B.3.2 Local Pose Sensor (ID 41)

- Vergleichbar zum Global Pose Sensor
- Bestimmung der lokalen Pose anhand eines kartesischen Koordinatensystems (x, y, z) angegeben in Relation zu einem definierten Referenzrahmen der Plattform.

B.3.3 Velocity State Sensor (ID 42)

- Ermittelt die aktuelle Geschwindigkeit der Plattform
- Die Geschwindigkeit wird bestimmt als die Summe der Teilgeschwindigkeiten eines definierten Punktes entlang der X-, Y- und Z-Achse des definierten Referenzrahmens der Plattform. Die hierfür verwendeten JAUS-Nachrichten sind:
  - `velocity x`
  - `velocity y`
  - `velocity z`
- Zusätzlich kann die Änderung der Ausrichtung bzw. Deren Winkelgeschwindigkeit über die folgenden Nachrichten angegeben werden:
  - `omega x`
  - `omega y`
  - `omega z`

B.3.4 Primitive Driver (ID 33)

- Beinhaltet alle Basisfunktionen zum Fahren inklusive Hilfsmitteln.
- Der Freiheitsgrad der Fortbewegung entspricht der Bewegung und Drehung entlang der X-, Y- und Z-Achse des definierten Referenzrahmens.
- Striktes open loop Verhalten (Steuerung), wobei keine Geschwindigkeit vorgegeben wird, da diese nicht ohne Geschwindigkeitssensor geregelt werden kann.
- Die genauen Aktoren-Kommandos sind nicht Teil der JAUS-Spezifikation.

B.3.5 Reflexive Driver (ID 43)

- Eine Art Filter vor dem Primitive Driver, bei dem Kommandos gefiltert und modifiziert werden können. Anwendung findet dies immer dann, wenn eine Operation die Sicherheit der Plattform gefährden könnte, zum Beispiel durch zu hohe kommandierte Geschwindigkeiten usw.
- Es werden hierfür oftmals zusätzliche Sensoren an der Plattform angebracht, welche die Ausführung und den Zustand einer Plattform überwachen. Auf Basis dieser Sensordaten werden die gegebenen Kommandos entsprechend modifiziert, dass die Sicherheit der Plattform nicht gefährdet wird.
- Die genaue Art der Sensoren ist nicht Teil der JAUS-Spezifikation.

B.3.6 Global Vector Driver (ID 34)

- Closed loop control (Regelung) von Richtung, Höhe und Geschwindigkeit einer Plattform
- Die Komponente legt anhand des globlen Koordonatensystems die Änderung der Richtung und Geschwindigkeit zum gegebenen Zielpunkt fest und gibt diesen Befehl dann an den Primitive Driver oder Reflexive Driver.

B.3.7 Local Vector Driver (ID 44)

- Gleiche Funktion wie der Global Vector Driver unter Verwendung eines lokalen Koordinatensystems

- Informationen beinhalten:
  - heading (Richtung)
  - pitch (Neigung)
  - roll (Neigung um Längst-Achse)
  - speed (Geschwindigkeit)
- Verwendet Daten von folgenden Sensoren:
  - Local Pose Sensor
  - Velocity State Sensor
- Verwendete Nachrichten
  - `Set Wrench Effort`

B.3.8 Global Waypoint Driver (ID 45)

- Bestimmt den neuen Wrench (Verschiebung) der Plattform aus den folgenden Größen:
  - Ziel-Wegpunkt
  - gewünschte Reisegeschwindigkeit
  - Pose
  - aktuelle Geschwindigkeit
- Ein Wegpunkt setzt sich aus den folgenden Informationen zusammen:
  - Position
  - Ausrichtung
- Wenn die Plattform den letzten Wegpunkt einer Liste erreicht hat, so muss sie in einen Standby Status übergehen.
- Verwendete Nachrichten:
  - `Set Global Waypoint`
  - `Set Travel Speed`
  - `Report Global Pose`
  - `Report Velocity State`

B.3.9 Local Waypoint Driver (ID 46)

- Gleiche Funktionalität, wie der Global Waypoint Driver, jedoch mit einem lokalen Koordinatensystem als Referenz.
- Verwendete Nachrichten:
  - `Set Local Waypoint`
  - `Set Travel Speed`
  - `Report Local Pose`
  - `Report Velocity State`

B.3.10 Global Path Segment Driver (ID 47)

- Closed loop Control (Regelung) von Position und Geschwindigkeit einer Plattform entlang eines festgelegten Pfades
- Der Unterschied zum Waypoint Driver liegt darin, dass der exakte Pfad zwischen zwei Wegpunkten strikt definiert ist und nicht nur aus einer generellen Richtungsangabe besteht.
- Ein Pfad wird durch drei Punkte (P0, P1, P2) und einem Gewichtskoeffizienten (w) bestimmt. Hierbei ist P0 oft der Startpunkt und P2 der Zielpunkt. Der Pfad zwischen P0 und P2 wird nun so berechnet, dass der Weg zwischen P0 und P2 entlang dem gewichteten Punkt P1 verläuft. Ist w=0, fällt der Punkt P1 aus der Gleichung heraus und der Pfad ist die direkte Verbindung zwischen P0 und P2. Wird w¿0 gebildet, so verformt sich der Pfad zu einer Kurve die je nach Größe von w in Richtung von P1 gekrümmt wird.

B.3.11 Local Path Segment Driver (ID 38)

- Laut Beschreibung in JAUS, gibt es keinen erkennbaren Unterschied zum Global Path Segment Driver.

B.4 Environment Sensor Components

B.4.1 Visual Sensor (ID 37)

- Steuert eine oder mehrere Kameras und deren Blickrichtungen

- Zuständig für die Kamera-Einstellungen, Bildformate, Audio usw.
- Jeder Kamera wird ein lokales Kooridatensystem zugeordnet.

B.4.2 Range Sensor (ID 50)

- Steuert einen oder mehrere Abstandssensoren pro Komponente.
- Abstandsinformationen relativ zum lokalen Koordinatensystem der Plattform.
- Abstandswerte müssen von den Sensoren in einer festgelegten Genauigkeit ermittelt werden.

B.4.3 World Model Vector Knowledge Store (ID 61)

- Zentrales Repository für vektorformatierte geospatiale Daten für die Elemente aller Ebenen der JAUS-Topologie.
- Speichern und Verteilen der Daten im Vektorformat. Vektorisierte Positions- und Richtungsangaben sind speicherplatzeffizient.
- Objekte können aus Punkten, Linien, Polylinien und Polygone bestehen.
- Objekte können einen Region Buffer besitzen. Das ist eine Umgebung (angegeben in Metern) um ein Objekt herum, die es einer Plattform erlaubt sich in die Nähe eines Objekts zu positionieren um dort Operationen ausführen zu können.
- Verwendete Nachrichten:
  - `Create Vector Knowledge Store Objects`
  - `Set Vector Knowledge Store Feature Class Metadata`
  - `Delete vector Knowledge Store Objects`
  - `Query Vector Knowledge Store Objects`
  - `Query Vector Knowledge Store Feature Class Metadata`
  - `Query Vector Knowledge Store Bounds`
  - `Terminate Vector Knowledge Store Data Transfer`
  - `Report Vector Knowledge Store Object(s) Creation`
  - `Report Vector Knowledge Store Feature Class Metadata`

- `Report Vector Knowledge Store Objects`
- `Report Vector Knowledge Store Bounds`
- `Report Vector Knowledge Store Data Transfer Termination`

B.5 Planning Components

B.5.1 Mission Planner (keine ID)

- Missionplaner ist in der vorliegenden JAUS-Version v3.3 der Referenzarchitektur noch nicht definiert.

B.5.2 Mission Spooler (ID 36)

- Zentraler Ort für Missionspläne während der Ausführung oder kurz davor.
- Aufgabe ist die Aufteilung von Planbestandteilen in einzelne Elmente und deren Auslieferungen an die entsprechenden Maschinenteile.
- Eine Mission ist definiert als eine Menge von JAUS-Nachrichten.

C Liste ausgewählter JAUS-Nachrichten

C.1 Platform Subgroup Command Class (Code 0400-05FF)

- Code 0405h: `Set Wrench Effort`
- Code 0407h: `Set Global Vector`
- Code 0408h: `Set Local Vector`
- Code 040A: `Set Travel Speed`
- Code 040Ch: `Set Global Waypoint`
- Code 040Dh: `Set Local Waypoint`
- Code 040Fh: `Set Global Path Segment`
- Code 0410h: `Set Local Path Segment`

C.2 Platform Subgroup Query Class (Code 2400-25FF)

- Code 2402h: `Query Global Pose`
- Code 2403h: `Query Local Pose`
- Code 2404h: `Query Velocity State`
- Code 2405h: `Query Wrench Effort`
- Code 2407h: `Query Global Vector`
- Code 2408h: `Query Local Vector`
- Code 240Ah: `Query Travel Speed`
- Code 240Ch: `Query Global Waypoint`
- Code 240Dh: `Query Local Waypoint`
- Code 240Fh: `Query Global Path Segment`
- Code 2410h: `Query Local Path Segment`

C.3 Platform Subgroup Inform Class (Code 4400-45FF)

- Code 4402h: `Report Global Pose`
- Code 4403h: `Report Local Pose`
- Code 4404h: `Report Velocity State`
- Code 4405h: `Report Wrench Effort`
- Code 4407h: `Report Global Vector`
- Code 4408h: `Report Local Vector`
- Code 440Ah: `Report Travel Speed`
- Code 440Ch: `Report Global Waypoint`
- Code 440Dh: `Report Local Waypoint`
- Code 440Fh: `Report Global Path Segment`
- Code 4410h: `Report Local Path Segment`

D Quelltexte

D.1 DrinkingGlass.java

```
1  /**
2   * DrinkingGlass.java
3   * A drinking glass container.
4   *
5   * @author Marcus Teske
6   */
7  package rtdl;
8
9  public class DrinkingGlass extends WorldObject {
10
11   private String contentType;
12
13   /**
14    * Constructor
15    * @param objectName Name of the drinking glass.
16    */
17   DrinkingGlass(String objectName) {
18     super.setObjectName(objectName);
19     super.isContainer = true;
20     super.isGrippeable = true;
21     this.contentType = new String();
22   }
23
24   /**
25    * Fill the drinking glass.
26    * @param w The World the takes place in.
27    * @param source The source that fills this container with its output
        .
28    * @return True if the operation was successful, false otherwise.
29    */
30   @Override
31   public boolean fillContainer(World w, String source) {
32     if (w.getWorldObjectByName(source).isSource) {
33       Source s = (Source) w.getWorldObjectByName(source);
34       this.contentType = s.getSourceType();
35     }
36     System.out.println(this.getObjectName() + " is now filled with "
37             + contentType + ".");
38     return true;
39   }
```

```
40
41   /**
42    * Emptying the drinking glass.
43    * @return True if the operation was successful, false otherwise.
44    */
45   @Override
46   public boolean emptyContainer() {
47     this.contentType = "";
48     return true;
49   }
50 }
```

Listing 6: DrinkingGlass.java

D.2 Main.java

```
1 /**
2  * Main.java
3  * A programm that instantiates a RobotSim object and let's the
       programmer play
4  * around with it.
5  *
6  * @author Marcus Teske
7  */
8 package rtdl;
9
10 public class Main {
11   // Simulation parameters
12
13
14   // Anfang Attribute
15   private static MyXmlParser mxp;
16   private static RobotSim rs;
17   private static RtdlParser rtdlp;
18   // Ende Attribute
19
20
21
22   // Anfang Methoden
23   public static void main(String[] args) {
24     // Example for Import of World
25         try {
26             mxp = new MyXmlParser("C:\\temp\\world.xml");
```

```
27              rs = mxp.parse(rs);
28          } catch (Exception e) {
29              System.err.println("ERROR beim Parsen!");
30          }
31
32
33      // Erzeugen der Simulationsumgebung
34 //     rs = new RobotSim(10, 10, 1);
35
36      // Erzeugen eines Roboters
37 //     rs.createRobot("Robot1", 0, 0, 0);
38
39      // Erzeugen eines Wasserhahns an Position [3, 3, 0]
40 //     rs.createWorldObject("Tap", "Tap1", 3, 3, 0);
41
42      // Positionieren eines Wasserglases an Position [6, 6, 0]
43 //     rs.createWorldObject("DrinkingGlass", "Glass1", 6, 6, 0);
44
45      // Task parsen und ausfÃ¼hren.
46      try {
47        rtdlp = new RtdlParser("C:\\temp\\fetchWater.xml");
48        rs = rtdlp.parse(rs, "Robot1");
49      } catch (Exception e) {
50
51        System.err.println("ERROR beim Parsen:\n" + e.toString());
52      }
53
54      // Bewegung des Roboters zur Position [6, 6, 0]
55 //         rs.move("Robot1", "Glass1");
56 //
57 //         rs.grip("Robot1", "Glass1");
58 //
59 //         rs.move("Robot1", "Tap1");
60 //
61 //         rs.ungrip("Robot1", "Glass1");
62 //
63 //         rs.fill("Robot1", "Tap1", "Glass1");
64 //
65 //         rs.grip("Robot1", "Glass1");
66 //
67 //         rs.move("Robot1", "Start");
68 //
69 //         rs.ungrip("Robot1", "Glass1");
70
```

```
71      // Example for saving the World.
72 //         try {
73 //             echo(rs.toXmlString());
74 //         } catch (NullPointerException e) {
75 //         }
76 
77   }
78 
79   private static void echo(String s) {
80     System.out.println(s);
81   }
82   // Ende Methoden
83 }
```

Listing 7: Main.java

D.3 MyXmlparser.java

```
1 /**
2  * MyXmlParser.java
3  * http://www.iks.hs-merseburg.de/~uschroet/Literatur/Java_Lit/
       JAVA_Insel/javainsel_15_004.htm#mjd0dbb53e72e3b53b011e15c20054078f
4  *
5  * @author Marcus Teske
6  */
7 package rtdl;
8 
9 import java.io.*;
10 import javax.xml.stream.XMLInputFactory;
11 import javax.xml.stream.XMLStreamReader;
12 import javax.xml.stream.XMLStreamException;
13 import javax.xml.stream.XMLStreamConstants;
14 
15 public class MyXmlParser {
16 
17   XMLStreamReader parser;
18 
19   /* Parser variables */
20   private int sizeX = 0; // holds the size along x of the world.
21   private int sizeY = 0; // holds the size along x of the world.
22   private int sizeZ = 0; // holds the size along x of the world.
23   private boolean inObject = false;
```

```
24   private String name = ""; // holds the name of the current
         WorldObject.
25   private String type = ""; // holds the type of the current
         WorldObject.
26   private int x = 0; // holds the x-coordinate of the current
         WorldObject.
27   private int y = 0; // holds the y-coordinate of the current
         WorldObject.
28   private int z = 0; // holds the z-coordinate of the current
         WorldObject.
29
30   MyXmlParser(String filename)
31           throws FileNotFoundException, XMLStreamException {
32     parser = XMLInputFactory.newInstance().createXMLStreamReader(
33             new FileInputStream(filename));
34   }
35
36   public RobotSim parse(RobotSim rs) throws XMLStreamException {
37
38     while (parser.hasNext()) {
39       switch (parser.getEventType()) {
40         case XMLStreamConstants.START_DOCUMENT:
41           break;
42
43         case XMLStreamConstants.END_DOCUMENT:
44           parser.close();
45           break;
46
47         case XMLStreamConstants.NAMESPACE:
48           break;
49
50         case XMLStreamConstants.START_ELEMENT:
51           if (parser.getLocalName().equals("sizex")) {
52             sizeX = Integer.parseInt(parser.getElementText());
53           }
54           if (parser.getLocalName().equals("sizey")) {
55             sizeY = Integer.parseInt(parser.getElementText());
56           }
57           if (parser.getLocalName().equals("sizez")) {
58             sizeZ = Integer.parseInt(parser.getElementText());
59           }
60
61           if (parser.getLocalName().equals("object")) {
62             System.out.println("DEBUG: inside object");
```

```
63              inObject = true;
64            }
65
66            if (inObject == true) {
67              if (parser.getLocalName().equals("name")) {
68                name = parser.getElementText();
69              } else if (parser.getLocalName().equals("type")) {
70                type = parser.getElementText();
71              } else if (parser.getLocalName().equals("x")) {
72                x = Integer.parseInt(parser.getElementText());
73              } else if (parser.getLocalName().equals("y")) {
74                y = Integer.parseInt(parser.getElementText());
75              } else if (parser.getLocalName().equals("z")) {
76                z = Integer.parseInt(parser.getElementText());
77              }
78            }
79            break;
80
81          case XMLStreamConstants.CHARACTERS:
82            break;
83
84          case XMLStreamConstants.END_ELEMENT:
85            // Check if sizes are set and create RobotSim
86            if (rs == null && sizeX > 0 && sizeY > 0 && sizeZ > 0) {
87              rs = new RobotSim(sizeX, sizeY, sizeZ);
88            }
89            if (inObject == true && parser.getLocalName().equals("object"
                  )) {
90              System.out.println("DEBUG(RobotSim): createWorldObject(" +
                    type
91                        + ", " + name + ", " + x + ", " + y + ", " + z + ")
                          ");
92              rs.createWorldObject(type, name, x, y, z);
93              inObject = false;
94            }
95            break;
96          default:
97            break;
98        }
99        parser.next();
100     }
101     return rs;
102   }
103 }
```

Listing 8: MyXmlparser.java

D.4 Robot.java

```
1  /**
2   * Robot.java
3   * An extension of the WorldObject class that resembles a virtual Robot
        that is
4   * able to act inside a virtual World.
5   *
6   * @author Marcus Teske
7   */
8  package rtdl;
9
10 public class Robot extends WorldObject {
11
12   private int facingDirection;
13   private static final int NORTH = 0;
14   private static final int EAST = 900;
15   private static final int SOUTH = 1800;
16   private static final int WEST = 2700;
17   private static final int UP = 10000;
18   private static final int DOWN = -10000;
19   private WorldObject arm;
20
21   /**
22    * Constructor
23    * @param objectName Name of the robot.
24    * @param facingDirection Initial facing direction.
25    */
26   Robot(String objectName, int facingDirection) {
27     super.setObjectName(objectName);
28     super.canMove = true;
29     this.facingDirection = facingDirection;
30   }
31
32   /**
33    * Constructor
34    * @param objectName Name of the robot.
35    */
36   Robot(String objectName) {
```

```
37      super.setObjectName(objectName);
38      this.facingDirection = NORTH;
39    }
40
41    /**
42     * The direction the robot is facing at.
43     * @return Direction the robot is facing at.
44     */
45    public int getFacingDirection() {
46      return this.facingDirection;
47    }
48
49    /**
50     * Turn the robot to a given facing direction.
51     * @param direction new facing direction.
52     */
53    public void setFacingDirection(int direction) {
54      this.facingDirection = direction;
55    }
56
57    /**
58     * Turning the robot to the right about 90 degree.
59     */
60    public void turnRight() {
61      this.facingDirection += 900;
62      this.facingDirection = this.facingDirection % 3600;
63    }
64
65    /**
66     * Turning the robot to the left about 90 degree.
67     */
68    public void turnLeft() {
69      this.facingDirection -= 900;
70      this.facingDirection = this.facingDirection % 3600;
71    }
72
73    /**
74     * Get the WorldPosition of the Robot.
75     * @param w The world.
76     * @return The current position of the robot.
77     */
78    public WorldPosition getPosition(World w) {
79      return w.getWorldLocationByWorldObjectName(super.getObjectName()).
          getPos();
```

```
80    }
81
82    /**
83     * Determins the position next to the robots front.
84     * @param w The world in which the action takes place
85     * @return The position next to the robots front.
86     */
87    public WorldPosition lookingAt(World w) {
88      WorldPosition pos = new WorldPosition(
89              this.getPosition(w).getX(),
90              this.getPosition(w).getY(),
91              this.getPosition(w).getZ());
92      switch (this.facingDirection) {
93        case NORTH:
94          pos.setY(pos.getY() + 1);
95          break;
96        case EAST:
97          pos.setX(pos.getX() + 1);
98          break;
99        case SOUTH:
100         pos.setY(pos.getY() - 1);
101         break;
102       case WEST:
103         pos.setX(pos.getX() - 1);
104         break;
105       case UP:
106         pos.setZ(pos.getZ() + 1);
107         break;
108       case DOWN:
109         pos.setZ(pos.getZ() - 1);
110         break;
111     }
112     return pos;
113   }
114
115   /**
116    * Moves this robot one step forward along his facing direction, if
         possible.
117    * @param w The World where the action takes place.
118    * @return true if the operation was successfull, false if not.
119    */
120   public boolean moveForward(World w) {
121     WorldPosition pos = this.lookingAt(w);
122     if (w.isPassable(pos)) {
```

```
123         if (w.setWorldObject(this, pos)) {
124           return true;
125         } else {
126           System.err.println("ERROR: moveForward() failed!");
127           return false;
128         }
129       }
130       System.err.println("WARNING: Cannot move to [" + pos.getX() + ","
131               + pos.getY() + "," + pos.getZ() + "]: Location impassable."
                  );
132       return false;
133     }
134
135     /**
136      * Turn the robot so he faces north.
137      */
138     public void faceNorth() {
139       this.facingDirection = NORTH;
140     }
141
142     /**
143      * Turn the robot so he faces east.
144      */
145     public void faceEast() {
146       this.facingDirection = EAST;
147     }
148
149     /**
150      * Turn the robot so he faces south.
151      */
152     public void faceSouth() {
153       this.facingDirection = SOUTH;
154     }
155
156     /**
157      * tun the robot so he faces west.
158      */
159     public void faceWest() {
160       this.facingDirection = WEST;
161     }
162
163     /**
164      * Turn the robot so he faces up.
165      */
```

```
166   public void faceUp() {
167     this.facingDirection = UP;
168   }
169
170   /**
171    * Turn the robot so he faces down.
172    */
173   public void faceDown() {
174     this.facingDirection = DOWN;
175   }
176
177   /**
178    * Returns a greeting message.
179    * @return Greeting message
180    */
181   public String greeting() {
182     return this.getObjectName() + ": Hello I am " + this.getObjectName
          ()
183             + "!";
184   }
185
186   /**
187    * Checks whether the next position the robot is facing at is
          passable.
188    * @param w The World.
189    * @return True if the position the robot is facing at is passable,
          otherwise
190    * false.
191    */
192   public boolean aheadPassable(World w) {
193     WorldPosition wp = this.lookingAt(w);
194     if (!w.locationValid(wp)) {
195       System.err.println("ERROR: " + this.getObjectName() + " cannot
            move to ["
196               + wp.getX() + "," + wp.getY() + "," + wp.getZ() +
197               "]: Location invalid!");
198       return false;
199     }
200     return w.isPassable(wp);
201   }
202
203   /**
204    * Lets the Robot walk along a simple path to a new location. The
          Robot
```

```
205     * follows the x-axis until it fits, then the y-axis until it fits
            and
206     * finally the z-axis until it fits.
207     * @param w The World in which the operation takes place.
208     * @param x x-coordinate
209     * @param y y-coordinate
210     * @param z z-coordinate
211     * @return True if the movement was successful, false if not.
212     */
213    @Override
214    public boolean move(World w, int x, int y, int z) {
215      if (!w.locationValid(x, y, z)) {
216        System.err.println("ERROR: " + this.getObjectName()
217                + " cannot move to [" + x + "," + y + "," + z
218                + "]: Location invalid!");
219        return false;
220      }
221      WorldPosition tp = new WorldPosition(x, y, z); //target
            WorldPosition
222      WorldPosition cp = this.getPosition(w); // current WorldPosition
223
224      if (cp.equals(tp)) {
225        return true;
226      }
227      for (long i = cp.getOffsetX(tp); i != 0;) { // Until x-coordinate
            matches.
228        if (i > 0) { // Target east
229          if (!this.moveEast(w)) {
230            break;
231          }
232          i--;
233        } else { // Target west
234          if (!this.moveWest(w)) {
235            break;
236          }
237          i++;
238        }
239      }
240      for (long i = cp.getOffsetY(tp); i != 0;) { // Until y-coordinate
            matches.
241        if (i > 0) { // Target north
242          if (!this.moveNorth(w)) {
243            break;
244          }
```

```
245         i--;
246       } else { // Target south
247         if (!this.moveSouth(w)) {
248           break;
249         }
250         i++;
251       }
252     }
253     for (long i = cp.getOffsetZ(tp); i != 0;) { // Until z-coordinate
          matches.
254       if (i > 0) { // Target above
255         if (!this.moveUp(w)) {
256           break;
257         }
258         i--;
259       } else { // Target below
260         if (!this.moveDown(w)) {
261           break;
262         }
263         i++;
264       }
265     }
266     cp = this.getPosition(w);
267     return true;
268   }
269
270   /**
271    * Gets the next possible step for the Robot on his way to a target
         position.
272    * @param w World
273    * @param x x-ccordinate of target
274    * @param y y-ccordinate of target
275    * @param z z-ccordinate of target
276    * @return Position of the next possible step
277    */
278   public WorldPosition nextStep(World w, int x, int y, int z) {
279     WorldPosition cp = this.getPosition(w);
280
281     if (cp.getX() == x && cp.getY() == y && cp.getZ() == z) {
282       return cp;
283     }
284     if (cp.getX() < x) {
285       this.faceEast();
286     } else if (cp.getX() > x) {
```

```
287         this.faceWest();
288       }
289       if (w.locationValid(this.lookingAt(w)) &&
290               w.isPassable(this.lookingAt(w))) {
291         return this.lookingAt(w);
292       } else {
293         if (cp.getY() < y) {
294           this.faceNorth();
295         } else if (cp.getY() > y) {
296           this.faceSouth();
297         }
298         if (w.locationValid(this.lookingAt(w)) &&
299                 w.isPassable(this.lookingAt(w))) {
300           return this.lookingAt(w);
301         } else {
302           if (cp.getZ() < z) {
303             this.faceUp();
304           } else if (cp.getZ() > z) {
305             this.faceDown();
306           }
307           if (w.locationValid(this.lookingAt(w)) &&
308                   w.isPassable(this.lookingAt(w))) {
309             return this.lookingAt(w);
310           } else {
311             return null;
312           }
313         }
314
315       }
316     }
317
318     /**
319      * Tries to move the robot one position to the north.
320      * @param w World.
321      * @return True if the operation was successful, false otherwise.
322      */
323     public boolean moveNorth(World w) {
324       this.faceNorth();
325       return this.moveForward(w);
326     }
327
328     /**
329      * Tries to move the robot one position to the south.
330      * @param w World.
```

```
331     * @return True if the operation was successful, false otherwise.
332     */
333    public boolean moveSouth(World w) {
334      this.faceSouth();
335      return this.moveForward(w);
336    }
337
338    /**
339     * Tries to move the robot one position to the east.
340     * @param w World.
341     * @return True if the operation was successful, false otherwise.
342     */
343    public boolean moveEast(World w) {
344      this.faceEast();
345      return this.moveForward(w);
346    }
347
348    /**
349     * Tries to move the robot one position to the west.
350     * @param w World.
351     * @return True if the operation was successful, false otherwise.
352     */
353    public boolean moveWest(World w) {
354      this.faceWest();
355      return this.moveForward(w);
356    }
357
358    /**
359     * Tries to move the robot one position up.
360     * @param w World.
361     * @return True if the operation was successful, false otherwise.
362     */
363    public boolean moveUp(World w) {
364      this.faceUp();
365      return this.moveForward(w);
366    }
367
368    /**
369     * Tries to move the robot one position down.
370     * @param w World.
371     * @return True if the operation was successful, false otherwise.
372     */
373    public boolean moveDown(World w) {
374      this.faceUp();
```

```
375     return this.moveForward(w);
376   }
377
378   /**
379    * Gets the WorldObject the Robot has in his arm.
380    * @return Object in the robot's arm.
381    */
382   @Override
383   public WorldObject getArmPayload() {
384     return this.arm;
385   }
386
387   /**
388    * The Robot grips an grippeable WorldObject.
389    * @param w The world the action takes place in.
390    * @param name Name of the WorldObject to grip.
391    * @return True if the operation was successful, false otherwise.
392    */
393   @Override
394   public boolean grip(World w, String name) {
395     WorldPosition myPos = this.getPosition(w);
396     WorldPosition oPos = w.getWorldLocationByWorldObjectName(
397             name).getPos();
398     // If the WorldObject is at my position
399     if (myPos.equals(oPos)) {
400       if (w.getWorldObjectByName(name).isGrippeable) {
401         this.arm = w.takeObjectByName(name);
402         System.out.println(this.getObjectName() + " has gripped "
403                 + name + ".");
404         return true;
405       }
406     }
407
408     System.err.println("ERROR: Grip failed: Object " + name
409             + " not found at position [" + myPos.getX() + "," + myPos.
                getY()
410             + "," + myPos.getZ() + "]!");
411     return false;
412   }
413
414   /**
415    * The Robot puts down the named WorldObject a his current position.
416    * @param w The World the action takes place in.
417    * @param name Name of the WorldObject to put down
```

```
418     * @return True if the operation is successful, false otherwise.
419     */
420    @Override
421    public boolean ungrip(World w, String name) {
422      if (this.arm != null) {
423        if (this.arm.getObjectName().equals(name)) {
424          System.out.println(this.getObjectName() + " has ungripped "
425                  + name + ".");
426          return w.setWorldObject(this.arm, this.getPosition(w));
427        }
428      }
429      System.err.println("ERROR: Ungrip failed: " + this.getObjectName()
430              + " does not carry object " + name + "!");
431      return false;
432    }
433  }
```

Listing 9: Robot.java

D.5 RobotSim.java

```
1  /**
2   * RobotSim.java
3   * A simulation environment for virtual robots. It consists of exactly
        one World
4   * and offers methods that capsulate the operations on the World and it
        's
5   * contained WorldObjects.
6   *
7   * @author Marcus Teske
8   */
9  package rtdl;
10
11 //import com.thoughtworks.xstream.*;
12 public class RobotSim {
13
14   private World w; // The World where everything takes place.
15
16   /**
17    * Constructor to instanciate a RobotSimulation.
18    * @param x Amount of WorldLocation along the X-axis, starting at 0.
19    * @param y Amount of WorldLocation along the Y-axis, starting at 0.
20    * @param z Amount of WorldLocation along the Z-axis, starting at 0.
```

```
21     */
22    RobotSim(int x, int y, int z) {
23      this.w = new World(x, y, z);
24    }
25
26    /**
27     * Creates an Object with a specified type and name at a specified
          location.
28     * @param type Type of the Object.
29     * @param name Name of the Object.
30     * @param x Place at this x-ccordinate.
31     * @param y Place at this y-ccordinate.
32     * @param z Place at this z-ccordinate.
33     * @return true if the operation was successful, false if not.
34     */
35    public boolean createWorldObject(String type, String name, int x, int
          y,
36            int z) {
37      if (!w.contains(name)) {
38        if (this.w.locationValid(x, y, z)) {
39          if ("Robot".equals(type)) {
40            return this.w.setWorldObject(new Robot(name), x, y, z);
41  //        } else if ("RfidTag".equals(type)) {
42  //          return this.w.setWorldObject(new RfidTag(name), x, y, z);
43          } else if ("Wall".equals(type)) {
44            return this.w.setWorldObject(new Wall(name), x, y, z);
45          } else if ("DrinkingGlass".equals(type)) {
46            return this.w.setWorldObject(new DrinkingGlass(name), x, y,
47                    z);
48          } else if ("Tap".equals(type)) {
49            return this.w.setWorldObject(new Tap(name), x, y, z);
50          } else if ("Waypoint".equals(type)) {
51            return this.w.setWorldObject(new Waypoint(name), x, y, z);
52          }
53          return false;
54        }
55        return false;
56      } else {
57        System.err.println("ERROR: Cannot create WorldObject " + name
58                + "! An Object by that name already exists!");
59        return false;
60      }
61    }
62
```

```
63   /**
64    * Creates a WorldObject of a specific type and with a specific name
          at the
65    * World's origin.
66    * @param type Type of the WorldObject [Robot]
67    * @param name Name of the WorldObject.
68    * @return true if the operation was successful, false if not.
69    */
70   public boolean createWorldObject(String type, String name) {
71     if ("Robot".equals(type)) {
72       return this.w.setWorldObject(new Robot(name), 0, 0, 0);
73     } else {
74       return false;
75     }
76   }
77
78   /**
79    * Simple Wrapper to create a robot at the world's origin.
80    * @param RobotName Name of the Robot.
81    * @return true if the operation was successful, false if not.
82    */
83   public boolean createRobot(String RobotName) {
84     return this.createWorldObject("Robot", RobotName);
85   }
86
87   /**
88    * Wrapper to create a Robot with a unique name at a specified
          position.
89    * @param RobotName Name of the Robot.
90    * @param x x-coordinate.
91    * @param y x-coordinate.
92    * @param z x-coordinate.
93    * @return true if the operation was successful, false if not.
94    */
95   public boolean createRobot(String RobotName, int x, int y, int z) {
96     return this.createWorldObject("Robot", RobotName, x, y, z);
97   }
98
99   /**
100   * Gets an WorldObject by it's name
101   * @param name Name of the WorldObject.
102   * @return the WorldObject object.
103   */
104  public WorldObject getObject(String name) {
```

```
105     return this.w.getWorldObjectByName(name);
106   }
107
108   /**
109    * Simple Wrapper for getting robots.
110    * @param name Name of the robot
111    * @return the robot object.
112    */
113   public Robot getRobot(String name) {
114     return (Robot) this.w.getWorldObjectByName(name);
115   }
116
117   /**
118    * Removes an Object from the World.
119    * @param name Name of the Object.
120    * @return true if the operation was successful, false otherwise.
121    */
122   public boolean removeObject(String name) {
123     return this.w.removeWorldObjectByName(name);
124   }
125
126   /**
127    * Gets the Position of an WorldObject as WorldPosition object.
128    * @param name name of the WorldObject.
129    * @return the position.
130    */
131   public WorldPosition getPosition(String name) {
132     return this.w.getWorldLocationByWorldObjectName(name).getPos();
133   }
134
135   /**
136    * Gets the Position of an WorldObject as String [x,y,z].
137    * @param name name of the WorldObject.
138    * @return the position.
139    */
140   public String getPositionString(String name) {
141     WorldPosition wp = this.getPosition(name);
142
143     return "Position of " + name + ": [" + wp.getX() + "," + wp.getY()
          + ","
144               + wp.getZ() + "]";
145   }
146
147   /**
```

```
148     * Places an WorldObject directly at the specified location.
149     * @param name Name of the WorldObject.
150     * @param x x-ccordinate
151     * @param y y-ccordinate
152     * @param z z-ccordinate
153     * @return true if the operation was successfull, false if not.
154     */
155    public boolean place(String name, int x, int y, int z) {
156      return this.w.setWorldObject(this.getObject(name), x, y, z);
157
158
159    }
160
161    /**
162     * Wrapper that lets an WorldObject move along a simple path to a new
163     * location.
164     * The WorldObject at first moves along the x-axis, then along the y-
            axis
165     * and at last along the z-axis.
166     * @param name Name of the WorldObject to move
167     * @param x x-ccordinate
168     * @param y y-coordinate
169     * @param z z-coordinate
170     * @return true if the operation was successfull, false if not.
171     */
172    public boolean move(String name, int x, int y, int z) {
173      if (!this.w.contains(name)) {
174        System.err.println("ERROR: Move failed: Object " + name
175                + " not found!");
176        return false;
177      }
178      if (this.getObject(name).move(this.w, x, y, z)) {
179        System.out.println(name + " has moved to [" + x + "," + y + "," +
               z
180                + "].");
181        return true;
182      }
183      System.err.println("ERROR: Moving to [" + x + "," + y + "," + z
184              + "] failed!");
185      return false;
186    }
187
188    /**
```

```
189     * Wrapper that lets an WorldObject name walk along a simple path to
            the
190     * location of another WorldObject destination.
191     * @param name Name of WorldObject to move.
192     * @param destination Name of the destination WorldObject.
193     * @return true if the operation was successfull, false if not.
194     */
195    public boolean move(String name, String destination) {
196      WorldLocation destLoc;
197      destLoc = w.getWorldLocationByWorldObjectName(destination);
198      if (destLoc == null) {
199        System.err.println("ERROR: Moving to " + destination + " failed!"
200                + " No such WorldObject.");
201        System.exit(1);
202      }
203      int x = destLoc.getX();
204      int y = destLoc.getY();
205      int z = destLoc.getZ();
206
207      return this.move(name, x, y, z);
208    }
209
210    /**
211     * Moves an WorldObject one Position to the North. Only Robots.
212     * @param name Name of The Object.
213     * @param steps The amount of steps to move.
214     * @return true if the operation was successfull, false if not.
215     */
216    public boolean moveNorth(String name, int steps) {
217      if (this.getObject(name).getObjectType().equals("Robot")) {
218        Robot r = (Robot) this.getObject(name);
219        while (steps > 0) {
220          steps--;
221          if (!r.moveNorth(w)) {
222            System.err.println("ERROR: MovingNorth failed!");
223            return false;
224          }
225        }
226        return true;
227      }
228      return false;
229    }
230
231    /**
```

```
232     * Moves an WorldObject several Position to the South. Only Robots.
233     * @param name Name of The Object.
234     * @param steps The amount of steps to move.
235     * @return true if the operation was successfull, false if not.
236     */
237    public boolean moveSouth(String name, int steps) {
238      if (this.getObject(name).getObjectType().equals("Robot")) {
239        Robot r = (Robot) this.getObject(name);
240        while (steps > 0) {
241          steps--;
242          if (!r.moveSouth(w)) {
243            System.err.println("ERROR: MovingSouth failed!");
244            return false;
245          }
246        }
247        return true;
248      }
249      return false;
250    }
251
252    /**
253     * Moves an WorldObject one Position to the East. Only Robots.
254     * @param name Name of The Object.
255     * @param steps The amount of steps to move.
256     * @return true if the operation was successfull, false if not.
257     */
258    public boolean moveEast(String name, int steps) {
259      if (this.getObject(name).getObjectType().equals("Robot")) {
260        Robot r = (Robot) this.getObject(name);
261        while (steps > 0) {
262          steps--;
263          if (!r.moveEast(w)) {
264            System.err.println("ERROR: MovingEast failed!");
265            return false;
266          }
267        }
268        return true;
269      }
270      return false;
271    }
272
273    /**
274     * Moves an WorldObject one Position to the West. Only Robots.
275     * @param name Name of The Object.
```

```
276     * @param steps The amount of steps to move.
277     * @return true if the operation was successful, false if not.
278     */
279    public boolean moveWest(String name, int steps) {
280      if (this.getObject(name).getObjectType().equals("Robot")) {
281        Robot r = (Robot) this.getObject(name);
282        while (steps > 0) {
283          steps--;
284          if (!r.moveWest(w)) {
285            System.err.println("ERROR: MovingWest failed!");
286            return false;
287          }
288        }
289        return true;
290      }
291      return false;
292    }
293
294    /**
295     * Lets an agent grip an grippeable WorldObject entity at his current
296     * position.
297     * @param agent Robot to grip the entity.
298     * @param entity Grippeable WorldObject to grip.
299     * @return True if the opeartion is successful, false if not.
300     */
301    public boolean grip(String agent, String entity) {
302      if (!this.w.contains(agent)) {
303        System.err.println("ERROR: Grip failed: Agent " + agent
304                + " not found!");
305        return false;
306      }
307      if (!this.w.contains(entity)) {
308        System.err.println("ERROR: Grip failed: Entity " + entity
309                + " not found!");
310        return false;
311      }
312      if (this.getObject(agent).grip(this.w, entity)) {
313        return true;
314      }
315      return false;
316    }
317
318    /**
319     * Lets an agent put down the named WorldObject entity at his current
```

```
320     * position.
321     * @param agent Robot to put down the entity.
322     * @param entity Grippeable WorldObject to put down.
323     * @return True if the operation is successful, false if not.
324     */
325    public boolean ungrip(String agent, String entity) {
326      if (!this.w.contains(agent)) {
327        System.err.println("ERROR: Ungrip failed: Agent " + agent
328                + " not found!");
329        return false;
330      }
331      if (!this.getObject(agent).getArmPayload().getObjectName().equals(
           entity)) {
332        System.err.println("ERROR: Ungrip failed: Entity " + entity
333                + " not found!");
334        return false;
335      }
336      if (this.getObject(agent).ungrip(this.w, entity)) {
337        return true;
338      }
339      System.err.println("ERROR: " + agent + " failed putting down " +
           entity);
340      return false;
341    }
342
343    /**
344     * An agent fills a container with the output of a source. Agent,
           source and
345     * container have to be at the same position.
346     * @param agent The agent to initiate the filling.
347     * @param source The source to fill the container.
348     * @param container The container to be filled with the output of the
           source.
349     * @return True if the operation is successful, false if not.
350     */
351    public boolean fill(String agent, String source, String container) {
352      if (!this.w.contains(agent)) {
353        System.err.println("ERROR: Fill failed: Agent " + agent
354                + " not found!");
355        return false;
356      }
357      if (!this.w.contains(source)) {
358        System.err.println("ERROR: Fill failed: Source " + source
359                + " not found!");
```

```
360       return false;
361     }
362     if (!this.w.contains(container)) {
363       System.err.println("ERROR: Fill failed: Container " + container
364           + " not found!");
365       return false;
366     }
367     WorldPosition agentPos =
368         this.w.getWorldLocationByWorldObjectName(agent).getPos();
369     WorldPosition sourcePos =
370         this.w.getWorldLocationByWorldObjectName(source).getPos();
371     WorldPosition containerPos =
372         this.w.getWorldLocationByWorldObjectName(container).getPos
            ();
373     // The agent needs to be at the same position as the source.
374     if (agentPos.equals(sourcePos)) {
375       if (sourcePos.equals(containerPos)) {
376         System.out.println(agent + " fills " + container + " at "
377             + source + ".");
378         return this.w.getWorldObjectByName(container).fillContainer(
              this.w, source);
379       } else {
380         System.err.println("ERROR: Fill failed: Container " + container
381             + " is not at the same position as source " + source
382             + ".");
383       }
384     } else {
385       System.err.println("ERROR: Fill failed: Agent " + agent
386           + " is not at the same position as source " + source + ".
              ");
387     }
388     return false;
389   }
390
391   /**
392    * Creates an xml representation of the world and its contents.
393    * @return xml representation of the world.
394    */
395   public String toXmlString() throws NullPointerException {
396     String[] objects = this.w.getWorldObjectNames();
397     String xml = new String();
398     xml += "<world>\n";
399     xml += "  <sizex>" + this.w.getSizeX() + "</sizex>\n";
400     xml += "  <sizey>" + this.w.getSizeY() + "</sizey>\n";
```

```
401     xml += "  <sizez>" + this.w.getSizeZ() + "</sizez>\n";
402     xml += "  <objects>\n";
403
404     for (int i = 0; i
405             < objects.length; i++) {
406       xml += "     <object>\n";
407
408       xml += "       <name>";
409       xml += objects[i];
410       xml += "</name>\n";
411
412       xml += "       <type>";
413       xml += this.w.getWorldObjectByName(objects[i]).getObjectType();
414       xml += "</type>\n";
415
416       xml += "       <x>";
417       xml += this.w.getWorldLocationByWorldObjectName(objects[i]).getX
              ();
418       xml += "</x>\n";
419
420       xml += "       <y>";
421       xml += this.w.getWorldLocationByWorldObjectName(objects[i]).getY
              ();
422       xml += "</y>\n";
423
424       xml += "       <z>";
425       xml += this.w.getWorldLocationByWorldObjectName(objects[i]).getZ
              ();
426       xml += "</z>\n";
427
428       xml += "     </object>\n";
429     }
430     xml += "  </objects>\n";
431     xml += "</world>";
432
433     return xml;
434   }
435 }
```

Listing 10: RobotSim.java

D.6 RtdlParser.java

```
1  /**
2   * RtdlParser.java
3   * http://www.iks.hs-merseburg.de/~uschroet/Literatur/Java_Lit/
        JAVA_Insel/javainsel_15_004.htm#mjd0dbb53e72e3b53b011e15c20054078f
4   *
5   * @author Marcus Teske
6   */
7  package rtdl;
8
9  import java.io.*;
10 import javax.xml.stream.XMLInputFactory;
11 import javax.xml.stream.XMLStreamReader;
12 import javax.xml.stream.XMLStreamException;
13 import javax.xml.stream.XMLStreamConstants;
14
15 public class RtdlParser {
16
17   XMLStreamReader parser;
18   String taskName;
19   boolean inTaskArgs = false;
20   String actionName;
21   int argscnt;
22   String[] actionArgs = new String[5];
23
24   RtdlParser(String filename)
25           throws FileNotFoundException, XMLStreamException {
26     parser = XMLInputFactory.newInstance().createXMLStreamReader(
27             new FileInputStream(filename));
28   }
29
30   public RobotSim parse(RobotSim rs, String robot) throws
        XMLStreamException,
31           InterruptedException {
32
33     while (parser.hasNext()) {
34       switch (parser.getEventType()) {
35         case XMLStreamConstants.START_DOCUMENT:
36           break;
37
38         case XMLStreamConstants.END_DOCUMENT:
39           parser.close();
40           break;
41
```

```
42          case XMLStreamConstants.NAMESPACE:
43            break;
44
45          case XMLStreamConstants.START_ELEMENT:
46            // Version checking, only 1.0 is valid
47            if (parser.getLocalName().equals("rtdl")) {
48              for (int i = 0; i < parser.getAttributeCount(); i++) {
49                if (parser.getAttributeLocalName(i).equals("version")
50                      && parser.getAttributeValue(i).equals("1.0")) {
51                  System.out.println("RTDL Version "
52                          + parser.getAttributeValue(i) + " OK!");
53                } else {
54                  System.err.println("RTDL Version "
55                          + parser.getAttributeValue(i)
56                          + " invalid.");
57                  System.exit(1);
58                }
59              }
60            }
61
62            // Extract task name
63            if (parser.getLocalName().equals("task")) {
64              for (int i = 0; i < parser.getAttributeCount(); i++) {
65                if (parser.getAttributeLocalName(i).equals("task")) {
66                  taskName = parser.getAttributeValue(i);
67                  System.out.println("Parsing task " + taskName);
68                }
69              }
70            }
71
72            // Extract action name
73            if (parser.getLocalName().equals("name")) {
74              actionName = parser.getElementText();
75              System.out.print(robot + ": " + actionName + "(");
76            }
77
78            // Reset args counter on entering args-element.
79            if (parser.getLocalName().equals("args")) {
80              argscnt = -1;
81            }
82
83            // Extract acion args
84            if (parser.getLocalName().equals("arg")) {
85              argscnt++;
```

```
86              actionArgs[argscnt] = parser.getElementText();
87              if (argscnt == 0) {
88                System.out.print('"' + actionArgs[argscnt] + '"');
89              } else {
90                System.out.print(", " + '"' + actionArgs[argscnt] + '"');
91              }
92            }
93
94            break;
95
96          case XMLStreamConstants.CHARACTERS:
97            break;
98
99          case XMLStreamConstants.END_ELEMENT:
100           // Mark leaving of task args tag
101           if (parser.getLocalName().equals("taskargs")) {
102             inTaskArgs = false;
103           }
104
105           // Execute actions with exactly one argument and clean up
106           if (parser.getLocalName().equals("action")) {
107             System.out.println(");");
108             if (actionName.equals("move")) {
109               rs.move(robot, actionArgs[0]);
110               Thread.sleep(1000);
111             } else if (actionName.equals("grip")) {
112               rs.grip(robot, actionArgs[0]);
113               Thread.sleep(1000);
114             } else if (actionName.equals("ungrip")) {
115               rs.ungrip(robot, actionArgs[0]);
116               Thread.sleep(1000);
117             } else if (actionName.equals("fill")) {
118               rs.fill(robot, actionArgs[0], actionArgs[1]);
119               Thread.sleep(1000);
120             } else if (actionName.equals("setCurrentPosition")) {
121               rs.createWorldObject("Waypoint",
122                       actionArgs[0],
123                       (int) rs.getPosition(robot).getX(),
124                       (int) rs.getPosition(robot).getY(),
125                       (int) rs.getPosition(robot).getZ());
126               Thread.sleep(1000);
127             }
128             actionName = "";
129             argscnt = -1;
```

```
130             }
131
132             // Reset action counter.
133             if (parser.getLocalName().equals("args")) {
134             }
135
136             break;
137           default:
138             break;
139         }
140         parser.next();
141       }
142       return rs;
143     }
144 }
```

Listing 11: RtdlParser.java

D.7 Source.java

```
1  /**
2   * Source.java
3   * A source that produces some output of a sourceType.
4   * Examples are a tap or a power socket.
5   *
6   * @author Marcus Teske
7   */
8  package rtdl;
9
10 public class Source extends WorldObject {
11
12   protected String sourceType;
13
14   Source(String objectName, String sourceType) {
15     super.setObjectName(objectName);
16     super.isSource = true;
17     this.sourceType = sourceType;
18   }
19
20   public String getSourceType() {
21     return this.sourceType;
22   }
23 }
```

Listing 12: Source.java

D.8 Tap.java

```
1  /**
2   * Tap.java
3   * A tap that can be used to fill container-like WorldObjects with an
        output.
4   *
5   * @author Marcus Teske
6   */
7  package rtdl;
8
9  public class Tap extends Source {
10
11   public Tap(String objectName) {
12     super(objectName, "water");
13   }
14 }
```

Listing 13: Tap.java

D.9 Wall.java

```
1  /**
2   * Wall.java
3   * A simple obstacle.
4   * @author Marcus Teske
5   */
6  package rtdl;
7
8  public class Wall extends WorldObject {
9
10   Wall(String objectName) {
11     super.setObjectName(objectName);
12     super.isPassable = false;
13   }
14 }
```

Listing 14: Wall.java

D.10 Waypoint.java

```
1  /**
2   * Waypoint.java
3   * A simple waypoint.
4   * @author Marcus Teske
5   */
6  package rtdl;
7
8  public class Waypoint extends WorldObject {
9
10   Waypoint(String objectName) {
11     super.setObjectName(objectName);
12   }
13 }
```

Listing 15: Waypoint.java

D.11 World.java

```
1  /**
2   * World.java
3   * A World is an environment consisting of multiple WorldLocations
        arranged in a
4   * 3D coordinate system. The World may contain an arbitrary number of
        different
5   * WorldObjects of different types that are referenced by unique names.
6   *
7   * @author Marcus Teske
8   */
9  package rtdl;
10
11 import java.util.HashMap;
12 import java.util.ArrayList;
13
14 public class World {
15   /* HashMap holding WorldPositions (coordinates) referenced by
16   WorldObjectNames. */
17
18   private HashMap<String, WorldLocation> positions =
19           new HashMap<String, WorldLocation>();
20   /* Array holding WorldLocations (which contain WorldObjects)
        referenced by
```

```
21    their coordinates. */
22    private WorldLocation worldLocations[][][];
23    private int sizeX;
24    private int sizeY;
25    private int sizeZ;
26
27    /**
28     * Constructor
29     * @param x Spread of the World along the x-axis.
30     * @param y Spread of the World along the y-axis.
31     * @param z Spread of the World along the z-axis.
32     */
33    World(int x, int y, int z) {
34      this.sizeX = x;
35      this.sizeY = y;
36      this.sizeZ = z;
37
38      worldLocations = new WorldLocation[x][y][z];
39      for (int i = 0; i < x; i++) {
40        for (int j = 0; j < y; j++) {
41          for (int k = 0; k < z; k++) {
42            worldLocations[i][j][k] = new WorldLocation(i, j, k);
43          }
44        }
45      }
46    }
47
48    /**
49     * Get the WorldLocation where the WorldObject with name "name" is
          located.
50     * @param name Name of the WorldObject that references the
          WorldLocation.
51     * @return TODO
52     */
53    public WorldLocation getWorldLocationByWorldObjectName(String name) {
54      if (this.contains(name)) {
55        return positions.get(name);
56      }
57      return null;
58    }
59
60    /**
61     * Get the WorldObject with the name "name"
62     * @param name Name of the WorldObject.
```

```
63     */
64    public WorldObject getWorldObjectByName(String name) {
65      if (this.contains(name)) {
66        WorldLocation wl = this.getWorldLocationByWorldObjectName(name);
67        return this.worldLocations[wl.getX()][wl.getY()][wl.getZ()]
68                .getWorldObject(name);
69      } else {
70        return null;
71      }
72    }
73
74    /**
75     * Gets the type/classname of the WorldObject referenced by "name".
76     * @param name Unique name of the WorldObject.
77     * @return Type/classname of the WorldObject.
78     */
79    public String getWorldObjectType(String name) {
80      return this.getWorldObjectByName(name).getObjectType();
81    }
82
83    /**
84     * Remove the WorldObject with name "name" from the World.
85     * @param name Name of the WorldObject to remove.
86     */
87    public boolean removeWorldObjectByName(String name) {
88      if (this.contains(name)) {
89        WorldLocation wl = this.getWorldLocationByWorldObjectName(name);
90        // Remove the WorldObject itself.
91        worldLocations[wl.getX()][wl.getY()][wl.getZ()].removeWorldObject
              (name);
92        // Remove the entry from the positions
93        this.positions.remove(name);
94        return true;
95      } else {
96        System.err.println("ERROR: Cannot remove WorldObject " + name
97                + "! It does not exists!");
98        return false;
99      }
100   }
101
102   /**
103    * Remove the WorldObject with name "name" from the world and return
           it.
104    * @param name Name of the WorldObject to remove.
```

```
105     * @return The WorldObject that has been removed from the World.
106     */
107    public WorldObject takeObjectByName(String name) {
108      WorldObject wo = this.getWorldObjectByName(name);
109      if (removeWorldObjectByName(name)) {
110        return wo;
111      }
112      return null;
113    }
114
115    /**
116     * Place a WorldObject at the WorldLocation specified by wl.
117     * @param wo WorldObject to place.
118     * @param wl WorldLocation where the WorldObject should be placed.
119     * @return true if the operation was successful, false otherwise.
120     */
121    public boolean setWorldObject(WorldObject wo, WorldLocation wl) {
122      return this.setWorldObject(wo, wl.getX(), wl.getY(), wl.getZ());
123    }
124
125    /**
126     * Place a WorldObject at the WorldLocation specified by wp.
127     * @param wo WorldObject to place.
128     * @param wp WorldLocation where the WorldObject should be placed.
129     * @return true if the operation was successful, false otherwise.
130     */
131    public boolean setWorldObject(WorldObject wo, WorldPosition wp) {
132      return this.setWorldObject(wo,
133              (int) wp.getX(),
134              (int) wp.getY(),
135              (int) wp.getZ());
136    }
137
138    /**
139     * Place a WorldObject at the WorldLocation specified by x, y and z.
140     * @param wo WorldObject to place.
141     * @param x X-coordinate of the Location.
142     * @param y Y-coordinate of the Location.
143     * @param z Z-coordinate of the Location.
144     * @return Returns true if the WorldObject has been placed, false if
            not.
145     */
146    public boolean setWorldObject(WorldObject wo, int x, int y, int z) {
147      if (this.locationValid(x, y, z)) {
```

```
148         this.positions.put(wo.getObjectName(), new WorldLocation(x, y, z)
                );
149         this.worldLocations[x][y][z].setWorldObject(wo);
150         return true;
151       } else {
152         return false;
153       }
154     }
155
156     /**
157      * Checks if a WorldObject with ObjectName "name" exists in the world
             .
158      * @param name Unique name of the WorldObject.
159      * @return True if the Object exists, False otherwise.
160      */
161     public boolean contains(String name) {
162       return this.positions.containsKey(name);
163     }
164
165     /**
166      * Gets the names of all Objects in the World as string array.
167      * @return String array with the names of all WorldObjects in this
             world.
168      */
169     public String[] getWorldObjectNames() {
170       String[] sa = new String[this.positions.keySet().size()];
171       int i = 0;
172       for (String s : this.positions.keySet()) {
173         sa[i] = s;
174         i++;
175       }
176       return sa;
177     }
178
179     /**
180      * Checks whether a WorldLocation/WorldPosition is valid or not.
181      * @param x x-ccordinate of the WorldLocation/WorldPosition.
182      * @param y y-ccordinate of the WorldLocation/WorldPosition.
183      * @param z z-ccordinate of the WorldLocation/WorldPosition.
184      * @return True if the coordinates are valid, False if not.
185      */
186     public boolean locationValid(int x, int y, int z) {
187       if (x >= this.sizeX || x < 0) {
```

```
188       System.err.println("ERROR: WordLocation [" + x + "," + y + "," +
              z
189               + "] invalid: X-coordinate is " + "outside of this world'
                  s "
190               + "boundaries!");
191       return false;
192     }
193     if (y >= this.sizeY || y < 0) {
194       System.err.println("ERROR: WordLocation [" + x + "," + y + "," +
              z
195               + "] invalid: Y-coordinate is " + "outside of this world'
                  s "
196               + "boundaries!");
197       return false;
198     }
199     if (z >= this.sizeZ || z < 0) {
200       System.err.println("ERROR: WordLocation [" + x + "," + y + "," +
              z
201               + "] invalid: Z-coordinate is " + "outside of this world'
                  s "
202               + "boundaries!");
203       return false;
204     }
205     return true;
206   }
207
208   /**
209    * Checks whether a WorldLocation/WorldPosition is valid or not.
210    * @param wp WorldPosition to check.
211    * @return True if the coordinates are valid, False if not.
212    */
213   public boolean locationValid(WorldPosition wp) {
214     return this.locationValid((int) wp.getX(), (int) wp.getY(), (int)
          wp.getZ());
215   }
216
217   public int getSizeX() {
218     return sizeX;
219   }
220
221   public int getSizeY() {
222     return sizeY;
223   }
224
```

```
225   public int getSizeZ() {
226     return sizeZ;
227   }
228
229   /**
230    * Gets all WorldObjects from the specified WorldLocation.
231    * @param x x-ccordinate of the location
232    * @param y y-ccordinate of the location
233    * @param z z-ccordinate of the location
234    * @return All Objects from that location
235    */
236   public ArrayList<WorldObject> getAllObjectsFromLocation(int x, int y,
          int z) {
237     return this.worldLocations[x][y][z].getObjects();
238   }
239
240   /**
241    * Returns whether a location is passable or not.
242    * @param x x-ccordinate of the location
243    * @param y y-ccordinate of the location
244    * @param z z-ccordinate of the location
245    * @return true if the location is passable, false otherwise
246    */
247   public boolean isPassable(int x, int y, int z) {
248     return this.worldLocations[x][y][z].isPassable();
249   }
250
251   /**
252    * Returns whether a location is passable or not.
253    * @param pos Postion of the location
254    * @return true if the location is passable, false otherwise
255    */
256   public boolean isPassable(WorldPosition pos) {
257     return this.worldLocations[(int) pos.getX()][(int) pos.getY()][(int
          ) pos
258               .getZ()].isPassable();
259   }
260 }
```

Listing 16: World.java

D.12 WorldLocation.java

```
1  /**
2   * WorldLocation.java
3   * A WorldLocation is a location within a World. It contains a unique
4   * Location and can hold an arbitrary number of named Worldobjects.
5   *
6   * @author Marcus Teske
7   */
8  package rtdl;
9
10 import java.util.HashMap;
11 import java.util.ArrayList;
12
13 public class WorldLocation {
14   /* The WorldPosition of this WorldLocation. */
15
16   WorldPosition pos;
17   /* A map where WorldObjects located in this Location are stored and
         referenced
18    * by their unique name.
19    */
20   HashMap<String, WorldObject> objects = new HashMap<String,
         WorldObject>();
21
22   /**
23    * Constructor of WorldLoction
24    * @param x The x-coordinate.
25    * @param y The y-coordinate.
26    * @param z The z-coordinate.
27    */
28   WorldLocation(int x, int y, int z) {
29     pos = new WorldPosition((long) x, (long) y, (long) z);
30   }
31
32   /**
33    * Get the named WorldObject referenced by its unique name.
34    * @param name The unique name of the named WorldObject.
35    * @return The named WorldObject.
36    */
37   public WorldObject getWorldObject(String name) {
38     return (WorldObject) objects.get(name);
39   }
40
41   /**
```

```
42      * Store a WorldObject indexed by its unique name in the location's
            object
43      * pool.
44      * @param wo WorldObject to store.
45      */
46     public void setWorldObject(WorldObject wo) {
47       this.objects.put(wo.getObjectName(), wo);
48     }
49
50     /**
51      * Gets the postion of this WorldLocation as position object.
52      * @return Position as position object.
53      */
54     public WorldPosition getPos() {
55       return pos;
56     }
57
58     /**
59      * Sets a new position for this WorldLocation.
60      * @param pos new position for this WorldLocation.
61      */
62     public void setPos(WorldPosition pos) {
63       this.pos = pos;
64     }
65
66     /**
67      * Gets the distance from the origin along the x-axis of this
            WorldLocation.
68      * @return Distance to the origin along the x-axis.
69      */
70     public int getX() {
71       return (int) this.pos.getX();
72     }
73
74     /**
75      * Gets the distance from the origin along the y-axis of this
            WorldLocation.
76      * @return Distance to the origin along the y-axis.
77      */
78     public int getY() {
79       return (int) this.pos.getY();
80     }
81
82     /**
```

```
83     * Gets the distance from the origin along the z-axis of this
           WorldLocation.
84     * @return Distance to the origin along the z-axis.
85     */
86    public int getZ() {
87      return (int) this.pos.getZ();
88    }
89
90    /**
91     * Remove the WorldObject with name "name".
92     * @param name Name of the WorldObject to remove.
93     */
94    public void removeWorldObject(String name) {
95      this.objects.remove(name);
96    }
97
98    /**
99     * Get the type/classname of an named object selected by its unique
           name.
100    * @param name Unique name of an named object in the World.
101    * @return The type/classname of the named object.
102    */
103   public String getWorldObjectType(String name) {
104     return this.getWorldObject(name).getObjectType();
105   }
106
107   /**
108    * Gets all objects of this location as ArrayList<WorldObject>
109    * @return All Objects as ArrayList
110    */
111   public ArrayList<WorldObject> getObjects() {
112     return new ArrayList<WorldObject>(this.objects.values());
113   }
114
115   /**
116    * Returns whether this location contains any impassable objects.
117    * @return true if all objects are passable, false otherwise.
118    */
119   public boolean isPassable() {
120     ArrayList<WorldObject> wos = this.getObjects();
121     for (WorldObject wo : wos) {
122       if (!wo.isPassable) {
123         return false;
124       }
```

```
125     }
126     return true;
127   }
128 }
```

Listing 17: WorldLocation.java

D.13 WorldObject.java

```
1  /**
2   * WorldObject.java
3   * A class that represents the general features of all WorldObjects.
        These
4   * features are the unique name of an object and the type of an object.
5   *
6   * @author Marcus Teske
7   */
8  package rtdl;
9
10 public class WorldObject {
11
12   private String objectName;
13   // WorldObject's predicates
14   protected boolean canMove = false;
15   protected boolean isGrippeable = false;
16   protected boolean isPassable = true;
17   protected boolean isContainer = false;
18   protected boolean isSource = false;
19
20   /**
21    * Get the name of this WorldObject.
22    * @return The name of this object.
23    */
24   public String getObjectName() {
25     return objectName;
26   }
27
28   /**
29    * Sets the name of an WorldObject.
30    * @param name The new name to set.
31    */
32   public void setObjectName(String name) {
33     this.objectName = name;
```

```
34    }
35
36    /**
37     * Gets the classname/type of this WorldObject.
38     * @return The ObjectType of this WorldObject as String.
39     */
40    public String getObjectType() {
41      return this.getClass().getSimpleName();
42    }
43
44    /**
45     * Placeholder method for moving inside the world.
46     * @param w The world the action takes place in.
47     * @param x The x-coordinate of the detination.
48     * @param y The y-coordinate of the detination.
49     * @param z The z-coordinate of the detination.
50     * @return false
51     */
52    public boolean move(World w, int x, int y, int z) {
53      if (!this.canMove) {
54        System.err.println("WARNING: A " + this.getObjectType()
55                + " cannot move!");
56      } else {
57        System.err.println("WARNING: Movement of " + this.getObjectType()
58                + "not yet implemented!");
59      }
60      return false;
61    }
62
63    public WorldObject getArmPayload() {
64      System.err.println("WARNING: A " + this.getObjectName()
65              + " has no arm!");
66      return null;
67    }
68
69    /**
70     * Placeholder method for filling a container WorldObject with a
           certain#
71     * content type.
72     * @param contentType Type of content to fill the container with.
73     * @return false
74     */
75    public boolean fillContainer(World w, String contentType) {
76      if (!this.isContainer) {
```

```
77        System.err.println("WARNING: A " + this.getObjectType()
78                + " is not a container!");
79      } else {
80        System.err.println("WARNING: Filling of " + this.getObjectType()
81                + "not yet implemented!");
82      }
83      return false;
84    }
85
86    /**
87     * Placeholder method for emptying a container WorldObject.
88     * @return false
89     */
90    public boolean emptyContainer() {
91      if (!this.isContainer) {
92        System.err.println("WARNING: A " + this.getObjectType()
93                + " is not a container!");
94      } else {
95        System.err.println("WARNING: Emptying of " + this.getObjectType()
96                + "not yet implemented!");
97      }
98      return false;
99    }
100
101   public boolean grip(World w, String name) {
102     return false;
103   }
104
105   public boolean ungrip(World w, String name) {
106     return false;
107   }
108 }
```

Listing 18: WorldObject.java

D.14 WorldPosition.java

```
1 /**
2  * WorldPosition.java
3  * A WorldPosition is a Point within a World that is specified by its x
       -, y- and
4  * z-coordinate in a 3D coordinate system.
5  *
```

```
6  * @author Marcus Teske
7  */
8 package rtdl;
9
10 public class WorldPosition {
11
12   private long x;
13   private long y;
14   private long z;
15
16   /**
17    * Constructor of WorldPosition.
18    * @param x The x-coordinate.
19    * @param y The y-coordinate.
20    * @param z The z-coordinate.
21    */
22   public WorldPosition(long x, long y, long z) {
23     this.x = x;
24     this.y = y;
25     this.z = z;
26   }
27
28   /**
29    * Gets the distance from the origin along the x-axis of this
          WorldLocation.
30    * @return Distance to the origin along the x-axis.
31    */
32   public long getX() {
33     return x;
34   }
35
36   /**
37    * Sets the distance from the origin along the x-axis of this
          WorldLocation.
38    */
39   public void setX(long x) {
40     this.x = x;
41   }
42
43   /**
44    * Gets the distance from the origin along the y-axis of this
          WorldLocation.
45    * @return Distance to the origin along the y-axis.
46    */
```

```
47    public long getY() {
48      return y;
49    }
50
51    /**
52     * Sets the distance from the origin along the y-axis of this
           WorldLocation.
53     */
54    public void setY(long y) {
55      this.y = y;
56    }
57
58    /**
59     * Gets the distance from the origin along the z-axis of this
           WorldLocation.
60     * @return Distance to the origin along the z-axis.
61     */
62    public long getZ() {
63      return z;
64    }
65
66    /**
67     * Sets the distance from the origin along the z-axis of this
           WorldLocation.
68     */
69    public void setZ(long z) {
70      this.z = z;
71    }
72
73    public long getOffsetX(WorldPosition wp) {
74      return wp.getX() - this.getX();
75    }
76
77    public long getOffsetY(WorldPosition wp) {
78      return wp.getY() - this.getY();
79    }
80
81    public long getOffsetZ(WorldPosition wp) {
82      return wp.getZ() - this.getZ();
83    }
84
85    public boolean equals(WorldPosition wp) {
86      return (this.getX() == wp.getX() && this.getY() == wp.getY() &&
87              this.getZ() == wp.getZ());
```

```
88    }
89 }
```

Listing 19: WorldPosition.java

D.15 fetchWater.xml

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <?xml-stylesheet type="text/xml" href="rtdl.xsl" version="1.0"?>
3  <rtdl version="1.0">
4    <task name="fetchWater">
5      <taskargs>
6        <var type="DrinkingGlass">Glass1</var>
7        <var type="Tap">Tap1</var>
8      </taskargs>
9      <entities>
10       <var type="Waypoint">start</var>
11     </entities>
12     <block>
13       <action>
14         <name>setCurrentPosition</name>
15         <args>
16           <arg>Start</arg>
17         </args>
18       </action>
19       <action>
20         <name>move</name>
21         <args>
22           <arg>Glass1</arg>
23         </args>
24       </action>
25       <action>
26         <name>grip</name>
27         <args>
28           <arg>Glass1</arg>
29         </args>
30       </action>
31       <action>
32         <name>move</name>
33         <args>
34           <arg>Tap1</arg>
35         </args>
36       </action>
```

```
37        <action>
38          <name>ungrip</name>
39          <args>
40            <arg>Glass1</arg>
41          </args>
42        </action>
43        <action>
44          <name>fill</name>
45          <args>
46            <arg>Tap1</arg>
47            <arg>Glass1</arg>
48          </args>
49        </action>
50        <action>
51          <name>grip</name>
52          <args>
53            <arg>Glass1</arg>
54          </args>
55        </action>
56        <action>
57          <name>move</name>
58          <args>
59            <arg>Start</arg>
60          </args>
61        </action>
62        <action>
63          <name>ungrip</name>
64          <args>
65            <arg>Glass1</arg>
66          </args>
67        </action>
68      </block>
69    </task>
70 </rtdl>
```

Listing 20: fetchWater.xml

D.16 rtdl.xsl

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <xsl:stylesheet
3       version="2.0"
4       xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
```

```
5       xmlns:w="http://wizzard.dyndns.info"
6   >
7     <xsl:output method="html" indent="yes"/>
8
9     <!-- This stylesheet transforms the xml representation of an defined
          RTDL task
10          into it the functional syntax.
11          Author: Marcus Teske
12          Date: 2011-01-11
13      -->
14
15    <!-- Starts the transformation. -->
16    <xsl:template match="/">
17      <html>
18        <head>
19          <title><xsl:value-of select="/rtdl/task/@name"/></title>
20          <style>
21            p {
22              margin-left: 2em;
23              font-family: monospace;
24              font-size: 12px;
25            }
26            p.root {
27              margin-left: 0em;
28            }
29          </style>
30        </head>
31        <body>
32          <xsl:apply-templates />
33        </body>
34      </html>
35    </xsl:template>
36
37    <!-- Create the "rtdl <version>" line. -->
38    <xsl:template match="rtdl">
39      <p class="root">
40        <xsl:text>rtdl </xsl:text>
41          <xsl:value-of select="@version"/>
42      </p>
43      <xsl:apply-templates/>
44    </xsl:template>
45
46    <!-- Transforms the comment elements. -->
47    <xsl:template match="comment">
```

```
48      <p class="root">
49        <xsl:text>/* </xsl:text>
50        <xsl:value-of select="."/>
51        <xsl:text> */</xsl:text>
52      </p>
53    </xsl:template>
54
55    <!-- Creates the signature of the defined task. -->
56    <xsl:template match="task">
57      <p class="root">
58        <xsl:value-of select="@name"/>
59        <xsl:text>(</xsl:text>
60        <xsl:for-each select="taskargs/var">
61          <xsl:value-of select="@type"/>
62          <xsl:text> </xsl:text>
63          <xsl:apply-templates/>
64          <xsl:if test="not(position()=last())">
65            <xsl:text>, </xsl:text>
66          </xsl:if>
67        </xsl:for-each>
68        <xsl:text>) {</xsl:text>
69        <br/>
70        <xsl:apply-templates select="retrycount"/>
71        <br/>
72        <xsl:apply-templates select="entities"/>
73        <br/>
74        <xsl:apply-templates select="block" />
75        <br/>
76        <xsl:text>}</xsl:text>
77      </p>
78    </xsl:template>
79
80    <!-- Create comma separated list of arguments and their types. -->
81    <xsl:template match="args">
82      <xsl:for-each select="var">
83        <xsl:value-of select="@type"/>
84          <xsl:text> </xsl:text>
85          <xsl:apply-templates/>
86          <xsl:if test="not(position()=last())">
87            <xsl:text>, </xsl:text>
88          </xsl:if>
89        </xsl:for-each>
90    </xsl:template>
91
```

```
92  <!-- Create comma separated list of arguments without their types.
        -->
93  <xsl:template match="args">
94    <xsl:for-each select="var">
95        <xsl:apply-templates/>
96        <xsl:if test="not(position()=last())">
97          <xsl:text>, </xsl:text>
98        </xsl:if>
99      </xsl:for-each>
100 </xsl:template>
101
102 <!-- Transforms the retrycount element. -->
103 <xsl:template match="retrycount">
104   <p>
105     <xsl:text>retrycount(</xsl:text>
106     <xsl:value-of select="."/>
107     <xsl:text>);</xsl:text>
108   </p>
109 </xsl:template>
110
111 <!-- Create for each entity a line "entity <type> <name>;". -->
112 <xsl:template match="entities">
113   <xsl:for-each select="var">
114     <p>
115       <xsl:text>entity </xsl:text>
116       <xsl:value-of select="@type"/>
117       <xsl:text> </xsl:text>
118       <xsl:apply-templates/>
119       <xsl:text>;</xsl:text>
120     </p>
121   </xsl:for-each>
122 </xsl:template>
123
124 <!-- Transforms block-elements. -->
125 <xsl:template match="block">
126   <xsl:apply-templates select="action"/>
127 <xsl:apply-templates select="while"/>
128 <xsl:apply-templates select="if"/>
129 <xsl:apply-templates select="par"/>
130 </xsl:template>
131
132 <!-- Transforms retry-elements. -->
133 <xsl:template match="retry">
134   <p>
```

```
135       <xsl:text>retry(</xsl:text>
136       <xsl:value-of select="@count"/>
137       <xsl:text>)</xsl:text>
138     <xsl:apply-templates/>
139     <xsl:text>endtry</xsl:text>
140     </p>
141   </xsl:template>
142
143   <!-- transforms actions to a function call. -->
144   <xsl:template match="action">
145     <p>
146       <xsl:if test="@optional='true'">
147         <xsl:text>optional </xsl:text>
148       </xsl:if>
149       <xsl:value-of select="name"/>
150       <xsl:text>(</xsl:text>
151       <xsl:for-each select="args/arg">
152         <xsl:apply-templates/>
153         <xsl:if test="not(position()=last())">
154           <xsl:text>, </xsl:text>
155         </xsl:if>
156       </xsl:for-each>
157       <xsl:apply-templates select="args"/>
158       <xsl:text>);</xsl:text>
159     </p>
160   </xsl:template>
161
162   <!-- Transforms while-blocks. -->
163   <xsl:template match="while">
164     <p>
165       <xsl:text>while</xsl:text>
166       <xsl:text>(</xsl:text>
167       <xsl:apply-templates select="cond"/>
168       <xsl:text>)</xsl:text>
169       <br/>
170       <xsl:text>do</xsl:text>
171       <xsl:apply-templates select="block"/>
172       <xsl:text>done</xsl:text>
173     </p>
174   </xsl:template>
175
176   <!-- Transforms condition of a while block as is. -->
177   <xsl:template match="cond">
178     <xsl:value-of select="."/>
```

```
179   </xsl:template>
180
181   <!-- Transforms if-block. -->
182   <xsl:template match="if">
183     <p>
184       <xsl:text>if</xsl:text>
185       <xsl:text>(</xsl:text>
186       <xsl:apply-templates select="cond"/>
187       <xsl:text>)</xsl:text>
188       <br/>
189       <xsl:apply-templates select="block"/>
190       <xsl:text>endif</xsl:text>
191     </p>
192   </xsl:template>
193
194   <!-- Transforms par-blocks. -->
195   <xsl:template match="par">
196     <p>
197       <xsl:text>par</xsl:text>
198       <br/>
199       <xsl:apply-templates select="block"/>
200       <xsl:text>to</xsl:text>
201       <br/>
202       <xsl:apply-templates select="block2"/>
203       <xsl:text>endpar</xsl:text>
204     </p>
205   </xsl:template>
206
207   <!-- Transforms first block of par. -->
208   <xsl:template match="block1">
209     <xsl:apply-templates/>
210   </xsl:template>
211
212   <!-- Transforms second block of par. -->
213   <xsl:template match="block2">
214     <xsl:apply-templates/>
215   </xsl:template>
216
217 </xsl:stylesheet>
```

Listing 21: rtdl.xsl

D.17 rtdl.dtd

```
<!-- RTDL v1.0 Document Type Definition -->

<!ELEMENT rtdl (comment*, task, comment*)>
<!ATTLIST rtdl
  version CDATA #REQUIRED
>
<!ELEMENT comment (#PCDATA)>
<!ELEMENT task (comment*,
                taskargs,
                comment*,
                retrycount?,
                comment*,
                entities,
                comment*,
                block,
                comment*
)>
<!ATTLIST task
  name CDATA #REQUIRED
>
<!ELEMENT taskargs (var | comment)*>
<!ELEMENT var (#PCDATA)>
<!ATTLIST var
  type CDATA #REQUIRED
>
<!ELEMENT retrycount (#PCDATA)>
<!ELEMENT entities (var | comment)*>
<!ELEMENT block (action | par | if | while | retry | comment)*>
<!ELEMENT action (comment*, name, comment*, args?, comment*)>
<!ATTLIST action
  optional CDATA #IMPLIED
>
<!ELEMENT name (#PCDATA)>
<!ELEMENT args (arg | comment)*>
<!ELEMENT arg (#PCDATA)>
<!ELEMENT par (block, block2)>
<!ELEMENT if (cond, block)>
<!ELEMENT cond (#PCDATA)>
<!ELEMENT while (cond, block)>
<!ELEMENT retry (block)>
<!ATTLIST retry
  count CDATA #REQUIRED
>
```

Listing 22: rtdl.dtd

D.18 example.xml

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <?xml-stylesheet type="text/xml" href="rtdl.xsl" version="1.0"?>
3
4  <!DOCTYPE rtdl SYSTEM "rtdl.dtd">
5
6  <rtdl version="1.0">
7    <comment>This is a comment.</comment>
8    <task name="exampleTask">
9      <taskargs>
10       <var type="FirstArg">a1</var>
11       <var type="SecondArg">a2</var>
12     </taskargs>
13     <retrycount>3</retrycount>
14     <entities>
15       <var type="Entity">entity1</var>
16       <var type="Entity">entity2</var>
17       <var type="Entity">entity3</var>
18     </entities>
19     <block>
20       <action>
21         <name>Task1</name>
22         <args>
23           <arg>entity1</arg>
24           <arg>entity2</arg>
25           <arg>entity3</arg>
26         </args>
27       </action>
28       <action>
29         <name>Task2</name>
30         <args>
31           <arg>entity1</arg>
32           <arg>entity2</arg>
33           <arg>entity3</arg>
34         </args>
35       </action>
36       <while>
37         <cond>condition1</cond>
```

```
38          <block>
39            <action>
40              <name>Task3</name>
41              <args>
42                <arg>entity1</arg>
43              </args>
44            </action>
45          </block>
46        </while>
47        <if>
48          <cond>condition2</cond>
49          <block>
50            <action>
51              <name>Task4</name>
52            </action>
53          </block>
54        </if>
55        <par>
56          <block>
57            <action>
58              <name>Task5</name>
59            </action>
60            <par>
61              <block>
62                <action>
63                  <name>Task6</name>
64                </action>
65              </block>
66              <block>
67                <action>
68                  <name>Task7</name>
69                </action>
70              </block>
71            </par>
72          </block>
73          <block>
74            <action>
75              <name>Task8</name>
76            </action>
77          </block>
78        </par>
79        <retry count="5">
80          <block>
81            <action optional="true">
```

```
82                <name>optionalTask</name>
83              </action>
84            </block>
85          </retry>
86        </block>
87    </task>
88 </rtdl>
```

Listing 23: example.xml

Literatur

[Bud09] Budde, Henning: *Wissensakquisition und maschinelle Task-Generierung für einen virtuellen Küchenroboter.* , Fachhochschule Köln, 2009. Masterthesis.

[Cok10] *CokeRob: Ein Blickfang am Tag der offenen Tür 2004*, Aufruf: 9. November 2010. http://www.nt-rt.fh-koeln.de/News/n006.html.

[Erd09] Erdmann, Ralph: *Steuerung eines autonom fahrenden Roboters auf der Basis von RFID-Positionsmarken und Richtungsinformationen.* , Fachhochschule Köln, 2009. Bachelorthesis.

[Hol10] Holz, Dirk: *Effiziente Kartopraghie und Navigation für mobile Service Roboter.* Informatik Spektrum, 04/2010. Springer Verlag.

[JAU05] JAUS WG: *JAUS Volume I: Domain Model*, 20. März 2005. v3.2.

[JAU06] JAUS WG: *JAUS Compliance Specification*, 26. Oktober 2006. v3.2.

[JAU07a] JAUS WG: *JAUS Volume II: Reference Architecture: Part 1 Architecture Framework*, 27. Juni 2007. v3.3.

[JAU07b] JAUS WG: *JAUS Volume II: Reference Architecture: Part 2 Message Definition*, 27. Juni 2007. v3.3.

[JAU07c] JAUS WG: *JAUS Volume II: Reference Architecture: Part 3 Message Set*, 27. Juni 2007. v3.3.

[McD11] McDermott, Drew V.: *Planning Domain Definition Language*, Aufruf: 11. Januar 2011. http://cs-www.cs.yale.edu/homes/dvm/.

[MT10] Moritz Ternorth, Dominik Jain, Michael Beetz: *Knowledge Processing for Cognitive Robots.* Künstliche Intelligenz, Nr. 24:233–240, 2010.

[Rob11] *RoboCup*, Aufruf: 11. Januar 2011. http://www.robocup.org.

[SV03] Spyros Vosinakis, Themis Panayiotopoulos: *A Task Definition Language for Virtual Agents.* , Universität von Piräus, Griechenland, 2003. http://wscg.zcu.cz/wscg2003/Papers_2003/D03.pdf.

[Tes10] Teske, Marcus: *The Joint Architecture for Unmanned Systems (JAUS): Eine Übersicht.* v1.1, 3. Juni 2010.

[Wik10] Wikipedia Enzyklopädie: *American Standard Code for Information Interchange*, Aufruf: 2. Dezember 2010. http://de.wikipedia.org/w/index.php?title=American_Standard_Code_for_Information_Interchange&oldid=82181574.