Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Andreas Basener

Integration des c't-Bots in das Robotics Developer Studio

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Andreas Basener

Integration des c't-Bots in das Robotics Developer Studio

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. rer. nat. Gunter Klemke
Zweitgutachter : Prof. Dr. Franz Korf

Abgegeben am 18. Dezember 2008

Andreas Basener

Thema der Bachelorarbeit

Integration des c't-Bots in das Robotics Developer Studio

Stichworte

c't-Bot, Robotics Development Studio, Visual Programming Language

Kurzzusammenfassung

Das Robotics Development Studio ist eine Entwicklungsumgebung für Roboter, die eine serviceorientierte Laufzeitumgebung enthält. Im Robotics Development Studio ist die Visual Programming Language enthalten, die besonders Einsteigern mit intuitiven und visuellen Mitteln einen Zugang in die Roboterprogrammierung bieten soll. Der c't-Bot ist ein Projekt der Zeitschrift c't. Der Roboter wird seit einigen Semestern im Roboterlabor der Hochschule für Angewandte Wissenschaften Hamburg eingesetzt. Da bei anspruchsvolleren Aufgaben die Leistung des Roboters oft nicht ausreicht, entstand die Idee, arbeitsintensive Aufgaben auf einen leistungsstarken Rechner auszulagern.

Andreas Basener

Title of the paper

Integrating the c't-Bot into the Robotics Developer Studio

Keywords

c't-Bot, Robotics Development Studio, Visual Programming Language

Abstract

The Robotics Development Studio is a development environment for robots, containing a service-oriented runtime environment. Integrated into the Robotics Development Studio is the Visual Programming Language, which offers an intuitive and visual access into the programming of robots to beginners. The c't-Bot is a project of the magazine c't. The robot has been uses for several semesters in the robot laboratory of the University of Applied Sciences Hamburg. For more demanding tasks, performance of the robot is often not sufficient. So the idea arose to externalise labor-intensive tasks onto a powerful computer.

Inhaltsverzeichnis

1 Einführung

Die c't-Bots werden seit einigen Semestern an der Hochschule für Angewandte Wissenschaften Hamburg für studentische Projekte eingesetzt. Mit ihnen soll der Umgang mit Robotern erlernt werden. Die Herausforderung besteht darin, mit den begrenzten Ressourcen eines Roboters komplexe Aufgaben zu bewältigen. So wurde bereits Finden und Sortieren von Objekten, kooperatives Energiemanagement und Roboter-Fußball mit den c't-Bots realisiert.

Bisher wurde ausschließlich die Rechenleistung der c't-Bots für die Lösung der Aufgaben verwendet. Hier stößt man aber schnell an die Grenzen der Roboter und wünscht sich mehr Leistung, ohne die Roboterplattform verändern zu müssen. Hier hilft es, einen leistungsstarken PC als Koordinator einzusetzen, der z. B. die Positionen der Roboter erfasst, die Umweltbedingungen überwacht oder eine allgemeine Steuerungsoberfläche für den Benutzer anbietet.
Dabei kann das Robotic-Studio weiterhelfen. Mit dem Studio ist es möglich, den Aufbau des c't-Bots nachzubilden und den Roboter fernzusteuern. Damit kann die rechenintensive Arbeit auf einen leistungsstarken Rechner ausgelagert werden.
Weiterhin bietet das Robotic-Studio mit der Visual Programming Language ein intuitives Konzept, Programme für den Roboter zu modellieren, ohne tiefer gehende Kenntnisse von Programmiersprachen und Elektronik bzw. Mechanik des Roboters haben zu müssen.

Diese Arbeit stellt den c't-Bot und das Robotic-Studio im Detail vor. Anschließend wird ein Konzept entwickelt, mit dem der c't-Bot in das Robotic-Studio integriert wird.

Um den c't-Bot in das Robotic-Studio integrieren zu können, müssen auf folgende Fragen Antworten gefunden werden:
Wie können Daten zwischen dem c't-Bot und dem Robotic-Studio ausgetauscht werden?
Welche Aufgaben können vom c't-Bot und vom Robotic-Studio sinnvoll übernommen werden, bzw. wie können sich beide ergänzen?

Außerdem wird mit der Visual Programming Language ein interessanter Weg vorgestellt, wie Laien an die Programmierung von Robotern herangeführt werden können.

1.1 Robotik

Die Robotik befasst sich mit der Entwicklung und Steuerung von Robotern. Ein Roboter ist ein mechanisches Gerät, dass sich mittels Aktoren bewegen kann und durch Sensoren seine Umwelt wahrnimmt. Mittlerweile gibt es eine weite Spanne an unterschiedlichen Robotern, wie z. B. industrielle Roboter, Spielzeugroboter, humanoide Roboter und viele mehr.
Diese Roboter teilen sich in zwei Kategorien auf: Autonome Roboter, die selbständig handeln, bzw. ein einprogrammiertes Programm abarbeiten, und ferngesteuerte Roboter, die vom Menschen gesteuert werden.
Die Robotik umfasst Teilgebiete der Informatik, Elektrotechnik und des Maschinenbaus.

Roboter werden mittlerweile in vielen verschiedenen Bereichen eingesetzt. Der größte Bereich ist dabei die Industrie. Hier werden Roboter für Aufgaben eingesetzt, die für den Menschen zu gefährlich sind, oder bei Arbeitsschritten, die immer wieder die gleichen Tätigkeiten fordern, z. B. am Fließband, um Autos zu montieren, oder in Regallagern, um Gegenstände automatisch ein- und auszulagern.
In der Medizin werden Roboter für Operationen eingesetzt und im Haushalt finden sich bereits Roboter die den Boden staubsaugen oder den Rasen mähen. Neuerdings werden auch Roboter entwickelt, die ältere Menschen betreuen und unterstützen sollen. Im Hobbybereich und in der Lehre und Forschung kommen oft kleinere Roboter zum Einsatz, mit denen die grundlegenden Prinzipien der Robotik erlernt werden können.

1.2 Visuelle Programmiersprachen

Ein Bild sagt mehr als tausend Worte. Dieser Gedanke liegt den visuellen Programmiersprachen zu Grunde. Intuitive Konzepte sollen dem Programmierer helfen, sich auf die Modellierung von Programmen konzentrieren zu können. Bestandteile von Programmiersprachen wie Schleifen und Bedingungen werden durch grafische Symbole dargestellt. Für den Benutzer soll so eine Umgebung geschaffen werden, in der er in kurzer Zeit die Logik und Struktur eines Programms erfassen kann.
Den Begriff *Visuell* definiert Schiffer wie folgt:

> Visuell ist die Bezeichnung für jene Eigenschaft eines Objekts, durch die mindestens eine Information über das Objekt, die für das Erreichen eines Handlungsziels unverzichtbar ist, nur durch das visuelle Wahrnehmungssystem des Menschen gewonnen werden kann. (Schiffer, 1998, S. 63)

1.3 Gliederung dieser Bachelorarbeit

Diese Bachelorarbeit ist in verschiedene Kapitel unterteilt, die aufeinander aufbauen.

In Kapitel 2 wird der c't-Bot vorgestellt. Die einzelnen Sensoren und Aktoren, über die der c't-Bot verfügt, werden vorgestellt. Zudem wird auf das Framework eingegangen, das für den c't-Bot entwickelt wurde.
Kapitel 3 geht auf die einzelnen Bestandteile des Robotics Studio ein. Die Grundlagen für die *Concurrency and Coordination Runtime* sowie für die *Decentralized Software Services* werden vorgestellt. Weiterhin wird ein Einblick in die *Visual Programming Language* gegeben sowie die Fähigkeiten des Simulators vorgestellt.

Um nun den c't-Bot in das Robotic-Studio integrieren zu können, sind verschiedene Voraussetzungen zu schaffen. Welche das sind, wird in Kapitel 4 genauer erläutert.

In Kapitel 5 werden die aus den vorherigen Kapiteln erlangten Erkenntnisse in die Praxis umgesetzt. Zudem werden die Erfahrungen mit dem c't-Bot und dem Robotic-Studio erläutert.

Den Schluss dieser Bachelorarbeit bildet das Kapitel 6. Hier wird auf die Vor- und Nachteile des c't-Bots und des Robotic-Studio eingegangen. Zusätzlich werden Aspekte angesprochen, auf die im Rahmen dieser Bachelorarbeit nicht eingegangen werden konnten.
Weiterhin werden Ideen vorgestellt, die das Ergebnis dieser Arbeit ergänzen können und interessante Projekte ermöglichen.

2 c’t-Bot

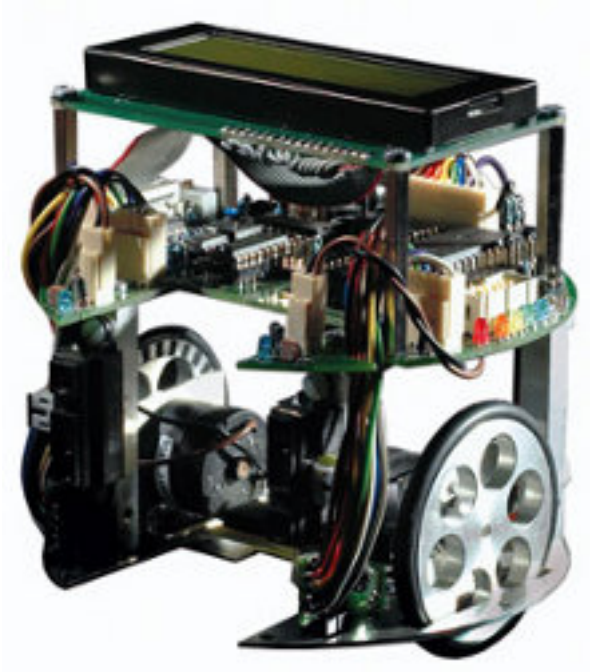
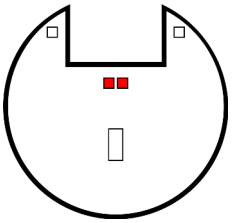


Abbildung 2.1: c’t-Bot (Quelle: heise, 2008)

Der c’t-Bot ist ein von der Zeitschrift c’t gestartetes Roboter Projekt. Es stellt einen fertigen Roboter mit Simulationsumgebung und Code Framework zur Verfügung. Der c’t-Bot ist in der Lage, einfache Aufgaben zu erledigen, wie Linien verfolgen, navigieren im Labyrinth oder Gegenstände einsammeln. Der c’t-Bot ist so konzipiert, dass auch Einsteiger ohne große Vorkenntnisse schnell zu ersten Ergebnissen kommen. Er wird als Bausatz geliefert, der mit einem einfachen Lötkolben und einem Schraubendreher schnell zusammen zubauen ist. Zusätzlich kann an den Roboter eine Klappe zum Schließen des Transportfachs und ein Erweiterungsmodul angebracht werden. Mit dem Erweiterungsmodul hat der Roboter Zugriff auf eine SD- oder MMC-Karte und ein (W)LAN-Modul. Durch das (W)LAN-Modul wird die RS232-Schnittstelle des c’t-Bots an ein bestehendes Netzwerk angebunden. An der HAW wurde ein zusätzliches Erweiterungsmodul entwickelt, dass mit einem AT90CAN128 Mikrocontroller von Atmel bestückt ist, der alle IO-Pins zur freien Verfügung stellt.

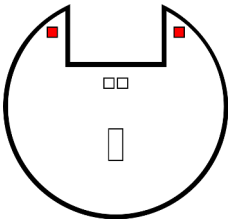
2.1 Sensoren

2.1.1 Liniensensoren



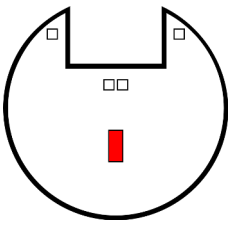
Auf der Unterseite des c't-Bots befinden sich mittig zwei CNY70 Reflexkoppler. Der CNY70 ist mit einer Infrarot-LED und einer Fotodiode ausgestattet. Er sendet infrarotes Licht aus, das von einer Oberfläche reflektiert wird. Das reflektierte Licht wird dann von der Fotodiode wahrgenommen. Je weiter die Oberfläche vom Sensor entfernt ist, desto weniger reflektiertes Licht wird von der Fotodiode erkannt.
Mit dem Sensor lassen sich dunkle von hellen Oberflächen unterscheiden. Die Fotodiode liefert dabei kein digitales Signal, sondern eine analoge Spannung.

2.1.2 Abgrundsensoren



Ebenfalls auf der Unterseite, jedoch rechts und links neben dem Transportfach, befinden sich zwei weitere CNY70 Reflexkoppler. Da sie sehr weit vorne am c't-Bot angebracht sind, lassen sich damit Abgründe erkennen. Damit kann vermieden werden, dass der Roboter vom Tisch fällt und Schaden nimmt. Die Abgrundsensoren sind nur dann als solche zu gebrauchen, wenn sich der Roboter auf hellem Untergrund bewegt. Ansonsten kann der Sensor nicht zwischen dem dunklen Untergrund und einem Abgrund unterscheiden.

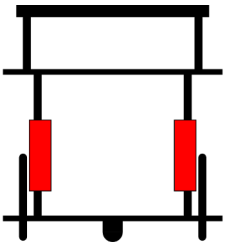
2.1.3 Maussensor



Auf der Unterseite des Roboters ist ein ADNS2610 Maussensor angebracht. Mit Hilfe des Maussensors kann der c't-Bot seine eigene Bewegungen überwachen und so u. a. Geschwindigkeit und Position ermitteln. Der Maussensor stellt hierfür die Positionsveränderungen gegenüber der letzten Messung in X- und Y-Richtung zur Verfügung. Der c't-Bot muss diese Werte lediglich über eine serielle Verbindung abfragen. Die Ermittlung der Differenzwerte wird vom Maussensor selber vorgenommen.

Der Maussensor hat eine Auflösung von 18 mal 18 Bildpunkten. Dieses Bild kann aus dem Sensor ausgelesen werden.

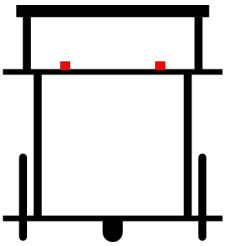
2.1.4 Distanzsensoren



An der Frontseite des c't-Bots sind rechts und links zwei Sharp GP2D12 Infrarot-Distanzsensoren montiert. Beide sind nach vorne ausgerichtet. Mit diesen Distanzsensoren kann der Roboter Gegenstände erkennen, die sich vor ihm befinden. Die Sensoren können Gegenstände im Abstand von 10 bis 80 cm erkennen. Die Entfernung geben die Sensoren über eine Signalspannung aus, die am Mikroprozessor des c't-Bots an einem Analog-Digital-Wandler angeschlossen sind. Bei einer Entfernung von 10 cm beträgt die Signalspannung des Sensors ca. 2,6 Volt, bei 80 cm ca. 0,4 Volt. Zu beachten ist, dass die Signalspannung bei Entfernungen unterhalb von 10 cm steil abfällt. Das kann zu Fehlern in der Auswertung führen. Zudem verläuft die Kurve der Signalspannung nicht linear.

In der c't-Bot Szene wurde bereits der SRF10 Ultraschallentfernungsmesser erfolgreich als Distanzsensor eingesetzt. Der SRF10 wird über den TWI-Bus angeschlossen. Der Ultraschallentfernungsmesser hat den Vorteil, dass er im Bereich von 4 cm bis 600 cm die Entfernung misst und die gemessene Entfernung in cm bereitstellen kann.

2.1.5 Helligkeitssensoren

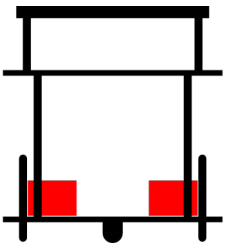


Der c't-Bot besitzt zwei MPY54C569 Fotowiderstände (LDR). Fotowiderstände bestehen aus einem Halbleitermaterial, das bei Lichteinfall seinen Widerstandswert ändert. Je mehr Licht auf den Fotowiderstand fällt, desto niedriger ist sein Widerstandswert.

Dadurch, dass der c't-Bot zwei dieser Sensoren besitzt, die jeweils auf der rechten und linken Seite angebracht sind, ist es möglich die Richtung zu ermitteln, aus der eine Lichtquelle kommt.

2.2 Aktoren

2.2.1 Motoren



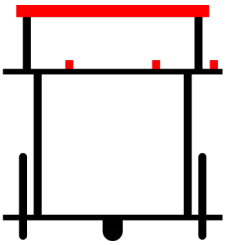
Der c't-Bot ist mit zwei Motoren ausgestattet. Mit diesen kann er sich im Raum bewegen. Neben vorwärts und rückwärts fahren kann er auch Kurven fahren oder sich sogar auf der Stelle drehen.

Dazu sind die Motoren mit einer H-Brückenschaltung versehen. Hierfür wird der IC L293D verwendet. Die Geschwindigkeitswerte der beiden Motoren werden in ein PWM-Signal umgesetzt und dem L293D zugeführt.

Die Räder des c't-Bots sind mit Encoderscheiben versehen, mit denen sich die tatsächliche Drehgeschwindigkeit der Räder überprüfen lässt.

Neben den beiden Motoren zur Fortbewegung kann der c't-Bot mit einem weiteren Motor ausgestattet werden. Dieser Motor ist ein Futaba S3107 Servomotor, an den eine Vorrichtung montiert wird, um das Transportfach des c't-Bots zu schließen.

2.2.2 Anzeigen



Auf dem Roboter kann ein LCD-Display montiert werden. Das Display kann 20 Zeichen in 4 Zeilen anzeigen. Das reicht aus, um z. B. die aktuellen Sensorwerte übersichtlich darzustellen oder für rudimentäres Debugging.
An der Seite und rechts und links neben dem Transportfach ist der c't-Bot mit LEDs bestückt. Sie geben auch aus der Ferne Aufschluss über den Zustand des c't-Bots.

Das LCD-Display und die LEDs sind über Schieberegister an den c't-Bot angeschlossen.

2.3 Framework

Zum Umfang des c't-Bots gehört auch ein komplettes Framework. Dieses bietet einen Zugriff auf die Grundfunktionen des Roboters sowie eine Laufzeitumgebung für ein verhaltensbasiertes Roboterprogramm.
Nach dem Einschalten des c't-Bots wird erst einmal ein Grundzustand initialisiert. Sämtliche Ein- und Ausgänge werden, ihrer Verwendung entsprechend, eingestellt. Danach beginnt

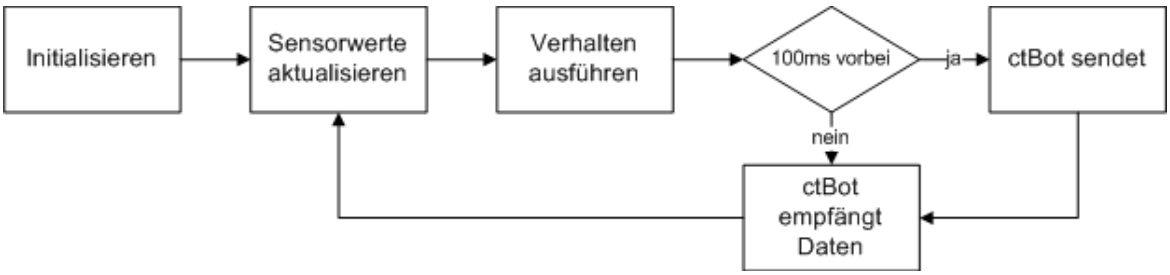


Abbildung 2.2: Programmablauf des c't-Bots

das Hauptprogramm des c't-Bots. In einer Endlosschleife werden zuerst die Sensordaten abgefragt und zur weiteren Verwendung abgespeichert. Danach werden die aktiven Verhalten des c't-Bots ausgeführt. Alle 100 ms aktualisiert der c't-Bot die LEDs und das Display. Weiterhin werden die aktuellen Sensordaten über die serielle Schnittstelle versendet. Anschließend wird der Empfangspuffer auf vollständig empfangene Kommandos überprüft. Wurde ein Kommando empfangen wird dieses evaluiert und die entsprechenden Aktionen werden ausgeführt. Danach fängt die Schleife wieder von vorne an.

2.3.1 Subsumption

Die Subsumption-Architektur, die 1986 von Rodney Brooks Brooks (1986) vorgestellt wurde, stellt die Logik eines Roboter als verhaltensbasierte Struktur dar. Damit kann die komplexe Struktur eines Roboters in mehrere Verhaltensmuster unterteilt werden, die in Schichten aufeinander aufbauen. Jedes Verhaltensmuster spiegelt einen Aspekt des Gesamtverhaltens des Roboters wider und berücksichtigt dabei die Verhalten in tiefer liegenden Schichten.
Ein Roboter, der seine Umwelt erforscht, würde z. B. in einer Ebene geradeaus fahren. In einer tiefer liegenden Schicht würde der Roboter darauf achten, nicht mit Hindernissen zu kollidieren, und in einer noch tiefer liegenden Schicht reagiert der Roboter auf Abgründe.
Alle Schichten erhalten als Eingangsdaten gleichzeitig die selben Sensorwerte des Roboters. Höhere Schichten fassen dabei die Reaktionen niedrigerer Schichten zusammen. Sollten zwei Schichten Einfluss auf die gleiche Ausgangsgröße haben, besitzt die untere Schicht die höhere Priorität. Fährt der Roboter z. B. auf einen Abgrund zu, sollte die höhere Schicht, die den Roboter geradeaus fahren lässt, der niedrigeren Schicht, die den Roboter vor einem Sturz in die Tiefe bewahrt, den Vortritt gewähren. Somit gilt allgemein, dass höhere Schichten eine niedrigere Priorität haben.

Mit der Subsumption-Architektur ist es möglich das gesamte Verhalten eines Roboters in mehreren kleinen Schritten zu entwickeln. Z. B. wird erst ein sehr einfaches Verhaltenssystem entwickelt und in der realen Welt getestet. Aufbauend auf diesem System werden dann iterativ weitere Verhalten hinzugefügt.

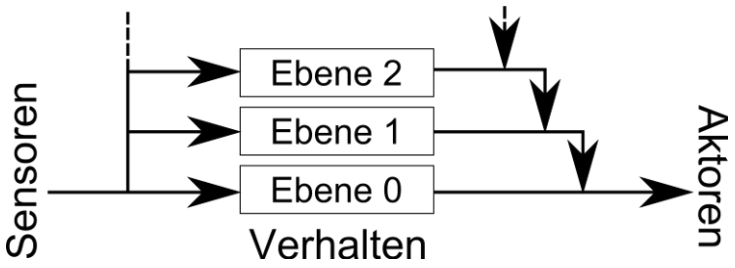


Abbildung 2.3: Subsumption Ebenen

Die Subsumption-Architektur, wie sie von Brooks vorgestellt wurde, sieht für jede Schicht einen eigenen Prozessor vor, in dem ein Zustandsautomat das Verhalten berechnet und ausführt. Aufgrund der beschränkten Ressourcen auf dem c't-Bot ist eine einfachere Variante der Subsumption-Architektur realisiert, die dennoch die Grundgedanken, mehrere Ziele und Sensoren, Robustheit und Erweiterbarkeit beinhaltet.

Ein Verhalten wird innerhalb des c't-Bots durch die Struktur Behaviour_t dargestellt.

```
typedef struct _Behaviour_t {
  void (*work) (struct _Behaviour_t *data)
  uint8 priority;
  struct _Behaviour_t *caller;
  uint8 active:1;
  uint8 subResult:2;
  struct _Behaviour_t *next;
}__attribute__ ((packed)) Behaviour_t;
```

Listing 2.1: Behaviour-Struktur

Das Feld **void** (*work) (**struct** _Behaviour_t *data) ist ein Pointer zu der eigentlichen Funktion, die dieses Verhalten beschreibt. Im Feld **uint8** priority ist die Priorität des Verhaltens festgehalten. Prioritäten können im Bereich von 0 bis 255 liegen, wobei 0 die niedrigste Priorität und 255 die höchste Priorität darstellt.

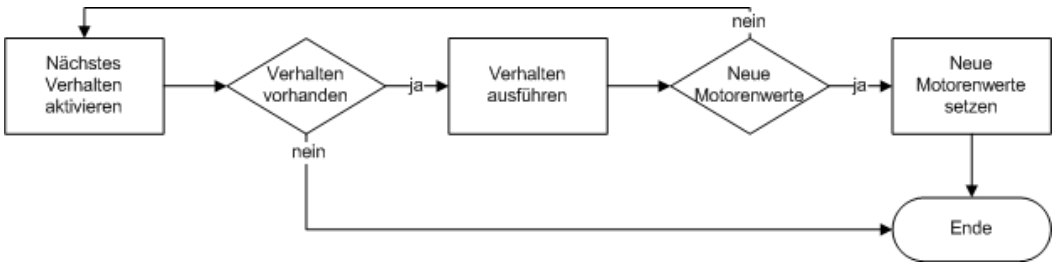


Abbildung 2.4: Verhaltensschleife

In der Funktion bot_behave(**void**), die in der Hauptschleife des Programms aufgerufen wird, werden die aktiven Verhalten nach ihrer Priorität aufgerufen. Das Verhalten mit der höchsten Priorität wird dabei als erstes aufgerufen. Sobald ein Verhalten die Werte für die Motoren setzen möchte, wird dies auch direkt ausgeführt und eventuelle Verhalten mit geringerer Priorität werden ignoriert.
Dadurch wird verhindert, dass Sicherheitsverhalten überschrieben werden. Andererseits werden niedere Verhalten solange nicht ausgeführt, wie sich der c't-Bot in einer kritischen Situation befindet. Das kann evtl. Nebenwirkungen zur Folge haben.

3 Robotic-Studio

Mit dem Microsoft Robotics Developer Studio (kurz Robotic-Studio) ist es möglich einen Roboter fern-zusteuern und in eine Softwareumgebung zu integrieren, die nicht auf dem Roboter läuft, sondern auf einem normalen Desktop-PC. Das Robotic-Studio ist eine Entwickler- und Laufzeitumgebung auf der Windows Plattform. Es unterstützt eine weite Palette an bereits existierenden Roboterplattformen, z. B. den LEGO NXT, den Roomba und weitere. Weiterhin ist es möglich, eigene Roboter in das Studio zu integrieren. Dabei stehen dem Programmierer das .NET Framework und die dadurch unterstützten Programmiersprachen zur Verfügung. In dem Robotic-Studio ist die VPL enthalten. Hiermit ist es Einsteigern möglich, einen Roboter zu steuern, ohne Kenntnisse über die Hardware haben zu müssen. Ist kein Roboter vorhanden, kann auf einen Simulator zurückgegriffen werden. Dieser kann neben dem Roboter auch eine Umwelt mitsamt physikalischen Effekten simulieren.

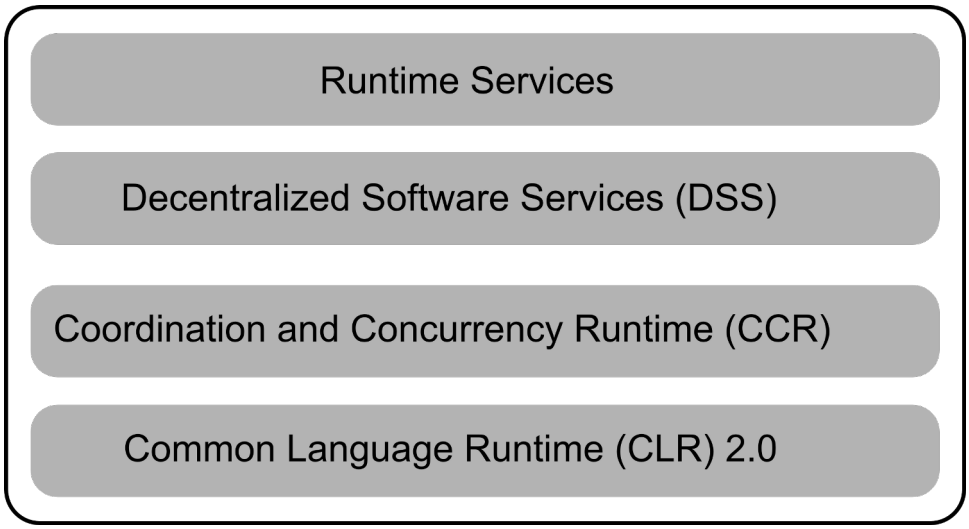


Abbildung 3.1: Robotic-Studio Runtime (Quelle: Morgan, 2008, S. 6)

3.1 Concurrency and Coordination Runtime

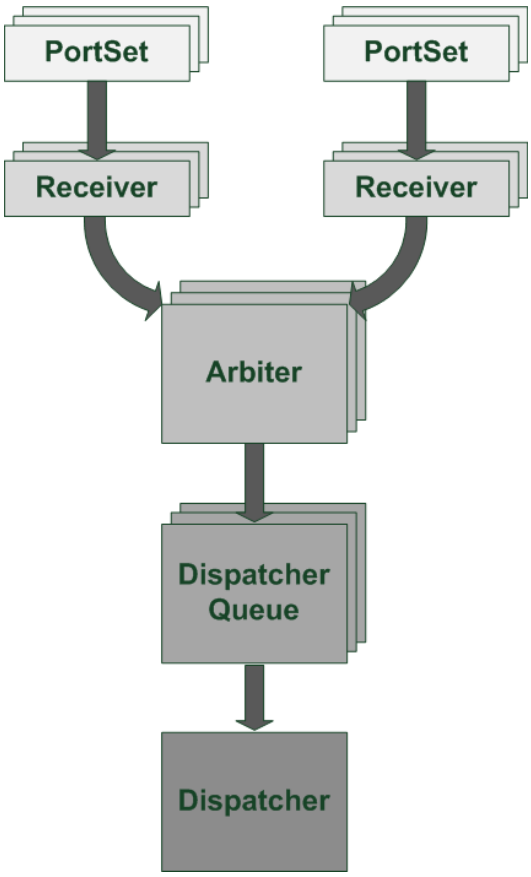
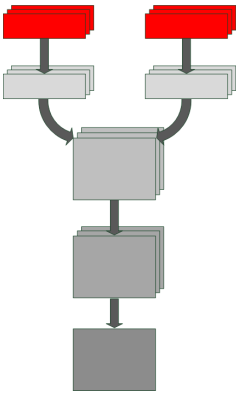


Abbildung 3.2: Scheduling Kette (Quelle: Johns und Taylor, 2008, S. 38)

Die Concurrency and Coordination Runtime (CCR) ist eine .NET Programmbibliothek, die die asynchrone und neben läufige Verwaltung von Operationen übernimmt. Sie ermöglicht ein Softwaredesign, in dem einzelne Services lose miteinander verbunden sind. Das Service orientierte Programm Modell verwaltet Nachrichten zwischen den einzelnen Programmbestandteilen und übernimmt die manuelle Handhabe von Threads, Locks, Semaphoren usw. Die CCR ist momentan für die Programmiersprachen C#, Python und Visual Basic verfügbar. Die CCR ist in drei Gruppen aufgeteilt: Die *Ports* und *PortSets*, die *Coordination Primitives* und das *Task Scheduling*.

3.1.1 Ports und PortSets



Die generische Port-Klasse ist eine FIFO-Queue, die jeden gültigen Typen der .NET Runtime (CLR) aufnimmt.

```
Port<int> portInt = new Port<int>();
```

Ports können auf zwei verschiedene Arten verwendet werden. Die erste und einfachste Variante ist, den Port als ganz normale FIFO Queue zu benutzen. Mit port.Post(10) kann dem obigen Port ein Element angefügt werden und mit int i = portInt.Test() wird das nächste Element abgerufen und aus der Liste entfernt.
Die zweite Möglichkeit wird im Robotic-Studio am häufigsten verwendet. Der Port wird als aktive Queue verwendet. D.h., die Queue wird mit einem oder mehreren Arbitern verbunden. Neu eintreffende Elemente erzeugen eine Task, die zur Ausführung gescheduled wird. Diese Eigenschaft der Ports ist die Grundlage für die Nebenläufigkeit des Robotic-Studio.

Da ein Port immer nur einen Typ aufnehmen kann, gibt es die Klasse PortSet. Diese kann mehrere Ports mit jeweils unterschiedlichen Typen aufnehmen. Ein PortSet stellt die Gesamtheit der Operationen dar, die ein Service anderen Services anbietet.

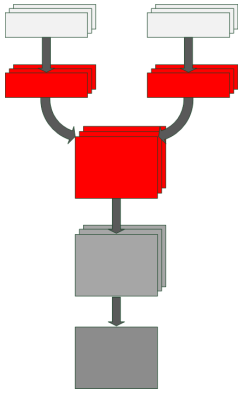
```
PortSet genericPortSet
  = new PortSet<int,
                double,
                string>
        ()
```

Jeder Service definiert eine eigene Klasse für ihren Main-Port (s. Kapitel 3.2.5), der von der Klasse PortSet abgeleitet ist. Dieser Main-Port gibt Auskunft darüber, welche Operationen ein Service anbietet.
Ein Service-PortSet **muss** die Operation *DsspDefaultLookup* implementieren. Diese Operation dient dazu, Informationen über einen Service zu erhalten.
In einem Motor-Service für den c't-Bot bietet es sich, an eine Operation zur Verfügung zu stellen, mit der die Geschwindigkeit eingestellt werden kann. Hierfür fügt man einfach die Operation *SetMotoSpeed* dem Service-PortSet hinzu.
Der Service-PortSet würde dann folgendermaßen aussehen:

```
public class myPortSet
        : PortSet<DsspDefaultLookup,
                  SetMotorSpeed>
{}
```

Zu beachten ist, dass ein PortSet, der auf Windows CE unter dem Compact Framwork laufen soll, maximal 8 generische Ports aufnehmen kann. Möchte man ein PortSet mit mehr als 8 Ports erstellen, kann der nicht generische Constructor verwendet werden. Hier werden die einzelnen Ports dann mittels typeof(int) übergeben. Die Standardausgabe des .NET Frameworks unterstützt immerhin PortSets mit bis zu 20 Einträgen. Im normalen Gebrauch sollte das ausreichen. Zur Not kann auch hier mit typeof(int) gearbeitet werden.

3.1.2 Coordination-Primitives



Die Coordination-Primitives sind in der Arbiter-Static-Class zusammengefasst. Ein Arbiter ist ein wichtiges Werkzeug, um das asynchrone und neben läufige Verhalten der Services zu ermöglichen.
Ein Arbiter ist ein Bindeglied zwischen einem Port bzw. einem PortSet (s. 3.1.1) und einem Service-Handler (s. 3.2.6).

Es stehen dabei unterschiedliche Varianten zur Verfügung. Mit dem *Single-Item-Receiver* kann auf einen bestimmten Antworttyp reagiert werden. Mit einem *Choice* kann zwischen zwei unterschiedlichen Antworttypen unterschieden werden. Das ist u. a. dann nützlich, wenn eine Antwort einen Erfolg oder einen Misserfolg darstellen kann. *Joins* bieten die Möglichkeit, auf mehrere Antworten zu warten. Sobald alle Antworten eingetroffen sind, reagiert das *Join*.

```
Activate<ITask>(
        Arbiter.Receive<serial.Notify>
                (true, _ctBotSerialNotify, NotifyHandler)
);
```

Listing 3.1: Receive-Arbiter

Für die serviceeigenen Operationen und Handler werden automatisch Receive-Handler beim Starten des Services erzeugt. Möchte ein Service Nachrichten empfangen, für die der Service keine eigene Operation im Operations-Port definiert hat, muss ein Arbiter per Hand erzeugt werden.
In Listing 3.1 ist ein Arbiter für das Subscribe-Modell (s 3.2.7) dargestellt. Der Parameter _ctBotSerialNotify ist ein PortSet des veröffentlichenden Services, in dem die Nachrichten eintreffen. Der Parameter NotifyHandler ist die Handler-Methode, die die eintreffende Nachricht bearbeitet.

3.1.3 Task-Scheduling

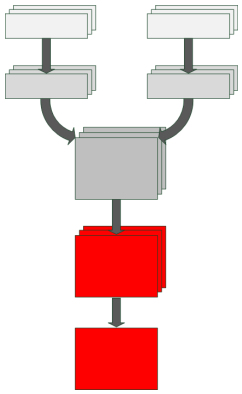
Wichtig ist das Scheduling der eintreffenden Nachrichten in einem Service. Hierfür stehen drei Klassen zur Verfügung.

Task und IterativeTask Diese beiden Klassen sind vom Interface ITask abgeleitet. Nur Klassen die ITask implementieren können gescheduled werden.

DispatcherQueue DispatcherQueue ist eine FIFO Queue von Tasks. Sie kann sich entweder am Thread-Pool der CLR bedienen oder aber einen Dispatcher der CCR benutzen.

Dispatcher Der Dispatcher bearbeitet Threads und kümmert sich um das Load-Balancing der Tasks aus einer oder mehreren Instanzen der Dispatcher-Queue.

```
Dispatcher dispatcher = new Dispatcher(0, "sample dispatcher");
```



Der erste Parameter für den Dispatcher-Konstruktor gibt an, wie viele Threads pro CPU der Dispatcher gleichzeitig unterstützt. Der Standardwert "0"bedeutet, dass genauso viele Threads erzeugt werden, wie CPUs vorhanden sind . Es werden jedoch immer mindestens zwei Threads erzeugt.
An einen Dispatcher können beliebig viele Dispatcher-Queues angebunden werden. Der Dispatcher wählt selbstständig aus, aus welcher Queue er welche Task zur Ausführung übernimmt.

```
DispatcherQueue taskQueue
  = new DispatcherQueue("sample queue", dispatcher);

Port<int> port = new Port<int>();
Arbiter.Activate(taskQueue,
  Arbiter.Receive(true, port, delegate(int item)
    {
      Console.WriteLine(item);
    )
);

port.Post(5);
```

Listing 3.2: Dispatcher-Queue mit Receive-Arbiter

Eine Dispatcher-Queue wird einem Dispatcher zugeordnet. Um Nachrichten, die auf einem Port auflaufen, automatisch der Queue zuzuordnen, wird die Queue mit einem Receive-Arbiter verbunden. Der Arbiter kombiniert die empfangene Nachricht mit dem verbundenen Handler zu einer Task, die der Dispatcher-Queue hinzugefügt wird.

3.2 Decentralized Software Services

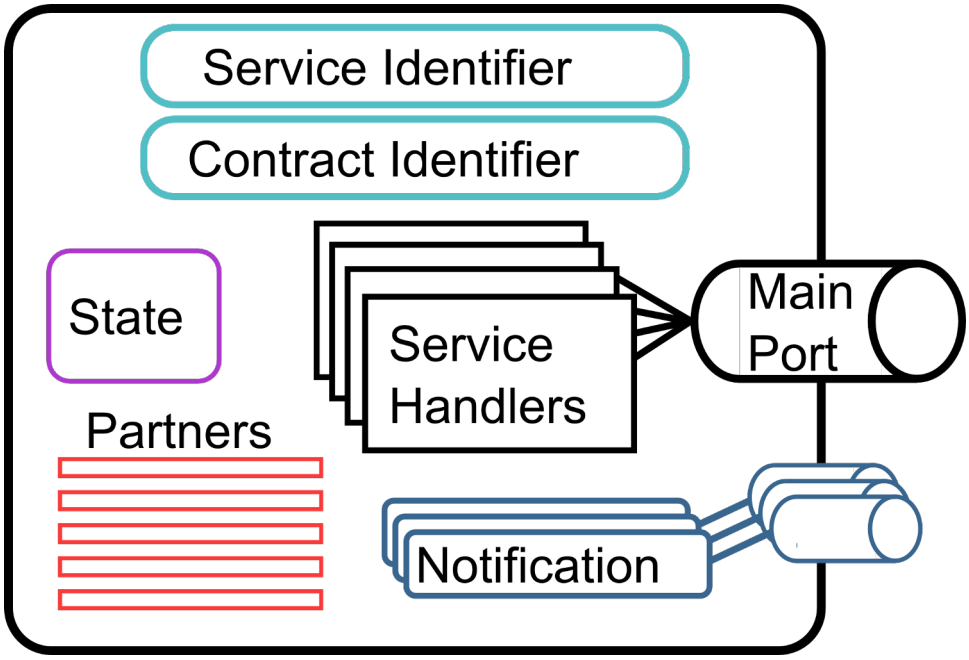
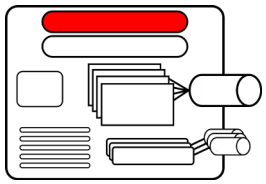


Abbildung 3.3: DSS-Modell (Quelle: Microsoft, 2008a)

Die Decentralized Software Services (DSS) setzen auf der CCR auf und bilden eine leichtgewichtige Laufzeitumgebung. Innerhalb der DSS werden zustandsorientierte Services verwaltet. Dafür stellt die DSS Methoden zur Verfügung, um Services zu starten, zu stoppen und zu manipulieren. Services basieren auf dem Representational State Transfer (REST) Fielding (2000) und bilden eine zusammenhängende Funktionseinheit. Um einen Roboter in das Robotic-Studio zu integrieren, werden seine einzelnen Komponenten in separate Services aufgeteilt.

Für die Kommunikation der Services untereinander steht neben HTTP das DSS-Protokoll (DSSP) zur Verfügung. Das DSSP basiert auf SOAP und ermöglicht die Übertragung von Daten, Fehlerbehandlung und ereignisgesteuerte Aktionen.
Im Gegensatz zu Web-Services sind im DSSP Zustand und Verhalten getrennt und der Zugriff auf Services wird über dessen Operationen ermöglicht. Der Zustand eines Services ist von jedem anderen Service jederzeit abrufbar. Dadurch können direkte Kontrollstrukturen zwischen den Services aufgebaut werden, ohne einen Kontrolleur implementieren zu müssen.

3.2.1 Service-Identifiers

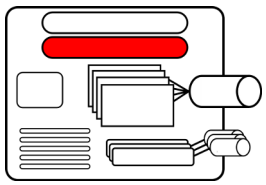


Wenn eine Instanz eines Services erschaffen wird, wird ihm eine URI zugewiesen. Diese URI ermöglicht das genaue Auffinden einer Service-Instanz in einem DSS-Node. Mit dem Identifier wird die Kommunikation zwischen Services und sogar der Zugriff über einen Web-Browser ermöglicht. Der Identifier gibt keine Auskunft über den Zustand oder das Verhalten eines Services, sondern nur über seine Identität. Wie der Service-Indentfier genau aussieht, kann über mehrere Wege beeinflusst werden.

1. mit einem Service-Manifest
2. bei einem Create-Request
3. über das ServicePort-Attribut

Angenommen es existiert ein Service, der den c't-Bot koordiniert. Möchte man mehr als einen c't-Bot gleichzeitig steuern, werden hierfür zwei Instanzen des Services benötigt. Damit die Instanzen, und damit die c't-Bots, auseinander gehalten werden können, erhält jede Instanz ihren eigenen eindeutigen Service-Identifier.

3.2.2 Contract-Identifier



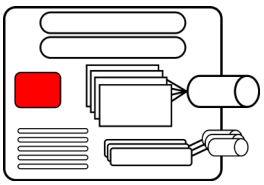
Der Contract ist eine zusammengefasste Beschreibung einer Service-Implementierung. Der Contract-Identifier gibt Auskunft über das Verhalten eines Services. Mit der Hilfe eines Contracts wird für den Service eine Proxy Dynamic Link Library (DLL) erstellt, auf die andere Services zugreifen können. Dies hat den Vorteil, dass bei Änderungen an einem Service der Contract nicht verändert werden muss. Somit müssen Services, die auf die Proxy-DLL verweisen, nicht neu übersetzt werden.
Standardmäßig wird beim Erstellen eines neuen Services diesem ein Contract Identifier nach folgendem Schema zugewiesen.

```
http://schemas.tempuri.org/[year]/[month]/[name].html
```

Der Identifier kann natürlich auch nach Belieben angepasst werden,
sollte aber eindeutig bleiben.

Ein Service kann neben seinem eigenen Contract-Identifier auch einen generischen Contract-Identifier (s. 3.2.8) angeben, den er implementiert.
Ein Service, der diesen Service zum Partner hat, kann sich nun aussuchen, in welcher Rolle er den Service benutzen möchte: als einen generischen Service oder als einen speziellen Service.

3.2.3 Service-State



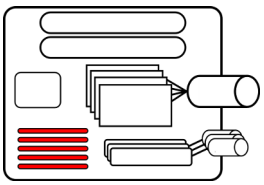
Der Service-State ist der Zustand eines Services zu jedem beliebigen Zeitpunkt. In einem Service-State werden alle Informationen gespeichert, die für diesen Service wichtig sind. So werden z. B. in einem Service für Reifen Druck, Temperatur oder Rollwiderstand gespeichert. Jeder Wert, den ein Service erhält, überwacht oder verändert, sollte als Teil des Service-State gespeichert werden. Da es sich bei dem Service-State im Kontext der Threads um einen kritischen Abschnitt handelt, sollten alle Operationen, die auf den Service-State zugreifen, exklusive Zugriffsrechte erhalten.

Wird auf einen Service über einen Webbrowser zugegriffen, wird der Zustand des Services als Teil eines XML-Streams ausgegeben. Per XSLT kann eine komfortable Anzeige gestaltet werden, die eine Interaktion mit dem Service gestattet.

Der Zustand eines Services ist als eigene Klasse implementiert, die mit dem *DataContact-Attribut* versehen ist. Jedes Feld in einem Zustand muss mit dem Attribut *DataMember* versehen werden, sonst wird das Feld nicht in der Assembly-Datei enthalten sein.

Der Service, der für den Datenaustausch mit dem c't-Bot zuständig ist, kann in dem Service-State die jeweils zuletzt empfangenen und versendeten Kommandos speichern. Zudem sind im Service-State Informationen über Geschwindigkeit und Einstellung der Datenübertragung gut aufgehoben.
Services für einzelne Sensoren speichern im Service-State die aktuellen Sensorwerte.

3.2.4 Service-Partner



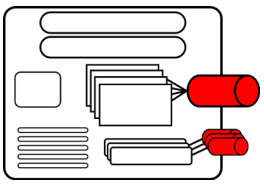
Da Services nur lose miteinander verbunden sind, ist es schwierig für einen Service zu wissen, ob ein anderer Service existiert, bzw. wo im Speicher dieser genau liegt. Den Partner-Service aufzufinden ist Aufgabe der Laufzeitumgebung. Um einen anderen Service als Partner einzurichten, ist es notwendig im Programmcode den Partner mit dem *Partner-Attribut* zu versehen.
Das Partner-Attribut enthält eine Menge an Optionen, mit denen angegeben werden kann, wie zwingend ein Service für einen anderen Service ist. So kann erst nach vorhandenen Instanzen eines Services gesucht werden und bei Nichtvorhandensein eine neue Instanz erzeugt werden oder der eigentliche Service wird erst gar nicht erzeugt.

```
[Partner("Serial", Contract =serial.Contract.Identifier,
  CreationPolicy = PartnerCreationPolicy.UseExistingOrCreate)]
private serial.CtBotSerialOperations _ctBotSerialPort
  = new serial.CtBotSerialOperations();
```

Listing 3.3: Service-Partner

Der c't-Bot wird im Robotic-Studio nicht als ein monolitischer Service implementiert. Vielmehr werden einzelne Komponenten in eigenständige Services aufgeteilt. Ist ein Empfänger-Service dafür zuständig, Sensordaten vom c't-Bot zu empfangen und ein Sensor-Service ist dafür zuständig, einen bestimmten Sensor zu überwachen, wird der Sensor-Service den Empfänger-Service als Partner eintragen. So kann der Sensor-Service auf die Sensordaten, die der Empfänger-Service bereithält, zugreifen.

3.2.5 Main-Port und Notification-Port



Der Main-Port ist ein CCR-Port (s. 3.1.1). Services greifen nicht direkt über Methoden auf einander zu, sondern versenden Nachrichten, die am Main-Port eintreffen. Der Main-Port ist ein Private-Member, der mit dem *Service-Port-Attribut* versehen ist.

```
[ServicePort("/sample", AllowMultipleInstances=false)]
private MySampleOperations _mainPort = new MySampleOperations();
```

Listing 3.4: Main Port

Der Parameter "/sample" des Service-Port-Attributs gibt den Namen des Services, mit Host und Port vorangestellt, an. Mit Localhost und Port 50000 würde der Name in diesem Beispiel wie folgt aussehen:

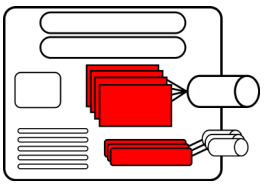
```
http://localhost:50000/sample
```

Der Parameter AllowMultipleInstances legt fest, ob von dem Service eine oder mehrere Instanzen erstellt werden können. Ist der Parameter auf true gesetzt, wird dem Service Identifier der jeweiligen Instanz eine eindeutige GUID angehängt

```
http://localhost:50000/sample/d88441c9-8319-407c-b554-0b0bfd90050b
```

Die Notification-Ports sind alle von Hand erstellte Ports, die mit einem Arbiter verbunden sind. Ein Notification-Port ist z. B. notwendig, um Benachrichtigungen eines Services zu erhalten, die er im Rahmen des Subscribe-Modells als Nachrichten veröffentlicht.

3.2.6 Service-Handler



Die Service-Handler sind die Teile des Services, die das eigentliche Verhalten bestimmen. In den Service-Handlern werden die Nachrichten, die am Main-Port empfangen wurden, verarbeitet.

```
[ServiceHandler(ServiceHandlerBehavior.Concurrent)]
public IEnumerator<ITask> GetHandler(Get get){
    get.ResponsePort.Post(_state);
    yield break;
}
```

Listing 3.5: Get-Handler

Der obige Handler liefert auf Anfrage den aktuellen Service-State an den anfragenden Service zurück. Das Attribut [**ServiceHandler**(ServiceHandlerBehavior.Concurrent)] deklariert die Methode als Handler-Methode. Der optionale Parameter ServiceHandlerBehavior .Concurrent gibt an, dass mehrere Anfragen parallel bearbeitet werden dürfen. ServiceHandlerBehavior.Exclusive würde hingegen einen exklusiven Zugriff auf diese Methode garantieren. Das ist wichtig, um kritische Daten vor einer gleichzeitigen Manipulation durch parallele Verarbeitung von Nachrichten zu schützen.

Die Service-Handler werden beim Erstellen einer neuen Service-Instanz mit dem Operation-Port verbunden. Dies passiert automatisch beim Aufruf der Start() Methode, die in jedem Service vorhanden sein muss. In dieser Methode wird die Methode base.Start() aufgerufen. Neben einer Log-Info und dem Eintragen des Services in das Node-Directory, wird für jeden Handler ein persistenter Receive-Arbiter erzeugt. Für den Get-Handler sieht der Arbiter folgendermaßen aus:

```
Arbiter.Receive<Get>(true, _mainPort, GetHandler);
```

Wird ein Arbiter mit einem Notification-Port verbunden, muss auch ein Handler angegeben werden. Der Handler wird durch eine normale Methode dargestellt. Diese Methode hat als Parameter den Nachrichtentyp des Notification-Ports.

```
private void myHandler(NachrichtenTyp nt)
{
  //Hier Code einfügen
}
```

3.2.7 Subscription-Modell

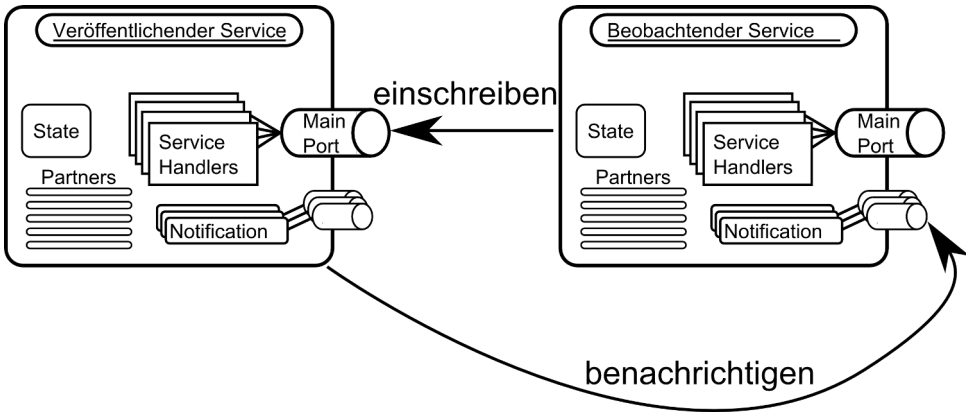


Abbildung 3.4: einschreiben und benachrichtigen

Damit ein Service nicht andauernd den Zustand eines anderen Services abfragen muss, gibt es das Subscription-Modell.
Das Subscription-Modell ist eine Umsetzung des Observer-Patterns[1] . Damit andere Services sich bei einem Service einschreiben können, um bei Änderungen benachrichtigt zu werden, sind ein paar Vorbereitungen nötig.

Der veröffentliche Service muss einen *Subscription-Manager* als Service-Partner einrichten. Wichtig ist, dass für jeden veröffentlichenden Service eine eigene Instanz des Subscription-Manager erstellt wird. Das wird mit der *=PartnerCreationPolicy.CreateAlways* im Partner-Attribut sicher gestellt.

```
[Partner("SubMgr", Contract=sm.Contract.Identifier,
        CreationPolicy=PartnerCreationPolicy.CreateAlways)]
sm.SubscriptionManagerPort _subMgrPort
        = new sm.SubscriptionManagerPort();
```

Listing 3.6: Subscription Manager

Nun muss für den Service eine Subscribe-Nachricht definiert werden. Diese Nachricht muss auch in dem Operations-Port des Services eingetragen werden. Anschließend wird noch ein Handler für die Nachricht erstellt. Dieser Service-Handler empfängt die Nachrichten anderer Services.

[1] Auch Publisher-Subscriber-Pattern genannt (Buschmann, S. 338ff)

```
[ServiceHandler(ServiceHandlerBehavior.Concurrent)]
public void SubscribeHandler(Subscribe subscribe)
{
    SubscribeHelper(_subMgrPort, subscribe.Body, subscribe.
        ResponsePort);
}
```

Listing 3.7: Subscribe-Handler

Ändert nun der veröffentlichende Service seinen Status, gibt er dieses dem Subscription-Manager bekannt, welcher dann alle in seiner Liste eingetragenen Beobachter über die Veränderung informiert.

Selective-Subscription Häufig will ein Beobachter aber nicht über jede Veränderung an einem Service informiert werden, sondern nur über eine konkrete oder einige wenige. Hierfür stellt das Robotic-Studio die so genannte *Selective Subscription* bereit. Hier sendet der Beobachter beim Einschreiben eine Liste mit, in der die Veränderungen stehen, an der er interessiert ist. Beispielsweise kann sich ein Service nur über Änderungen an den Abstandssensoren eines Roboters informieren lassen. Treten nun Änderungen in einem Service auf, erstellt dieser eine Liste mit den geänderten Daten und übergibt sie dem Subscription-Manager. Der Subscription-Manager informiert nun lediglich die Beobachter, die sich für diese speziellen Änderungen interessieren.

Event-Notification-Filtering Eine weiter Variante ist das *Event-Notification-Filtering*. Hiermit ist eine noch feinere Abstufung gegenüber dem Selective-Subscription möglich. Der Beobachter kann sehr spezielle Filter erstellen, die es ihm ermöglichen, z. B. nur über Änderungen am rechten Abstandssensor, unterhalb von 20 cm informiert zu werden. Der veröffentlichende Service muss hierfür nur dem Subscription-Manager eine Struktur übergeben, anhand der er die Filter erkennen kann. Der Beobachter übergibt beim Anmelden eine Liste mit den gewünschten Filtern. Der veröffentlichende Service hat im Folgenden keine Arbeit mehr. Das Filtern wird vollständig vom Subscription-Manager übernommen.

Damit der Beobachter über Veränderungen benachrichtigt werden kann, muss dieser entsprechend erweitert werden. Zum einen muss er den zu beobachtenden Service als Partner eintragen. Zum anderen muss er für jede Art der Veränderung einen eigenen Handler implementieren. Das ist nötig, da ein Service unterschiedliche Arten der Veränderung erfahren kann (Update, Delete, Add). Jede dieser Veränderungen wird mit einer eigenen Nachricht behandelt. Da jeder Handler nur einen Nachrichtentyp empfangen kann, muss für jeden Nachrichtentyp ein eigener Handler implementiert werden.

3.2.8 Generic-Service

Generic Services bilden in etwa das nach, was in C++ die Templates und in C# oder Java die Interfaces sind. Mit einem generischen Service wird eine Funktionshülle erstellt, die nur die Schnittstellen deklariert, aber keine Funktion bereitstellt.
Konkret kann mit einem generischen Service angegeben werden, welche Operationen ein Service unterstützen soll, wie der Service-Contract aussieht und welche Felder der Service-State enthalten soll. Natürlich ist ein Service, der diesen generischen Service implementiert, nicht auf diese Vorgaben beschränkt, sondern kann sie beliebig erweitern. Der Compiler erkennt nicht, ob für jede Operation des generischen Services ein entsprechender Handler implementiert wurde. Darauf zu achten, ist die Aufgabe des Programmierers.

Mit Hilfe von generischen Services kann ein Service verschiedene Rollen annehmen. Er stellt neben seinem eigenen Service-Contract auch den des generischen Contracts zur Verfügung. Wählt ein anderer Service nun diesen Service als Service-Partner aus, kann er sich entscheiden, ob er ihn als generischen Service behandeln möchte oder als einen konkreten Service.

Der Vorteil von generischen Services ist, dass die Services, die sie implementieren, beliebig untereinander austauschbar sind. Dazu sind nur wenige Änderungen am Programmcode notwendig. Welcher konkrete Service verwendet wird, ist auch über das Service-Manifest einstellbar.

Für den c't-Bot bedeuten generische Services, dass einzelne Komponenten, wie z. B. Sensoren, in der Software ohne großen Aufwand augetauscht werden können.

3.3 Service-Manifest

Manifeste sind XML-Dateien die vom Manifest-Loader-Service ausgewertet werden. Manifeste werden beim Start eines Services ausgewertet und enthalten Informationen, wie der Service konfiguriert werden soll.

Manifeste können entweder per Hand bearbeitet werden oder aber mit einem XML-Editor wie dem Manifest-Editor, der dem Robotic-Studio beiliegt. Der Editor bietet einen Zugriff auf andere Services welche bequem in das Manifest eingefügt und konfiguriert werden können.

Mit jedem Service Projekt wird auch immer ein Manifest erstellt. Dieses sieht dann so aus:

```
<Manifest
  xmlns="http://schemas.microsoft.com/xw/2004/10/manifest.html"
  xmlns:dssp="http://schemas.microsoft.com/xw/2004/10/dssp.html"
  xmlns:s="http://schemas.tempuri.org/2007/04/sample.html">
  <CreateServiceList>
    <ServiceRecordType>
      <dssp:Contract>
        http://schemas.abc.de/sample.html
      </dssp:Contract>
    </ServiceRecordType>
  </CreateServiceList>
</Manifest>
```

Listing 3.8: Service-Manifest

Services müssen nicht in einem Manifest niedergeschrieben worden sein, um erstellt zu werden. Sie sind auch nicht auf ein einziges Manifest beschränkt.
Einzelne Services innerhalb eines *CreateServiceList-Elementes* werden mit dem Element *ServiceRecordType* angegeben. Es kann eine beliebige Anzahl an *ServiceRecordType* Elementen angegeben werden. Jede davon erzeugt eine neue Instanz eines Services.

Normalerweise ergibt sich der Name eines Services aus dem ServicePort-Attribut des MainPorts (s. 3.2.5). Mit dem Manifest kann aber auch ein anderer Name für die Instanz angegeben werden. Dazu wird dem *ServiceRecordType* ein *Service-Element* hinzugefügt.

```
<dssp:Service>http://localhost/NewSampleName</dssp:Service>
```

Hat ein Service Partner-Services, können diese durch ein Manifest ausgetauscht werden. In dem *ServiceRecordType* des Services wird dann das Element *Partner* eingefügt.

```
<ServiceRecordType>[caption={Service-Manifest mit Partner-Element}]
  <dssp:Contract>http://schemas.abc.de/sample.html</dssp:Contract>
  <dssp:PartnerList>
    <dssp:Partner>
      <dssp:Service>http://localhost/PartnerService</dssp:Service>
      <dssp:Name>p:Partner</dssp:Name>
    </dssp:Partner>
  </dssp:PartnerList>
</ServiceRecordType>
```

Innerhalb des *Partner-Elements* wird dann der Name, der in dem Partner-Attribut des Services angegeben wurde, eingetragen und der Service, der als Partner agieren soll.
Einen anderen Partner anzugeben ist dann hilfreich, wenn z. B. ein Sensor gegen einen anderen Sensor ausgetauscht werden soll. Die beiden Sensoren müssen natürlich die gleichen Operationen implementieren. Der Tausch per Manifest erspart den Eingriff in den Code.

Da Services sich prinzipiell auf verschiedenen DSS-Nodes auf verschiedenen Rechnern befinden können, kann über ein Manifest angegebn werden, auf welchem entfernten DSS-Node sich der zu erzeugende Service befinden soll. Soll z. B. ein Partner-Service auf einem entfernten Rechner erzeugt werden, wird das durch das *ServiceDirectory-Element* angezeigt.

```
<dssp:Partner>
  <dssp:Name>s:MyPartner</dssp:Name>
  <dssp:ServiceDirectory>
    http://example.org:50000/directory
  </dssp:ServiceDirectory>
</dssp:Partner>
```

Listing 3.9: Partner-Element mit explizitem Service-Directory

Manifeste bieten noch eine Reihe weiterer Möglichkeiten, Services zu konfigurieren. Eine vollständige Auflistung sämtlicher Möglichkeiten würde an dieser Stelle aber zu weit gehen.

Existieren für den einen Roboter verschiedene Hardware-Kombinationsmöglichkeiten, kann für jede einzelne Kombination ein Manifest mit den jeweiligen Komponenten erstellt werden.

3.4 System-Services

Die System-Services können von jedem Service erreicht und benutzt werden. Sie sind für jede Programmiersprache erreichbar, welche die DSS-Nodes unterstützt. Mit diesen System-Services stehen den Services eine Reihe nützlicher Funktionen zur Verfügung. So können sie auf das Dateisystem zugreifen, Logging-Funktionen oder Diagnose-Werkzeuge benutzen. Durch die prinzipielle Verteilung der Services auf mehrere Rechner können diese Funktionen auch aus der Ferne verwendet werden.
Die meisten dieser Services sind bereits erreichbar, wenn ein DSS-Node gestartet wurde. Dazu kann entweder von der DSS-Kommandozeile

```
dsshost /p:50000
```

eingegeben werden oder *Run DSS Node* im Startmenü ausgewählt werden. Natürlich kann auch ein Service gestartet werden. Dann wird automatisch ein DSS-Node mitgestartet.

Control-Panel-Service

Das Control-Panel gibt ausführliche Informationen über einen DSS-Node. Mit dem Control-Panel können Services gestartet und wieder beendet werden. Der Service ist über den Webbrowser über die Adresse und Portnummer erreichbar, auf der er läuft, z. B. http :// localhost :50000. Über das Control-Panel sind weiter System-Services erreichbar.

Constructor-Service

Mit dem Constructor-Service können neue Instanzen von Services erzeugt werden. Üblicherweise wird zur Erzeugung von neuen Services ein Manifest oder das Partner-Attribut verwendet. Ein Service kann aber auch direkt mit dem Construktor-Service kommunizieren, wenn er einen neuen Service erzeugen möchte.

Manifest-Loader-Service

Mit dem Constructor-Service wird ein Service-Manifest 3.3 ausgewertet und die entsprechenden Services gestartet. Der Manifest-Loader bietet über einem Webbrowser ein übersichtliches Interface, an auf der er die Ergebnisse ausgibt, die beim Starten eines Services mit einem Manifest entstehen.

Console-Output-Service

Mit dem Console-Output-Service können Logging-Informationen über den Webbrowser angezeigt werden. Die Log-Einträge können dabei in drei Kategorien unterteilt werden: normaler Text, Warnungen und Fehlermeldungen. Somit lassen sich Log-Informationen nicht nur durch eine lokale Systemkonsole oder eine Log-Datei anzeigen, sondern auch bequem über das Internet vom anderen Ende der Welt.

Security-Manager-Service

Für jeden DSS-Node lassen sich differenzierte Sicherheitseinstellungen vornehmen. Der Zugriff auf einen DSS-Node lässt sich in drei Einstellungen konfigurieren.

Für den c't-Bot können so verschiedene Benutzergruppen eingerichtet werden. Während eine Benutzergruppe einen vollen Zugriff auf die Funktionen des c't-Bots erhält, bekommt eine andere Benutzergruppe lediglich die aktuellen Sensordaten angezeigt.

Authentifizierung Wenn der Zugriff auf einen DSS-Node eine Authentifizierung benötigt, wird eine Authentifizierung nach NT-LAN-Manager oder besser benötigt. Der Benutzername wird dann mit einer Liste erlaubter Benutzer oder Gruppen verglichen und entsprechend wird der Zugriff gewehrt oder abgelehnt.

Service-Assembly-Loading Das Laden und Erstellen von Services kann durch erforderliche Zertifikate beschränkt werden.

Network-Access-Permission Die Verbindung zwischen Services kann auf drei verschiedenen Wegen aufgebaut werden. Zum einen kann die Verbindung über ein Partner-Attribut hergestellt werden, zum anderen über einen entsprechenden Eintrag in eine Manifest-Datei. Der dritte Weg ist ein Forwarder der mit DsspServiceBase.ServiceForwarder<T>() oder DsspServiceBase.ServiceForwarderUnknownType() zur Laufzeit erstellt wurde.
Bei der letzten Variante benötigt ein Service eine *DssNetworkPermission*-.NET-Code-Access-Security-(CAS)-Erlaubnis, um einen Forwarder auf einen Service herzustellen, der auf einem anderen DSS-Node läuft.

3.5 Visual Programming Language

Abbildung 3.5: Hello World in der VPL

Die Visual Programming Language (nachfolgend VPL genannt) ist eine grafische Entwicklungsumgebung für das Robotic-Studio. Mit der VPL ist es für Einsteiger sowie für Fortgeschrittene möglich, bestehende Services zu verwenden und neue Services zu entwickeln, um so einen Roboter zu steuern.

Ein Programm in der VPL wird als *Diagramm* bezeichnet. Ein Diagramm besteht aus mehreren Activities und Services. Diese einzelnen Blöcke werden, mit Datenpfaden untereinander verbunden, über die Nachrichten verschickt. Das gesamte Diagramm stellt einen Datenfluss dar. D. h., dass die Daten von einem Block zum nächsten weiter gereicht werden, bis sie beim letzten Datenblock angekommen sind.
Eine Activity bzw. ein Service hat immer nur einen einzigen Eingangsport, an dem nur ein Datenpfad angeschlossen werden kann. Der Eingangsport kann aber mehrere Daten entgegennehmen. Genauso können in einer Nachricht mehrere Daten zur gleichen Zeit übermittelt werden. Ist es notwendig, dass von mehr als einer Quelle Nachrichten an einen Block gesendet werden, muss ein *Merge* oder ein *Join* vor den Eingang geschaltet werden.
Ausgänge gibt es in zwei unterschiedlichen Varianten. Der *Ergebnis-Ausgang* liefert ein direktes Ergebnis für die Daten, die am Eingang angekommen sind und innerhalb des Blocks berechnet werden. Dieser Ausgang eignet sich daher für alle Arten von Berechnungen, bei denen das Ergebnis sofort berechnet werden soll.
Der zweite Ausgang ist der *Benachrichtigungs-Ausgang*. Dieser Ausgang folgt dem Subscription-Modell aus der DSS. An diesem Ausgang wird eine Nachricht erzeugt, ohne dass dem Block zuvor Daten am Eingang zur Verfügung gestellt werden müssen. Ein Block, der regelmäßig Nachrichten an seinem Benachrichtigungs-Ausgang zur Verfügung stellt, eignet sich dazu am Anfang eines Diagramms zu stehen.
Steht am Anfang eines Diagramms kein Block mit einem Benachrichtigungs-Ausgang, sondern z. B. ein Variablen-Feld, wird dieser feste Wert als erste Nachricht an den dahinter liegenden Block gesendet.

Welche unterschiedlichen Ausprägungen eine Activity haben kann ist in den folgenden Unterkapiteln genauer aufgeführt.

3.5.1 Activity

Eine Activity ermöglicht die Entwicklung eigener Programme und diese als einen zusammenhängenden Block zu kapseln. Eine Activity kann über einen Eingangs-Port mit Daten versorgt werden und gibt entweder ein Ergebnis aus oder benachrichtigt andere Activities oder Services über Zustandsänderungen.

3.5.2 Calculate

Diese Funktion ermöglicht es, auf die Werte im Datenflussdiagramm die grundlegenden Rechenoperationen Addition, Subtraktion, Multiplikation, Division und Modulo anzuwenden. Weiterhin können mit der Calculate-Activity Zeichenketten erzeugt werden.

3.5.3 Bedingungen

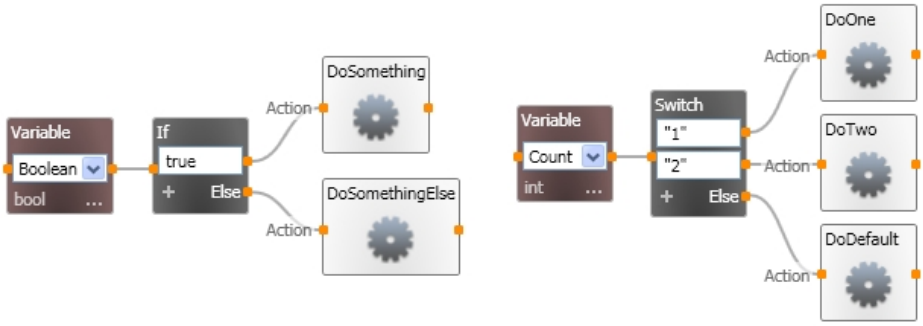


Abbildung 3.6: Bedingungen in der VPL

Bedingungen funktionieren wie in jeder anderen Programmiersprache auch. Einer If- oder switch-Bedingung wird ein Wert übergeben, welcher ausgewertet wird. Die möglichen Ergebnisse dieser Auswertung führen dann in Verzweigungen weiter im Programm.

3.5.4 Join

Activities können mehrere Eingangsparameter gleichzeitig aufnehmen. Diese Parameter müssen aber in einer einzigen Nachricht verpackt sein. Es kann vorkommen, dass diese Daten im Programm an unterschiedlichen Stellen berechnet werden. Die Join-Activity nimmt diese einzelnen Daten auf und kombiniert sie zu einer einzigen Nachricht, welche dann an

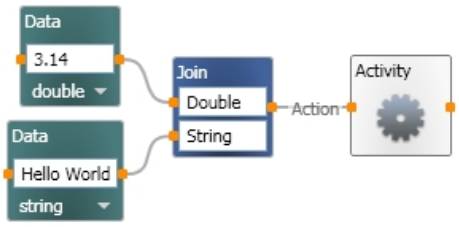


Abbildung 3.7: Join-Activity

die Activity weitergegeben werden kann. Die Nachricht wird erst abgesendet, wenn alle Daten bei der Join-Activity eingetroffen sind.

3.5.5 Merge

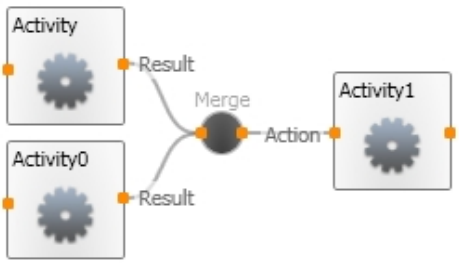


Abbildung 3.8: Merge-Activity

Es kann vorkommen, dass an verschiedenen Stellen des Programms Daten des gleichen Typs errechnet werden und diese derselben Activity übergeben werden sollen. Eine Activity nimmt auf der Eingangsseite aber immer nur eine Nachricht auf. Das Element Merge kann auf der Eingangsseite mehrere Nachrichten des gleichen Typs aufnehmen und diese dann an die Activity weitergeben.
Hinweis: Ein Merge verknüpft die ankommenden Nachrichten nicht, sondern sendet sie nur an die nachfolgende Activity weiter. Hat ein Merge z. B. zwei Eingänge und auf beiden Eingängen kommen Nachrichten an, dann leitet das Merge erst die eine und dann die andere Nachricht weiter. Möchte man zwei Nachrichten miteinander verknüpfen oder gleichzeitig in einer Activity ankommen lassen muss ein Join verwendet werden.

3.5.6 Schleifen

Schleifen sind kein eigenständiges Element in der VPL. Sie lassen sich aber mit einfachen Mitteln realisieren. Abbildung 3.9 zeigt, wie eine einfache Schleife aussieht, die einen Integer-

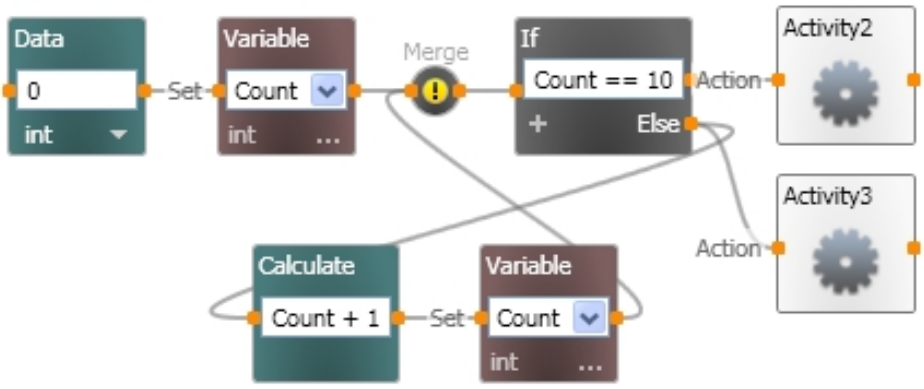


Abbildung 3.9: Schleife in der VPL

Wert inkrementiert. Im Robotic-Studio ist es nicht nötig, auf Ereignisse zu pollen oder Busy-Loops zu verwenden. Hier kann auf den Subscription-Mechanismus zurückgegriffen werden. Dadurch verringert sich der Einsatz von Schleifen erheblich. Schleifen lassen sich auch durch rekursive Architekturen ersetzen. Das erhöht die Übersicht und dadurch evtl. auch das Verständnis der Programme.

3.6 Simulator

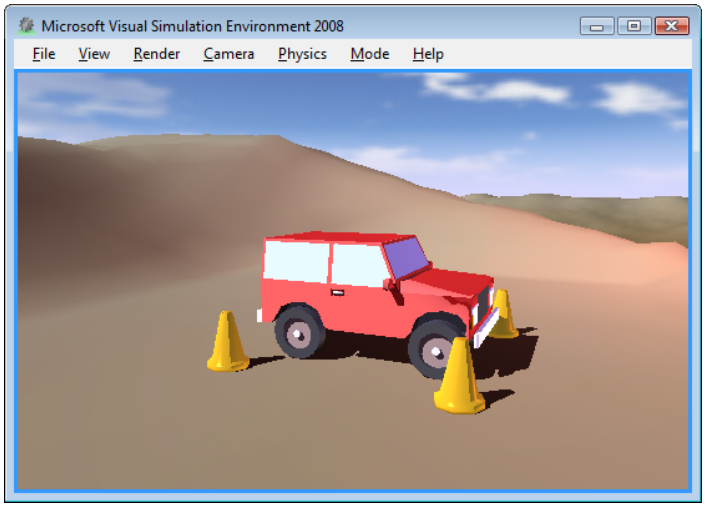


Abbildung 3.10: Simulator (Quelle: Microsoft, 2008d)

Bestandteil des Robotic-Studio ist ein Simulator. Mit dem Simulator lassen sich virtuelle Welten erschaffen, in denen realistische erscheinende physikalische Bedingungen herrschen.

Die Anschaffung von Roboterhardware ist oft sehr teuer oder nicht realisierbar. Vielleicht möchte man auch unterschiedliche Roboter ausprobieren oder Situationen testen, die schädlich für den Roboter sind. In größeren Arbeitsgruppen kommt es auch häufig vor, dass verschiedene Aufgaben in Untergruppen gelöst werden, die verfügbare Hardware aber nur in begrenzter Stückzahl für die gesamte Gruppe vorrätig ist.

Hier hilft es, wenn der Roboter in einer sicheren und günstigen Softwareumgebung simuliert werden kann. Ein Simulator bietet die Vorteile, dass in einer kontrollierten virtuellen Welt Aufgaben und Situationen getestet werden können, ohne den Roboter in Gefahr zu bringen oder zusätzliche Kosten für Hardware zu erzeugen.

Bei allen Vorteilen birgt der Einsatz eines Simulators auch ein paar Probleme.
Der Simulator bietet eine saubere und kontrolliere Welt. Es treten keine Störungen auf und der Akku des Roboters wird niemals leer. Die echte Welt ist aber oft voller Tücken und unberechenbar. Hier lösen sich Verbindungen, Kontakte korrodieren und der Roboter verhält sich anders als erwartet. Möchte man Services für einen realen Roboter erstellen bleibt also nichts anderes übrig, als hin und wieder das Programm am echten Objekt zu testen.

3.6.1 Engine und API

AGEIA-PhysX-Engine Der Simulator ist in der Lage realistische physikalische Effekte darzustellen. Hierfür benutzt der Simulator die AGEIA-PhysX-Engine.
Für die PhysX-Engine sind eigene Physikbeschleunigerkarten erhältlich. Mit ihnen wird die CPU des Rechners entlastet und die Effekte können schneller berechnet werden. Mittlerweile werden solche Physikbeschleuniger in Grafikkarten integriert.

Ist keine Harware vorhanden, die die PhysiX-Engine unterstüzt, werden die Effekte per Software berechnet.

Microsoft DirectX und XNA DirectX ist eine Sammlung von Systemschnittstellen, die einen Zugriff auf Grafik, Sound und Eingabegeräte ermöglichen.

XNA wurde entwickelt, um einheitliche Schnittstellen für die Spieleprogrammierung für PC und XBox360 zu haben. Dem Programmierer werden einige Aufgaben abgenommen, um sich besser auf die eigentliche Programmierung des Spieles konzentrieren zu können.

3.6.2 Grundbestandteile

Eine Simulation wird in einem Service implementiert. Die Simulation ist also den gleichen Freiheiten und Beschränkungen wie ein Service ausgesetzt. Neben ein paar zwingend erforderlichen Bestandteilen kann eine Simulation mit beliebig vielen Objekten versehen werden.

Ein Simulation-Service besteht mindestens aus folgenden Bestandteilen.

Simulation-Engine-Service Die Engine ist nötig, um Gegenstände darzustellen und physikalische Effekte zu berechnen. Zudem wird mit diesem Service die grafische Oberfläche der Simulation erzeugt.

Managed-Physics-Engine-Wrapper Der Engine-Wrapper kapselt die tieferen Funktionen der Physik-Engine ab und stellt eine vereinfachte API zur Verfügung

Entities Entities füllen die Welt mit Leben. Hardware, Gegenstände, usw., werden alle durch Entities dargestellt.

Der Simulation-Engine-Service ist der wichtigste Bestandteil einer Simulation. Er muss im jeweiligen Simulationsservice als Partner eingetragen werden und stellt alle notwendigen Operationen für den Simulator zu Verfügung.

```
[Partner("Engine",
  Contract = engineproxy.Contract.Identifier,
  CreationPolicy = PartnerCreationPolicy.UseExistingOrCreate)]
private engineproxy.SimulationEnginePort _engineStub =
  new engineproxy.SimulationEnginePort();
```

Listing 3.10: Partner-Engine

Eine Welt besteht aus einem Himmel, dem Erdboden und Objekten, die die Welt bevölkern. Um einen Himmel und einen Boden hinzuzufügen ist folgender Code nötig

```
void AddSky()
{
  SkyDomeEntity sky =
    new SkyDomeEntity("skydome.dds", "sky_diff.dds");
  SimulationEngine.GlobalInstancePort.Insert(sky);

  LightSourceEntity sun = new LightSourceEntity();
  sun.State.Name = "Sun";
  sun.Type = LightSourceEntityType.Directional;
  sun.Color = new Vector4(0.8f, 0.8f, 0.8f, 1);
  sun.Direction = new Vector3(0.5f, -.75f, 0.5f);
  SimulationEngine.GlobalInstancePort.Insert(sun);
}

void AddGround()
{
  HeightFieldEntity ground = new HeightFieldEntity(
    "simple ground", // name
    "03RamieSc.dds", // texture image
    new MaterialProperties("ground",
      0.2f, // restitution
      0.5f, // dynamic friction
      0.5f) // static friction
    );
  SimulationEngine.GlobalInstancePort.Insert(ground);
}
```

Listing 3.11: AddSky und AddGround

Ein Himmel besteht aus einem Würfel, der die restliche Welt einfasst. Mit der *SkyDomeEntity* wird ein Himmel für den Simulator erzeugt. Die beiden Parameter im Konstruktor geben die Texturen und Informationen für die Beleuchtung an.

Damit die Welt nicht dunkel bleibt, wird eine Lichtquelle installiert. Mit der *LightSourceEntity* kann eingestellt werden, welche Farbe die Lichtquelle haben soll und in welche Richtung das Licht ausgesendet wird.

Böden werden mit der *HeightFieldEntity* erzeugt. Der oben angegebene Konstruktor erzeugt eine flache Oberfläche mit der angegebenen Textur *"03RamieSc.dds"*. Mit dem *MaterialProperties-Parameter* werden die Eigenschaften Federwirkung und Reibungswiderstand des Bodens festgelegt.

Nun haben wir eine leere Welt. Um einen Würfel als Objekt in die Welt einzufügen, ist folgender Code nötig:

```
void AddBox(Vector3 position)
{
  Vector3 dimensions =
    new Vector3(0.2f, 0.2f, 0.2f); // meters

  SingleShapeEntity box = new SingleShapeEntity(
    new BoxShape(
      new BoxShapeProperties(
        100, // mass in kilograms.
        new Pose(), // relative pose
        dimensions)), // dimensions
      position);

  box.State.MassDensity.Mass = 0;
  box.State.MassDensity.Density = 0;

  box.State.Name = "box";

  SimulationEngine.GlobalInstancePort.Insert(box);
}
```

Listing 3.12: AddBox

Allen Entities ist gemein, dass sie der Simulator-Engine mit der asynchronen Methode Insert (Entity entity) übergeben werden.
In diesem Beispiel wird mit der *SingleShapeEntity* eine Box in die Welt eingefügt. Werden für eine Entity keine Werte für die Masse oder Dichte angegeben, wird die Entity als statisches Objekt behandelt. Die Entity hat eine unendlich hohe Masse und kann nicht bewegt werden. Wichtig ist zu beachten, dass für jede einzelne Entity ein eigener einzigartiger Name gewählt wird. Entities können später nur über ihren Namen referenziert werden.

Mit dem Simulator lassen sich weit komplexere Gegenstände als Boxen erschaffen. Diese sogenannten *MultiShapetEntities* werden durch die Kombination aus mehreren einfachen geometrischen Formen wie Würfel, Kugeln, usw. aufgebaut. Mit Texturen versehen, kann eine *Multi-Shaped-Entity* z. B. zu einem Roboter geformt werden.

Um den c't-Bot simulieren zu können, muss er aus vielen einzelnen Teilen zusammengefügt werden. Für jedes Bauteil, wie Sensor, Aktor oder Platine, muss eine eigene Entity erzeugt werden. Für manche Sensoren und für den Motor stehen bereits vorgefertigte Entities bereit.

4 Analyse

4.1 Aufgabenstellung

Wenn man sich längere Zeit mit dem c't-Bot intensiv auseinandersetzt und komplexere Aufgabestellungen anstrebt, stößt man schnell an die Grenzen des c't-Bots. Möchte man z. B. mehrere c't-Bots eine gemeinsame Aufgabe bewältigen lassen, müssen die c't-Bots Aufgaben übernehmen, die sich auf allen Robotern wiederholen, aber nicht unmittelbar mit der Lösung der Aufgabe zu tun haben. Solche Aufgaben belegen aber häufig einen Großteil der Ressourcen des Roboters und führen zu Problemen bei der eigentlichen Aufgabenlösung.
Es ist daher sinnvoll, den c't-Bot von den Aufgaben zu befreien, die er nicht unbedingt selber erledigen muss, und diese auf einen leistungsfähigen Rechner auszulagern.

Es wird somit eine Lösung gesucht, die den c't-Bot entlastet. Hier kommt das Robotic-Studio ins Spiel. Das Robotic-Studio ist in der Lage, mit relativ wenig Aufwand jede denkbare Hardware zu integrieren und zu steuern.

Hierfür müssen die anfallende Arbeiten sinnvoll auf den c't-Bot und das Robotic-Studio aufgeteilt werden. Notwendige Aufgaben, wie Sensoren auslesen und Aktoren stellen, kann nur von dem c't-Bot erledigt werden, wohingegen das Robotic-Studio die Sensordaten auswerten und entsprechende Werte für die Aktoren berechnen kann. Eine gesunde Balance muss gefunden werden.

Die VPL bietet einen leichten und intuitiven Einstieg in die Programmierung. Kai Rosseburger hat im Rahmen seiner Bachelorarbeit Rosseburg (2007) eine Entwicklungsumgebung für Roboter-basierten Informatikunterricht entwickelt. Wenn man seinen Ansatz verfolgt und mit dem c't-Bot ähnlich vorgehen will, müssen die Schnittstellen der Services und des c't-Bots soweit vorbereitet werden, dass diese einfach benutzt werden können, ohne spezielle Anforderungen an den Benutzer zu stellen.

Als weiteres Highlight des Robotic-Studio kann der Simulator angesehen werden. Dadurch, dass er auf den gleichen Grundfunktionen wie jeder andere Service basiert, ist ein leichter Einstieg möglich. Zudem bilden zahlreiche Beispiele bestehender Roboter die Grundlage, eigene Roboter für den Simulator zu entwickeln. Dabei kann der Roboter wie gewohnt Stück

für Stück entwickelt und erweitert werden. Es müssen nur die Komponenten tatsächlich implementiert werden, die für die Aufgabe wichtig sind.

Für erfahrene Programmierer bietet die Integration des c't-Bots in das Robotic-Studio schier endlose Möglichkeiten. Das Robotic-Studio bietet einen Zugriff auf das komplette .NET Framework, wodurch dem Programmierer eine riesige Palette an Werkzeugen zur Verfügung steht, auf die er zurückgreifen kann. Was noch nicht vorhanden ist, kann selber hinzugefügt werden.

4.2 Vorgehensweise

Um die einzelnen Aufgaben lösen zu können, müssen sie Schritt für Schritt abgearbeitet werden.

Zuerst muss eine Verbindung zwischen c't-Bot und Robotic-Studio hergestellt werden. Diese Verbindung dient dazu, die Sensorwerte des c't-Bots an das Robotic-Studio zu senden. Umgekehrt müssen Befehle vom Robotic-Studio an den c't-Bot gesendet werden.
Diese Verbindung muss sowohl auf der Hardware-Ebene, als auch auf der Software-Ebene realisiert werden.

Ist die Datenverbindung zwischen c't-Bot und Robotic-Studio hergestellt, muss entschieden werden, wo welche Aufgaben gelöst werden können. Das Robotic-Studio soll den c't-Bot entlasten, kann aber nicht alle Aufgaben selber übenehmen.

4.3 Kommunikation

Die Kommunikation zwischen c't-Bot und PC ist ein wichtiger Bestandteil für die Integration des Roboters in das Robotic-Studio. Der Roboter muss seine Sensordaten an den PC senden können und der PC seine Befehle an den Roboter. Die Anforderungen an die Kommunikation sind dabei recht hoch:

- schnell
- zuverlässig
- fehlerfrei
- bidirektional
- Ressourcen schonend
- allgemein verfügbar

Um eine vernünftige Kommunikation zwischen c't-Bot und Robotic-Studio sicherstellen zu können, müssen die Daten so schnell wie möglich ausgetauscht werden. Das Robotic-Studio darf nicht zu lange auf aktuelle Sensorwerte warten müssen, sonst sind die Reaktionen des Robotic-Studio evtl. bereits veraltet, bevor diese überhaupt den c't-Bot erreicht haben. Dabei müssen die Daten auch zuverlässig und fehlerfrei übertragen werden. Eine schnelle Reaktion des Robotic-Studio nützt nicht, wenn die Daten verloren gehen, bzw. fehlerhaft beim c't-Bot ankommen.
Da der c't-Bot und das Robotic-Studio parallel, aber nicht synchron arbeiten, ist für jede Senderichtung ein eigener Kanal nötig. Damit können Daten gleichzeitig vom c't-Bot zum Robotic-Studio und umgekehrt gesendet werden. Prinzipiell ist es denkbar, das Senden und Empfangen über einen einzigen Kanal abzuwickeln, allerdings erhöht sich dadurch der Aufwand enorm und der Datendurchsatz pro Richtung verringert sich.
Die Kommunikation sollte auch mit Mitteln, die in den meisten Haushalten zu finden sind, bzw. leicht und günstig anzuschaffen sind, realisierbar sein. Eine spezielle Lösung ist wahrscheinlich aus technischer Sicht sinnvoller, Anfängern wird damit aber der Einstieg in die Robotik unnötig erschwert oder gänzlich unattraktiv gemacht.
Die Übertragung der Daten sollte möglichst keine Fehler beinhalten. Erkennen und Korrigieren von Fehlern ist mit zusätzlichem Aufwand verbunden, der die ohnehin schon knappen Ressourcen des c't-Bots strapaziert.

4.3.1 Hardware

Für die Datenübertragung zwischen c't-Bot und PC stehen zwei grundsätzliche Varianten zur Verfügung: Mit oder ohne Draht.
Eine drahtgebundene Kommunikation hat den Vorteil, dass sie recht einfach und preiswert zu realisieren ist. Meistens werden einfach nur zwei Leitungen benötigt. Der Atmeg32 des c't-Bots bietet mit einer RS232, einer TWI und einer SPI Schnittstelle ausreichend Möglichkeiten zur Kommunikation. Zur Not lässt sich auch eine eigene Variante über die IO-Ports realisieren.
Der Nachteil dieser Variante ist die beschränkte Bewegungsfähigkeit des Roboters. Durch den angeschlossenen Draht ist zum einen die Reichweite des Roboters beschränkt. Zum anderen stellt ein Draht auf der Fahrbahn für den c't-Bot ein großes Hindernis dar, das er kaum, bzw. gar nicht überwinden kann.
Bei einer drahtlosen Kommunikation fallen diese Aspekte nahezu weg. Der c't-Bot hängt an keiner Schnur, an der er sich aufhängen kann. Ein Problem sind evtl. die Reichweiten der einzelnen Funkstandards. Hier müssen die Positionen der Sender und Empfänger für den jeweiligen Einsatz berücksichtigt werden. Allerdings ist eine drahtlose Kommunikation auch mit vielen Nachteilen behaftet. Für eine Kommunikation per Funk ist das Übertragungsmedium die Luft. Allerdings ist dieses Medium bereits durch vorhandene Datenübertragungen

stark belastet, sei es der nächst gelegene Handymast, ein bestehendes WLAN-Netzwerk oder eine Funktastatur. Zu jeder Zeit werden sehr viele Informationen übertragen. All diese Funkstrahlen können die Kommunikation zwischen c't-Bot und Robotic-Studio stören oder im schlimmsten Fall ganz verhindern.

Ausschlaggebend für die Wahl der Kommunikationsart sind die zur Verfügung stehenden Ressourcen des c't-Bots. Seine Ein- und Ausgänge sind alle bereits belegt. Nur die Pins der RS232-Schnittstelle des AtMega32 sind noch unbenutzt. Die Pins der TWI-Schnittstelle werden für SDA und RCLK der Schieberegister benutzt. Die SPI-Schnittstelle wird für den Maussensor benötigt. Allerdings können sowohl die TWI als auch die SPI doppelt belegt werden. Hierfür müssen die verschiedenen Komponenten des c't-Bots über ihre Enable-Leitungen (de-)aktiviert werden.
Die SPI Schnittstelle ist bereits mit dem Maussensor und mit dem SD/MMC-Karteninterface doppelt belegt. Eine dritte Belegung würde den effektiven Datendurchsatz enorm verringern. Somit fällt die SPI-Schnittstelle weg.
Der TWI-Bus ist zwar mit den Schieberegistern belegt, da diese aber nur für die LEDs, das LCD-Display und die Enable-Leitungen benötigt werden, könnte man den Bus zumindest teilweise nutzen. Wird der c't-Bot als einziger Master eingesetzt kann die Leitung durch geschickte Programmierung störungsfrei sowohl für den TWI-Bus, sowie für die Schieberegister verwendet werden. Wird der c't-Bot nur als Slave auf dem TWI-Bus eingesetzt, ist der Aufwand erheblich größer. Der c't-Bot muss nun jederzeit auf reinkommende Datenpakete reagieren können. Dazu muss entweder ständig den TWI-Bus aktiviert sein oder ein strenger Zeitplan erarbeitet werden, der vorschreibt, wann der c't-Bot den TWI-Bus aktivieren muss. Aufgrund der beschränkten Kapazitäten des eingesetzten Mikrocontrollers und der hohen Anforderungen an die Kommunikation kommt ein Einsatz des TWI-Busses nicht in Frage.
Übrig bleibt die RS232-Schnittstelle. Sie hat einige Vorteile. So ist sie ein weit verbreiteter Standard, einfach in der Handhabung, und es existieren eine Menge Produkte auf dem Markt, die auf der RS232-Schnittstelle aufbauen. Es gibt einfache RS232 auf Funk Adapter, die auf dem 433 MHz Band arbeiten, aber auch leistungsfähigere, die über Bluetooth, ZigBee oder WLAN arbeiten. Das Erweiterungsmodul des c't-Bots bietet mit dem Lantronix WiPort die Möglichkeit, den Roboter über die RS232-Schnittstelle an ein WLAN anzubinden. Im Roboterlabor der Hochschule für Angewandte Wissenschaften Hamburg wurden bereits erfolgreich ZigBee-Module für die c't-Bots eingesetzt. Das WLAN Modul des c't-Bot Erweiterungsboards bietet den Vorteil, dass der Roboter sehr einfach in ein bestehendes Netzwerk integriert werden kann. Die Reichweite des WLANs ist für die typischen Einsatzgebiete des c't-Bots mehr als ausreichend. Der c't-Bot kann so z. B. im Keller arbeiten und vom Dachboden oder sogar vom sonnigen Palmenstrand aus ferngesteuert werden. Nachteilig ist allerdings der erhöhte Strombedarf, der durch ein WLAN Modul entsteht. Der Lantronix WiPort hat immerhin einen Spitzenstrombedarf von 650mA. Ist das WLAN-Modul gerade nicht beschäftigt und schlummert im Idle Modus benötigt es noch ca. 90mA. Im Einsatz wird das WLAN Modul also die Lebensdauer einer Batterie deutlich reduzieren.

4.3.2 Software

Um die ausgewählte Hardware verwenden zu können sind Funktionen im Programmcode notwendig. Je nach eingesetzter Hardware ist der Programmier- und Rechenaufwand hierfür unterschiedlich hoch.
Um die RS232-Schnittstelle benutzen zu können, ist relativ wenig Aufwand notwendig. Bevor die Schnittstelle verwendet werden kann, müssen einige Parameter festgelegt werden. Dazu gehören u. a. die Baudrate, Start- und Stoppzeichen und die verwendete Codierung der übertragenen Zeichen. Diese Parameter werden üblicherweise eingestellt, bevor die Verbindung aufgebaut wird. Zudem müssen die Einstellungen auf beiden Endpunkten der Verbindung vorgenommen werden.

Für das WLAN-Modul ist ein etwas größerer Aufwand notwendig. Der Verbindungsaufbau zwischen Lantronix Wiport und WLAN Access Point wird automatisch vom WiPort-Modul übernommen. Auch eine eventuelle Verschlüsselung wird vom WiPort übernommen. Der WiPort stellt eine Weboberfläche zur Verfügung, die Zugriff auf alle notwendigen Einstellungen erlaubt. Eine genaue Anleitung ist u. a. auf der c't-Bot Projektseite[1] zu finden.
Die beiden RS232-Schnittstellen des WiPort sind über das Netzwerk erreichbar. Dazu müssen die Netzwerkadresse und der Port der jeweiligen Schnittstelle bekannt sein. Nun könnte man mit den bekannten Netzwerktechniken auf die Daten auf der RS232-Schnittstelle zugreifen und neue Daten übermitteln. Lantronix stellt hierfür aber ein nützliches Werkzeug zur Verfügung. Der *Com Port Redirector*[2] emuliert auf dem lokalen Rechner einen COM Port, der der RS232-Schnittstelle auf dem WiPort entspricht. Damit stellt sich die Verbindung zwischen c't-Bot und Robotic-Studio für beide Endpunkte als normale RS232-Schnittstelle dar.
Der Vorteil dieser Methode ist, dass für das WiPort-Modul kein umständlicher oder spezialisierter Code nötig ist. Das WiPort-Modul kann jederzeit durch eine normale kabelgebundene serielle Verbindung ausgetauscht werden, ohne dass Änderungen am Programmcode vorgenommen werden müssen. Der Nachteil ist aber, dass ein weiteres Programm auf dem lokalen Rechner nötig ist, damit der emulierte COM Port zu verwenden ist.

4.3.3 Protokoll

Damit Daten von dem c't-Bot zum Robotic-Studio und umgekehrt gesendet werden können und diese Daten auch von der Gegenseite verstanden werden, ist ein Protokoll notwendig. Dieses Protokoll muss beiden Seiten bekannt sein und eingehalten werden.
Ein Protokoll muss folgende Informationen enthalten Wikipedia (2008a):

[1] http://www.heise.de/ct/projekte/ct-bot/erweiterung.shtml
Stand 03.12.2008
[2] http://www.lantronix.com/device-networking/utilities-tools/com-port-redirector.html
Stand 03.12.2008

- Datenflusskontrolle (Handshaking)
- Vereinbarung der verschiedenen Verbindungscharakteristiken
- Wie eine Botschaft beginnt und endet
- Wie eine Botschaft formatiert ist
- Was mit beschädigten oder falsch formatierten Botschaften getan wird (Fehlerkorrekturverfahren)
- Wie unerwarteter Verlust der Verbindung festgestellt wird und was dann zu geschehen hat
- Beendigung der Session oder der Verbindung

Um Botschaften richtig empfangen zu können, muss die Gegenstelle wissen, wo eine Botschaft anfängt und wo sie aufhört. Hierfür werden Markierungen für Anfang und Ende benötigt, die eindeutig sind. Im einfachsten Fall ist das ein eindeutiges Signal oder Zeichen.

Für den Inhalt einer Botschaft muss eine feste Struktur geschaffen werden. Informationen, die in jeder Botschaft vorkommen, müssen in eine feste Reihenfolge gebracht werden. Variable Daten müssen in der Botschaft so untergebracht werden, dass sie erkannt werden können und nicht mit dem Rest der Botschaft kollidieren.

Verschiedene Sensoren haben unterschiedliche Methoden, ihre Werte darzustellen. Der Maussensor des c't-Bots z. B. hat eine Auflösung von 18 mal 18 Bildpunkten bei einer Bildpunktauflösung von einem Byte. Die meisten anderen Sensoren, wie Entfernungs- oder Liniensensoren, stellen ihre Werte in einem 16-Bit-Wort dar. Die meisten Sensoren sind in doppelter Ausführung jeweils für die rechte und linke Seite vorhanden. Um die Daten eines Sensorpaares zu übertragen, sind also zwei 16-Bit-Wörter notwendig. Für das Mausbild sind jedoch weit mehr Daten zu übertragen. Wird aber ausreichend Platz für das Mausbild innerhalb einer Botschaft vorgehalten, wird in den meisten Fällen kostbare Zeit für die Übertragung leerer Informationen verschwendet. Hier wird also ein Format für die Botschaft gesucht, das sich in der Länge den Erfordernissen unterschiedlicher Sensoren anpasst.
Für die Botschaft wird also ein Feld benötigt, das anzeigt, wie groß die Daten sind, die übertragen werden, und ein Feld variabler Größe, das die Sensordaten enthält.

Die Natur der Datenübertragung bringt es mit sich, dass nicht alle Daten so empfangen werden, wie sie gesendet wurden. Entweder geht ein Teil der Informationen verloren oder die Daten werden unterwegs verändert.
Diese Fehler sollten mit dem Protokoll zu erkennen sein. Wenn ein Fehler erkannt wurde, sollte es auch Möglichkeiten geben, die Fehler zu beheben. Kann der Fehler nicht behoben

werden, sollte die Botschaft verworfen werden und eine neue, hoffentlich diesmal fehlerfreie Botschaft angefordert werden können.

Der Punkt *Vereinbarung der Verbindungscharakteristika* ist hier vernachlässigbar. Für die Kommunikation zwischen c't-Bot und Robotic-Studio wird eine RS232-Schnittstelle verwendet. Die Schnittstelle muss auf beiden Seiten vor der Verwendung eingestellt werden. Die Werte für die Einstellungen müssen vorher bekannt sein und können später zur Laufzeit nicht mehr geändert werden.

4.4 Arbeitsverteilung auf Robotic-Studio und c't-Bot

Wie können Robotic-Studio und c't-Bot effektiv gemeinsam eingesetzt werden? Diese Frage ist ein zentraler Gedanke, wenn damit begonnen wird, Software für den c't-Bot und das Robotic-Studio zu entwickeln. Der c't-Bot kann durch das Robotic-Studio entlastet werden. Üblicherweise läuft das Robotic-Studio auf einem Rechner, der über ein Vielfaches der Ressourcen des c't-Bots verfügt. Aber nicht alle Aufgaben lassen sich sinnvoll mit dem Robotic-Studio bewältigen. Welche Aufgaben das Robotic-Studio übernimmt und welche Aufgaben auf dem c't-Bot verbleiben lässt sich erst dann entscheiden, wenn die genaue Aufgabenstellung bekannt ist. Generell lassen sich aber drei Varianten herausstellen, wie die Aufgaben untereinander aufgeteilt werden können.

4.4.1 Autonomer c't-Bot

Diese Variante bietet sich an, wenn der Roboter leichte bis mittelschwere Aufgaben bewältigen muss, bzw. Aufgaben, für die die eigenen Ressourcen des c't-Bots ausreichen.
Für eine ganze Reihe von Aufgaben, die für den c't-Bot denkbar sind, ist es nicht nötig, die Berechnung auf einen externen Rechner auszulagern. Grundsätzlich kommt der c't-Bot nicht darum herum einige Aufgaben selber zu erledigen. Das Erfassen der Sensordaten und Stellen der Aktoren ist ausschließlich innerhalb des c't-Bots möglich. Diese Aufgaben beanspruchen aber kaum Ressourcen. Die meiste Zeit wird der Prozessor des Roboters brach liegen oder aber die Sensoren werden in einem unnötig schnellen Intervall abgefragt. Die freien Kapazitäten können für Aufgaben verwendet werden, die keinen großen Aufwand für den Roboter darstellen.
Das Robotic-Studio muss in dieser Variante keine wichtigen Aufgaben übernehmen. Denkbar wäre z. B., dass das Robotic-Studio lediglich die Sensordaten vom c't-Bot empfängt und die Sensordaten übersichtlich für den Benutzer darstellt.
Da das Robotic-Studio keine zeitkritischen Berechnungen ausführen muss, ist die Geschwindigkeit für diese Variante nicht ausschlaggebend. Es muss aber darauf geachtet werden,

dass der c't-Bot seine Sensordaten schnell genug versendet bekommt, sodass er noch genügend Zeit übrig behält, um seine eigentlichen Aufgaben zu bewältigen.

Es stellt sich aber die Frage, ob es für diese Aufgabe nötig ist, die Komplexität des Robotic-Studio in Kauf zu nehmen, nur um einfache Daten auf dem Bildschirm darzustellen. Hierfür reicht eine normale Programmierumgebung aus. Zudem würde die Idee, den Roboter zu entlasten, außer Acht gelassen.

4.4.2 Fernsteuerung durch Robotic-Studio

Ein radikaler Ansatz ist, das Robotic-Studio die komplette Arbeit übernehmen zu lassen und den Einsatz des c't-Bots darauf zu beschränken die Sensordaten zu erfassen und die Aktoren zu stellen. Hierdurch würde das komplette Verhalten des Roboters ferngesteuert. Da das Robotic-Studio auf einem PC läuft und dieser kaum Beschränkungen seiner Ressourcen unterliegt, kann davon ausgegangen werden, dass das Robotic-Studio alle denkbaren Aufgaben erledigen kann.
Diese Variante birgt aber auch Gefahren. Sollte der Roboter z. B. auf einen Abgrund zusteuern und die Datenleitung zum steuernden Rechner überlastet oder sogar unterbrochen sein, wird der Roboter unweigerlich Schaden nehmen.

Damit das Robotic-Studio schnell genug auf Hindernisse oder andere Störimpulse reagieren kann, sind drei Faktoren von entscheidender Bedeutung: Die *Abtastrate* der Sensoren, die *Geschwindigkeit* der Datenübertragung und die *Verzögerung*, die durch die Länge der Datenstrecke entsteht. Die Sensorwerte müssen in kurzen Abständen aktualisiert werden. Sind die Abstände zwischen den Sensorwerten zu hoch, wird ein Hindernis evtl. zu spät erkannt. Sind die Sensordaten ermittelt, müssen sie so schnell wie möglich vom c't-Bot an das Robotic-Studio gesendet werden.
Eine hohe Datenrate nützt aber nichts, wenn die Daten zu lange auf den Leitungen unterwegs sind, bis sie im Robotic-Studio angekommen sind.

Soll der c't-Bot z. B. eine Linie auf dem Boden verfolgen, muss er in kurzen Abständen neue Werte für seine Liniensensoren ermitteln und an das Robotic-Studio senden.
Der c't-Bot kann Geschwindigkeiten bis knapp über 0,3 m/s erreichen. Eine Linie wird er aber mit der Maximalgeschwindigkeit auch unter den besten Bedingungen nicht verfolgen können. Die Erfahrung zeigt, dass sich eine Geschwindigkeit von 0,1 m/s sehr gut zum Linienverfolgen eignet. Damit schnell erkannt werden kann, wann der c't-Bot die Linie verlässt, sollte spätestens nach 1 mm ein neuer Sensorwert gemessen werden. Daraus folgt die Frage, nach welcher Zeit ein neuer Sensorwert bereitstehen muss.

$$\frac{1\ s}{100\ mm} = 0,01\ \frac{s}{mm} = 10\ \frac{ms}{mm} \tag{4.1}$$

Aus der oben stehenden Gleichung wird ersichtlich, dass der c't-Bot alle 10 ms einen neuen Sensorwert ermittelt muss, um bei einer Geschwindigkeit von 0,1 m/s jeden Millimeter erfassen zu können. Anders ausgedrückt, der c't-Bot muss 100 Messungen pro Sekunde vornehmen.

Innerhalb dieser 10 ms muss nun der Sensorwert an das Robotic-Studio gesendet werden, um dort neue Werte für die Motoren zu berechnen. Diese Motorenwerte müssen an den c't-Bot gesendet und umgesetzt werden. D. h., dass die Geschwindigkeit und eine geringe Verzögerung der Datenübertragung erheblich zum Erfolg beitragen.

Angenommen, die Datenleitung verursacht eine Verzögerung von 2 ms in jede Senderichtung und die Verarbeitung der Daten innerhalb des Robotic-Studio dauert 1 ms, bleiben für die eigentliche Datenübertragung nur 5 ms, bzw. 2,5 ms pro Richtung, übrig. Wenn ein Datenpaket 10 Byte groß ist, müssen die Daten mit einer Geschwindigkeit von

$$\frac{10\,Byte}{2,5\,ms} = \frac{4\,Byte}{1\,ms} = 4000\,\frac{kByte}{s} \tag{4.2}$$

versendet werden.

4.4.3 Kooperation von c't-Bot und Robotic-Studio

In den oberen beiden Abschnitten ist ein großes Problem ersichtlich, nämlich die Datenverbindung zwischen c't-Bot und Robotic-Studio. Die serielle Verbindung zwischen c't-Bot und Robotic-Studio kann nur eine begrenzte Menge an Daten befördern. Die Sensordaten, die der c't-Bot ermittelt brauchen relativ lange, bis sie im Robotic-Studio angekommen sind. Umgekehrt müssen berechnete Ergebnisse vom Robotic-Studio erst an den c't-Bot gesendet werden, bevor der c't-Bot sie umsetzen kann. Zudem ist nicht sichergestellt, dass die Daten überhaupt auf der Gegenseite ankommen.
Daraus kann eine dritte Variante abgeleitet werden. Dem Roboter werden die Aufgaben übertragen, die eine schnelle Reaktion erfordern, bzw. bei einer Störung der Kommunikation zum Robotic-Studio Schaden abwenden können. Zu diesen Aufgaben gehören z. B. eine Kollisionsvermeidung mit Wänden oder anderen Objekten und das Erkennen von Abgründen.
Das Robotic-Studio übernimmt die Aufgaben, die nicht unbedingt das Überleben des c't-Bots sichern. Der c't-Bot befindet sich somit immer in einem sicheren Zustand und verfügt über genügend Sicherheitsreserven, wenn eine gefährliche Situation auftritt.

Diese Variante kann auch dazu verwendet werden, die einzelnen Sensoren des c't-Bots zu einem hybriden Sensor zusammenzufassen. Durch die sogenannte Sensor-Fusion kann aus einzelnen unpräzisen Sensoren ein Wert berechnet werden, der genauer ist, als der einzelne

Sensorwert. Das Rauschen eines einzelnen Sensors wird durch die Kombination mit anderen Sensoren reduziert.
Innerhalb des c't-Bots können die einzelnen Sensorwerte vorverarbeitet werden, um dem Robotic-Studio eine genauere Aussage über den momentanen Zustand anbieten zu können.

Ein weiterer Vorteil der Kooperation von c't-Bot und Robotic-Studio besteht darin, dass die Subsumption-Architektur des c't-Bots auf das Robotic-Studio ausgeweitet werden kann.
Die einzelnen Befehle, die das Robotic-Studio sendet, können in unterschiedlichen Verhalten bearbeitet werden. Diesen einzelnen Verhalten können unterschiedliche Prioritäten zugewiesen werden.

Da zeitkritische Aufgaben vom c't-Bot übernommen werden, ist die Geschwindigkeit der Datenverbindung für diese Variante nicht so sehr ausschlaggebend wie in 4.4.2. Trotzdem darf die geschwindigkeit der Datenübertragung nicht zu niedrig sein. Wenn der c't-Bot viel Zeit dafür aufwenden muss, die Daten an das Robotic-Studio zu senden, bleibt evtl. nicht genug Zeit für die eigenen Aufgaben übrig. Zudem darf die Verzögerung der Daten nicht zu groß sein. Brauchen die Daten zu lange, um zum Robotic-Studio zu gelangen, sind diese evtl. bereits veraltet und unbrauchbar.
Wie kritisch die Geschwindigkeit und Verzögerung der Datenübertragung für die jeweiligen Aufgaben ist, muss von Fall zu Fall entschieden werden.

5 Design

5.1 Eingesetzte Mittel

5.1.1 Robotic-Studio

Der Grund, warum im Rahmen dieser Bachelorarbeit das Robotic-Studio verwendet wird, ist ziemlich einfach. Es gibt kaum eine Alternative, die es mit dem Umfang und den Leistungen des Robotic-Studio aufnehmen kann.

Es existieren zwar viele Roboterplattformen für den Heimgebrauch, aber kein Framework, das in der Lage ist, die unterschiedlichen Roboter zu integrieren. Jede Roboterplattform enthält auch ein eigenes Framework zur Programmierung. Der Umfang und die Leistungsfähigkeit variiert jedoch stark. Zudem sind die Plattformen meist eine spezialisierte Lösung für den jeweiligen Roboter und lassen sich nur mit Aufwand für andere Projekte einsetzen.
Die wenigen Entwicklungsumgebungen, die auf dem Markt erhältlich sind, befinden sich oft noch im Anfangsstadium und bieten nur einen beschränkten Leistungsumfang.

Das Programm *LabView* der Firma *National Instruments* ist eine Programmierumgebung, die am ehesten mit dem Robotic-Studio zu vergleichen ist. In LabView werden Programme als Datenflüsse, vergleichbar mit der VPL, modelliert. Grundsätzlich wird das eigentliche Programm als Blockschaltbild entwickelt. Darauf aufgebaut werden Anzeigen und Kontrollelemente, mit denen der Benutzer mit dem Programm interagieren kann.
Für die Lego-NXT-Roboter ist eine Entwicklungsumgebung erhältlich, die auf LabView basiert. Mit ihr sind die unterschiedlichen Hardwarekomponenten der NXT-Roboter steuer- und programmierbar.

Im November 2008 hat Microsoft eine neue Version der Entwicklungsumgebung unter dem Namen *Microsoft Robotics Developer Studio 2008* veröffentlicht. Diese Version ist in 3 verschiedenen Versionen erhältlich. Der volle Funktionsumfang ist in den Standard- und Academic-Versionen erhältlich. Die Standardversion kostet allerdings 999,90 US-Dollar und die Academic-Version ist nur über das MSDNAA-Netzwerk verfügbar. Die dritte Variante ist die kostenlose Express-Edition. Die grundlegenden Funktionen wie CCR und DSS sind auch hier enthalten, allerdings sind einige nützliche Funktionen nicht verfügbar. Dazu zählen unter

anderem die Unterstützung des Kompakt-Frameworks für PDAs usw. und die automatische C# Codegenerierung. Bei der Express-Edition ist die Weitergabe der CCR und DSS Laufzeitumgebung untersagt und richtet sich somit nur an Hobby-Programmierer.

5.1.2 c't-Bot

In der letzten Zeit hat die Verfügbarkeit von Roboterplattformen für den Hobbybereich stark zugenommen. Die bekanntesten unter ihnen sind die Lego-NXT-Roboter und der Staubsaugerroboter Roomba der Firma iRobot. Beide sind bereits im Robotic-Studio integriert und durch Tutorien beschrieben.
Der c't-Bot bietet in seiner Preisklasse ein gutes Preis-Leistungs-Verhältnis. Bei der Entwicklung des c't-Bots wurde viel Wert auf eine hohe Einsteigerfreundlichkeit gelegt. Der Roboter ist als Bauteilesatz erhältlich und ist mit geringen handwerklichen Kenntnissen zusammenzubauen. Defekte lassen sich ohne großen Aufwand beheben.
Dadurch, dass der c't-Bot vollständig dokumentiert ist und unter der GPL[1] veröffentlicht wurde, sind eigene Veränderungen und Erweiterungen für den c't-Bot ohne großen Aufwand möglich.

5.1.3 Visual Studio 2008

Visual Studio bietet sich als Entwicklungsplattform für das Robotic-Studio an. Die zahlreichen Beispiele und Tutorien, die im Robotic-Studio enthalten sind, enthalten eine fertige Projektdatei für Visual Studio und können somit leicht verwendet werden. Außerdem unterstützt Visual Studio das Erstellen neuer Services. Die verschiedenen Parameter, die bei der Verwendung der Kommandozeile nötig sind, müssen nicht mühsam nachgeschlagen werden, sondern können bequem über eine Benutzeroberfläche eingegeben werden.
Microsoft bietet eine kostenlose Version unter dem Namen Visual Studio 2008 Express an. Diese Version bietet nicht den vollen Funktionsumfang, reicht aber für die Entwicklung mit dem Robotic-Studio völlig aus.
Wer Visual Studio nicht mag, kann auch auf andere Entwicklungsumgebungen zurückgreifen. Es existieren eine Reihe Alternativen, sowohl aus dem kommerziellen als auch aus dem Open-Source-Bereich.

[1] GNU General Public License
http://www.gnu.de/documents/gpl-2.0.de.html

5.1.4 AvrStudio und AVRISP mkII

Die Programme für den c't-Bot werden nicht auf dem c't-Bot selber entwickelt, sondern auf einem PC mit einer entsprechenden Entwicklungsumgebung. Der Programmcode wird von der Entwicklungsumgebung in Maschinencode übersetzt. Dieser Maschinencode wird dann mit einem Programmieradapter auf den Mikrocontroller auf dem c't-Bot übertragen.
Leider sind im c't-Bot die vorhandenen Anschlüsse für die Programmierung mittels JTAG-Schnittstelle bereits belegt. Die Programmierung des Roboters ist nur über eine SPI-Schnittstelle möglich. Der Nachteil hierbei ist, dass es keine Möglichkeit gibt, den Roboter aus der Entwicklungsumgebung heraus zu debuggen. Auf der anderen Seite ist für die Programmierung über die SPI-Schnittstelle nur ein Programmieradapter notwendig, der mit ein paar Drähten, Widerständen und Dioden leicht selber herzustellen ist.
Im Roboter-Labor ist eine ausreichende Anzahl an AVRISP mkII Programmierandapter vorhanden, aus diesem Grund wird dieser Progammieradapter auch im Rahmen dieser Bachelorarbeit verwendet. Zusammen mit AvrStudio ist eine komfortable Programmierung des c't-Bots möglich. Zudem können die Fuse-Bits des c't-Bots über eine übersichtliche Benutzeroberfläche eingestellt werden.
Sowohl das AvrStudio als auch der AVRISP mkII sind problemlos durch andere Werkzeuge zu ersetzen. Als Alternative zu dem Programmieradapter AVRISP mkII kann auch ein selbst gebauter Programmieradapter verwendet werden. Hierfür ist ein Bausatz im Rahmen des c't-Bot Projekt erhältlich.
Anstatt des AvrStudio kann für die Programmierung des c't-Bots auch ein anderes Programm verwendet werden. In der Robotik-Szene ist das Programm PonyProg[2] weit verbreitet. PonyProg bietet eine große Palette Einstellmöglichkeiten an und ist für diverse Betriebssysteme erhältlich.

5.1.5 Eclipse

Die Entscheidung, Eclipse als Entwicklungsumgebung für den c't-Bot zu verwenden, ist zum großen Teil durch die Entwickler des c't-Bots vorgegeben. Für das komplette Framework des c't-Bots existiert eine fertige Lösung für Eclipse.
Bevor aber mit der Programmierung angefangen werden kann sind ein paar Vorkehrungen notwendig. Die Eclipse-Plattform benötigt ein paar Erweiterungen und Einstellungen. Eine genaue Anleitung [3] kann auf der Projektseite des c't-Bots im Internet gefunden werden. Generell ist es aber auch möglich, den Code für den c't-Bot in der Entwicklungsumgebung AvrStudio zu entwickeln. Um den Code zu compilieren kann in den Projekteinstellungen das

[2] http://www.lancos.com/prog.html
[3] http://www.heise.de/ct/projekte/machmit/ctbot/wiki/Installationsanleitung
Stand 29.10.2008

Makefile des c't-Bots verwendet werden. Für die Größe des c't-Bot Frameworks bietet sich aber eher die Eclipse-Entwicklungsumgebung an.

5.1.6 RS232 und WLAN

Trotz ihres „biblischen“ Alters ist die RS232-Schnittstelle immer noch sehr beliebt und verbreitet. Das liegt unter anderem daran, dass die Anforderungen dieser Schnittstelle gering sind. An modernen Computern ist meistens noch ein serieller Anschluss zu finden. Anschlusskabel sind entweder bereits vorhanden oder für geringes Geld in jedem Laden mit Computerzubehör erhältlich. Der Programmieraufwand für diese Schnittstelle ist sehr gering. In den meisten Softwarebibliotheken sind Funktionen enthalten, die die Handhabung mit der Schnittstelle vereinfachen. Sind keine Funktionen vorhanden, können sie ohne großen Aufwand selber geschrieben werden.

WLAN hat in den letzten Jahren eine weite Verbreitung erfahren. In vielen Haushalten ist ein Internetrouter mit integriertem WLAN-Zugangspunkt vorhanden. Moderne Desktop-Computer und Laptops sind oft bereits mit einer WLAN-Karte ausgestattet. Fehlende Hardware kann für einige Euro nachgekauft werden. Zudem unterstützt die Anbindung des c't-Bots an das Robotic-Studio über WLAN die generelle Verteilung der Services auf unterschiedliche Rechner.

Das Für und Wider der RS232-Schnittstelle und des Erweiterungsmoduls des c't-Bot Projektes werden in den Kapiteln 4.3.1 und 5.2.1 genauer besprochen.

5.2 Kommunikation

5.2.1 Hardware

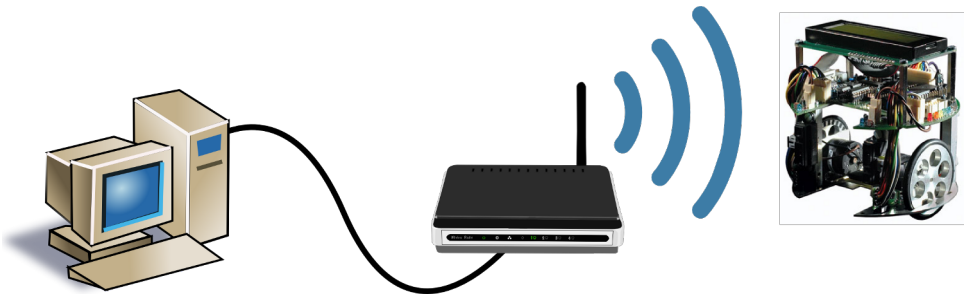


Abbildung 5.1: Verbindung zwischen PC und c't-Bot

Die Frage, welche Hardware zum Einsatz kommt ist größtenteils durch das c't-Bot Projekt vorgegeben. Der c't-Bot ist ein abgeschlossenes Projekt, an dem Änderungen an der Hardware kaum möglich, aber auch kaum notwendig sind.

Im Rahmen dieser Bachelor-Arbeit wird neben dem c't-Bot auch noch dessen Erweiterung, das WLAN-Modul, verwendet. Dieses Modul benötigen wir, um die Kommunikation zwischen PC und c't-Bot per Funk abwickeln zu können. Generell ist das Funkmodul gegen jede andere drahtlose Verbindung austauschbar. Es muss nur darauf geachtet werden, dass einerseits die Geschwindigkeit für den Datenaustausch ausreicht und dass andererseits der Zugriff über einen COM-Port erfolgt. Ansonsten müssen Änderungen an der Software vorgenommen werden.

Das Erweiterungsmodul für den c't-Bot verfügt über ein *WiPort*-Modul der Firma Lantronix. Das Modul stellt im Grunde einen Tunnel für die RS232-Schnittstelle durch ein TCP/IP-Netzwerk dar. Am WiPort-Modul können zwei RS232-Schnittstellen mit jeweils einer Empfangs- und Sendeleitung angeschlossen werden. Diese sind dann über das Netzwerk über die IP-Adresse des Moduls und über die Portnummer für die jeweilige Schnittstelle erreichbar.
Die Firma Lantronix stellt die Software *Com Port Redirector* kostenlos zur Verfügung. Mit dieser Software lässt sich der im Netzwerk verfügbare RS232-Port als COM-Port emulieren. Dadurch muss die Software im Robotic-Studio keine direkte Verbindung zu dem WiPort über das Netzwerk aufbauen, sondern kann den c't-Bot über eine normale serielle Schnittstelle ansprechen.

Für Debugging-Zwecke ist der USB-2-Bot-Adapter nützlich. Dieser bildet eine Verbindung zwischen der RS232-Schnittstelle des c't-Bots und dem PC per USB. Sollte das WLAN-Erweiterungsmodul oder das Netzwerk ausfallen, bietet der Adapter eine günstige und effi-

ziente Möglichkeit, die Kommunikation aufrechtzuerhalten. Zudem werden mögliche Fehlerquellen im Netzwerk umgangen.

Als weitere Hardware ist noch ein bestehendes Netzwerk mit WLAN notwendig sowie ein einigermaßen moderner Rechner mit mindestens Windows XP als Betriebssystem.

5.2.2 Protokoll

In der Softwarebibliothek des c't-Bots befindet sich bereits ein Protokoll für die Kommunikation zwischen c't-Bot und c't-Sim. Das Protokoll eignet sich auch für den Einsatz des c't-Bots im Robotic-Studio.

```
typedef struct {
        uint8 startCode;
        request_t request;
        uint8 payload;
        int16 data_l;
        int16 data_r;
        int16 seq;
        uint8 CRC;
} command_t;

typedef struct {
        uint8 command:8;
        uint8 subcommand:7;
        uint8 direction:1;
} request_t;
```

Listing 5.1: Kommando-Struktur

Die Felder **uint8** startCode und **uint8** CRC geben den Anfang und das Ende eines c't-Bot Kommandos an. Standardmäßig werden hierfür die Zeichen '>' für den Anfang bzw. '<' für das Ende eines Kommandos verwendet. Bei dieser Art der Kennzeichnung von Anfang und Ende eines Kommandos droht aber ein Fallstrick. Die Werte der Sensoren können zufällig den ASCII-Werten der Zeichen entsprechen. Dadurch kann es vorkommen, dass Anfang und Ende des Kommandos falsch erkannt werden.
request_t request ist eine eigene Struktur. In dieser Struktur ist mit dem Feld command festgehalten, um welche Informationen es sich bei den mitgelieferten Daten handelt, und mit den Feldern subcommand und direction, um was für eine Art des Kommandos es sich handelt.
Das Feld payload gibt an, ob der Struktur **command_t** weitere Daten folgen. Die Payload ist

nicht Bestandteil der Protokoll-Struktur, sondern werden als eigenständige Daten hintenangestellt.
In den Feldern **int16** data_l und **int16** data_l stehen die eigentlichen Nutzdaten eines Kommandos. Da die meisten Sensoren und der Antrieb des c't-Bots jeweils auf der rechten und auf der linken Seite vorhanden sind, ist es sinnvoll, beide Werte als Paar zusammenzufassen und gemeinsam zu versenden.
Mit dem Feld **int16** seq werden die versendeten Kommandos fortlaufend nummeriert. Der Empfänger kann dann die gültig empfangenen Pakete mitzählen und mit der Sequenznummer vergleichen. Dadurch lässt sich erkennen, wie viele Pakete entweder verloren gegangen sind oder defekt angekommen angekommen sind und deshalb verworfen wurden.

Möchte man z. B. ein Mausbild übertragen, passen die Daten des Mausbildes nicht in die data_l- und data_r-Felder. Deshalb werden die Daten des Mausbildes erst nach dem Kommando versendet. Dazu wird in dem Kommando im Feld payload die Anzahl der noch folgenden Daten eingetragen. Das empfangene Programm muss nun das payload-Feld auswerten und die nachfolgenden Daten des Mausbildes zusätzlich zum Kommando empfangen und abspeichern.

Die Struktur eignet sich auch, um Befehle an den Roboter zu senden.
Da die Struktur command_ t alle Bedürfnisse nach einer einheitlichen Kommunikation erfüllt und bereits in der Softwarebibliothek des c't-Bots enthalten ist, wird die Struktur auch im Robotic-Studio eingesetzt.

Durch die Definition dieser Struktur können ein paar grundsätzliche Annahmen für die Kommunikation zwischen c't-Bot und dem Robotic-Studio getroffen werden: Ein Kommando wird durch das Start- und Endzeichen eindeutig identifiziert. Durch die festgelegte Struktur ist jedes Kommando immer genau 11 Byte lang. Weicht das empfangene Kommando davon ab, kann davon ausgegangen werden, dass das Kommando ungültige Daten enthält und verworfen werden kann.
Durch das Feld **request_t** request sind die enthaltenen Daten eines Kommandos immer einem bestimmten Sensor- bzw. Aktorentyp zuzuordnen.

Von den Anforderungen, die in Kapitel 4.3.3 aufgezählten sind, kann dieses Protokoll aber nicht alle erfüllen. Es können zwar defekte Kommandos erkannt werden, aber eine Korrektur von defekten Kommandos ist nicht möglich.
Fällt das Robotic-Studio oder der c't-Bot aus, kann keiner von beiden das erkennen. Das Robotic-Studio könnte Vermutungen anstellen, wenn das regelmäßige Eintreffen neuer Sensordaten ausbleibt. Aber auch dann kann das Robotic-Studio nicht erkennen, ob der c't-Bot ausgefallen ist oder ob die Verbindung unterbrochen wurde.

5.3 Verhalten im c't-Bot

Der Programmieraufwand für den c't-Bot ist ziemlich gering. Durch das umfangreiche Framework des c't-Bots ist das grundlegende Programm bereits fertig. Um den c't-Bot für die eigenen Zwecke verwenden zu können müssen nur eigene Verhalten geschrieben werden, die in den vorhandenen Programmcode integriert werden.

5.3.1 Vorhandene Verhalten

Das Framework enthält bereits eine Reihe an nützlichen Verhalten. Interessant für den Einsatz des c't-Bots mit dem Robotic-Studio sind die Verhalten bot_avoid_border_behaviour und bot_avoid_col_behaviour. Diese Verhalten haben beide eine sehr hohe Priorität. Das ist sinnvoll, da sie potentiellen Schaden abwehren können.
Mit dem Verhalten bot_avoid_border_behaviour erkennt der c't-Bot Abgründe und verhindert einen Absturz. Dazu wertet das Verhalten die beiden Abgrundsensoren aus. Überschreiten einer oder beide Sensoren einen bestimmten Wert, fährt der c't-Bot rückwärts. Zusätzlich kann eine vorher registrierte Notfallroutine ausgeführt werden.
Das Verhalten bot_avoid_col_behaviour erkennt Wände und Gegenstände, die sich vor dem c't-Bot befinden und weicht diesen Hindernissen aus. Der c't-Bot wertet beide Distanzsensoren aus. Der Raum vor ihm wird dabei in mehrere Zonen unterteilt. In der dichtesten Zone, der Kollisionszone, dreht sich der c't-Bot zu der Seite, auf der der meiste Platz vorhanden ist. In den weiter entfernten Zonen bremst der c't-Bot den Motor auf der gegenüber liegenden Seite des Sensors, der ein Hindernis erkannt hat.

Der c't-Bot verfügt noch über eine Reihe weiterer Verhalten für unterschiedliche Situationen und Aufgaben. Alle Verhalten können beliebig miteinander kombiniert werden. Ob das immer sinnvoll ist, muss aber in der jeweiligen Situation entschieden werden.

5.3.2 Robotic-Studio-Verhalten

Der Aufwand, einen eigenen Service zu entwickeln, hält sich in Grenzen. Damit das eigene Verhalten überhaupt im c't-Bot verwendet werden kann, sind ein paar Vorbereitungen notwendig.
Zuerst müssen zwei Dateien angelegt werden: Eine Datei für den Quellcode des Verhaltens und eine Headerdatei, in der der Funktionskopf des Verhaltens eingetragen wird. Die Headerdatei wird in der Datei available_behaviours.h eingetragen.
Die Namenskonvention für die Dateinamen ist behaviour_VERHALTENS_NAME. Für die Funktionsnamen gilt das Schema bot_VERHALTENS_NAME_behaviour.

Damit die Sensordaten vom c't-Bot an das Robotic-Studio gesendet werden und die Befehle vom Robotic-Studio empfangen werden können, muss in der Datei ct−bot.h das Makro #define BOT_2_PC_AVAILABLE einkommentiert werden. Dadurch wird zum einen die RS232-Schnittstelle initialisiert, zum anderen werden nun automatisch die aktuellen Sensordaten an das Robotic-Studio gesendet. Weiterhin empfängt und speichert der c't-Bot Daten. Kommandos erkennt der c't-Bot automatisch und speichert sie in einem global zugänglichen Puffer ab.

Der c't-Bot kann bereits eine Menge an Befehlen in den empfangenen Kommandos erkennen und ausführen. So kann er auf Befehl ein aktuelles Bild des Maussensors an den PC senden. Die meisten der Befehle, die der c't-Bot empfängt wertet er automatisch über die Funktion *evaluate* aus. Für die Auswertung der Motorbefehle, die vom Robotic-Studio gesendet werden, ist aber ein zusätzliches Verhalten notwendig.

```
void bot_MRDS_behaviour(Behaviour_t * data){
  if(received_command.request.command == CMD_AKT_MOT){

    speedWishLeft  = received_command.data_l * 4;
    speedWishRight = received_command.data_r * 4;
  }
}
```

Listing 5.2: Robotic-Studio-Verhalten

Das Verhalten überprüft, ob das zuletzt empfangene und im Puffer gespeicherte Kommando ein CMD_AKT_MOT-Kommando ist. Ist das der Fall, werden die neuen Werte für die Motoren berechnet. Die Motorenwerte, die das Robotic-Studio sendet, sind Prozentangaben im Bereich -100% bis +100%. Der c't-Bot erwartet aber absolute Werte. Diese Werte liegen im Bereich -450 bis +450.
Die empfangenen Werte müssen also noch umgerechnet werden. Der Mikrocontroller des c't-Bots verfügt über keine leistungsfähige arithmetische Recheneinheit. Additionen und Subtraktionen kann der Mikrocontroller noch einfach und schnell berechnen. Eine Multiplikation oder Division wird aber in mehrere einzelne Rechenschritte aufgeteilt. Zudem stellt eine Berechnung mit Gleitkommazahlen eine zusätzliche Belastung dar.
Für eine korrekte Umrechnung der Werte müsste der übertragene Wert mit dem Faktor 4,5 multipliziert werden. Da die Motoren nicht sehr präzise sind und für dieses Beispiel die volle Geschwindigkeit des c't-Bots nicht notwendig ist, wird der Faktor 4 verwendet. Das erspart dem Mikrocontroller unnötige Rechenarbeit.

5.4 Services

Ausgehend von den Überlegungen in Kapitel 4 muss jetzt entschieden werden, welche Services benötigt werden.
Als wichtigste Komponente ist der Datenaustausch zwischen c't-Bot und Robotic-Studio anzusehen. Hierfür wird ein Service benötigt, der die Verbindung zum c't-Bot aufbaut, die Daten empfängt und Befehle an den c't-Bot sendet. Dieser Service analysiert die empfangenen Nachrichten und benachrichtigt die bei ihm eingetragenen Services über neue Werte. Ein Service braucht sich bei diesem Service nur für die Sensoren einzutragen, die für ihn von Interesse sind, und wird auch nur über Aktualisierungen dieser Sensoren informiert.
Weitere Services können auf diesem Service aufbauen. Um die benötigten Sensordaten zu erhalten, schreibt sich der Service bei dem Kommunikations-Service mit den jeweiligen Sensoren ein. Befehle für den c't-Bot können mit einer Nachricht an den Kommunikations-Service ausgesprochen werden.

5.4.1 CtBotHelper

Das Package *CtBotHelper* ist kein Service, sondern eine Bibliothek von Klassen und Methoden, die von den Services gemeinsam verwendet werden. Damit diese Klassen und Funktionen nicht an einen bestimmten Service gebunden sind, sind sie als eigenständiges Paket implementiert.

CtBotCommand Die Klasse *CtBotCommand* implementiert die Kommando-Struktur, die der c't-Bot für die Kommunikation verwendet (s. Kapitel 5.2.2). Zusätzlich ist noch ein Puffer für Daten integriert, die dem Kommando als Payload angehängt worden sind, z. B. ein Maussensorbild. Zum Erstellen eines CtBotCommand stehen diverse Konstruktoren zur Verfügung. Einem Konstruktor kann die empfangene Zeichenkette übergeben werden. Daraus wird dann ein Kommando erstellt.
Für eine übersichtliche Darstellung auf der Konsole verfügt die Klasse über eine commandToString-Methode, die die Werte des Kommandos übersichtlich ausgibt.

CtBotCommandCode Die Klasse CtBotCommandCode stellt sämtliche Kommando-Codes aus der command.h-Datei des c't-Bots Frameworks zur Verfügung. Jedes Kommando ist als eigenständige *public*-Konstante implementiert, die mit einem entsprechendem Wert initialisiert ist. Das Feld für das Startzeichen eines Kommandos sieht z. B. so aus:

```
public const char CMD_STARTCODE = '>';
```

CtBotSensor Diese Klasse stellt einen Sensor des c't-Bots dar. Er besitzt jeweils ein Feld für den linken und den rechten Sensor. Werden beim Erstellen eines neuen Sensors keine Werte übergeben wird, für den rechten Sensor der Wert 888 und für den linken Sensor der Wert 999 vergeben.
Als Erweiterung für die Klasse CtBotSensor ist die Klasse *CtBotMouseSensor* vorgesehen. Die CtBotMouseSensor-Klasse enthält zusätzlich zu den beiden Sensorenfeldern noch ein Feld für eine Zeichenkette, in der das Mausbild abgespeichert werden kann.

CtBotSensorType Diese Klasse enthält eine Reihe von konstanten Strings nach dem Muster **public** const string BORDER = "BorderSensor";. Diese Strings werden u. a. für das *Selective-Subscribe* verwendet.

5.4.2 CtBotSerial-Service

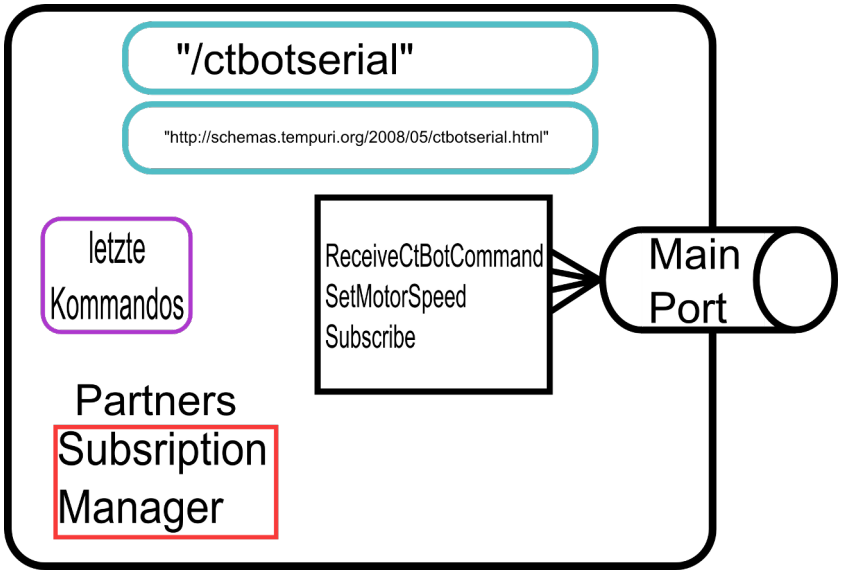


Abbildung 5.2: Vereinfachtes Service-Modell

Dieser Service ist der wichtigste. Er sichert den Datenaustausch zwischen c't-Bot und Robotic-Studio. Zum einen empfängt er die aktuellen Sensordaten, die vom c't-Bot kommen und speichert sie in seinem Service-State. Zum anderen sendet der Service Befehle an den c't-Bot, die von anderen Services kommen.
Somit handelt es sich beim CtBotSerial-Service um einen Vermittler zwischen dem c't-Bot und dem Robotic-Studio.

Für die Kommunikation mittels RS232-Schnittstelle wird die Klasse SerialComm verwendet. Die Klasse initialisiert die RS232-Schnittstelle und stellt Methoden zur Verfügung, mit denen Daten gesendet werden und empfangene Daten weiterverarbeitet werden können. Dazu wird die SerialPort-Klasse aus dem .NET Framework verwendet.
Für das Empfangen der Daten wird ein eigener Thread gestartet, der ausschließlich den Empfangspuffer der RS232-Schnittstelle darauf überprüft, ob ein Kommando vollständig empfangen wurde. Die RS232-Schnittstelle wurde so eingerichtet, dass als NewLine-Zeichen das *'<'*-Zeichen erwartet wird. Durch die Anweisung serialPort.ReadLine() wird eine Zeichenkette zurückgegeben, die von dem angegebenen NewLine-Zeichen terminiert wird. Somit kann ein Kommando mit wenig Aufwand empfangen werden.
Ist ein Kommando empfangen worden, werden die empfangenen Zeichen in ein c't-Bot Kommando umgewandelt und in einer Nachricht an den Main-Port des CtBotSerial-Services gesendet. In dem angeschlossenen Handler werden die eingeschriebenen Services über das neue Kommando informiert.

Die Operationen, die dieser Service bereitstellt sind:

```
[ServicePort]
public class CtBotSerialOperations
    : PortSet<DsspDefaultLookup,
              DsspDefaultDrop,
              Get,
              HttpGet,
              HttpPost,
              ReceiveCtBotCommand,
              SetMotorSpeed,
              SelectiveSubscribe,
              Subscribe,
              Notify>
{}
```

Listing 5.3: CtBotSerial-Service-Operation-Port

Für jede Operation muss ein eigener Handler implementiert werden, wenn die entsprechende Nachricht nicht verloren gehen soll. Die einzigen Ausnahmen bilden die Operationen *DsspDefaultLookup* und *DsspDefaultDrop*. Für diese Operationen werden von der DSS-Laufzeitumgebung automatisch Handler registriert.
Zusammen mit der Operation *Get* bilden die drei Operationen das Minimum an Operationen, die ein Service anbieten muss.

Die Operationen HttpGet und HttpPost werden im Kapitel 5.4.3 näher beschrieben.

Mit der Operation *ReceiveCtBotCommand* empfängt der CtBotSerial-Service Kommandos, die vom c't-Bot gesendet wurden. Wie bereits weiter oben beschrieben bedient sich der Service dazu der SerialComm-Klasse. Der Handler für die ReceiveCtBotCommand-Operation analysiert den Kommandotypen und informiert eingeschriebene Services über das neu empfangene Kommando.

Interessant sind die Operationen *Subscribe* und *SelectiveSubscribe*. Die Unterschiede sind in Kapitel 3.2.7 genauer beschrieben. Während sich ein Service per *Subscription* über Änderungen an allen Sensortypen benachrichtigen lassen kann, kann er über *SelectiveSubscription* die Sensoren angeben, die ihn interessieren. Die Liste der unterstützen Sensoren und deren Namen sind in der CtBotHelper-Klasse CtBotSensorTypes definiert.
Die Reihenfolge in der die Operationen aufgelistet sind ist sehr wichtig. Die Operation SelectiveSubscribe ist von der Operation Subscribe abgeleitet. Wäre erst die allgemeine Operation Subscribe aufgeführt, würden Nachrichten für die spezialisierte Operation SelectiveSubscribe nicht an den entsprechenden Handler weitergeleitet, sondern an den Handler für Subscribe.

Die Operation *SetMotorSpeed* ermöglicht es, Befehle an den Roboter zu senden, um die Motoren auf die übermittelten Werte zu stellen. Die Werte für den linken und rechten Motor müssen im Bereich zwischen -100 und +100 liegen. Sie geben die relative Geschwindigkeit in Prozent an. Werte, die den Wertebereich von -100 bis +100 überschreiten, werden automatisch auf die maximalen Werte -100 oder +100 gesetzt. Der Wert +123 wird z. B. auf den Wert +100 gesetzt.

Die Operation *Notify* wird dazu verwendet, eingeschriebene Services über eingetroffene Sensorwerte zu informieren. Jeder eingeschriebene Service muss diese Operation in seinen Service-Port übernehmen und einen Service-Handler implementieren. Ansonsten kann er nicht über neue Sensordaten informiert werden. Die Notify-Nachricht enthält den Sensortyp, einen linken und einen rechten Sensorwert und evtl. eine Payload. Somit braucht der empfangende Service bei einer Benachrichtigung nicht extra die neuen Werte abzufragen, sondern bekommt sie gleich mitgeliefert.
Der CtBotSerial-Service implementiert keinen eigenen Handler für diese Operation. Die Notify-Operation dient ihm nur dazu, andere Services über neue Kommandos zu informieren.

5.4.3 UserInterface-Service

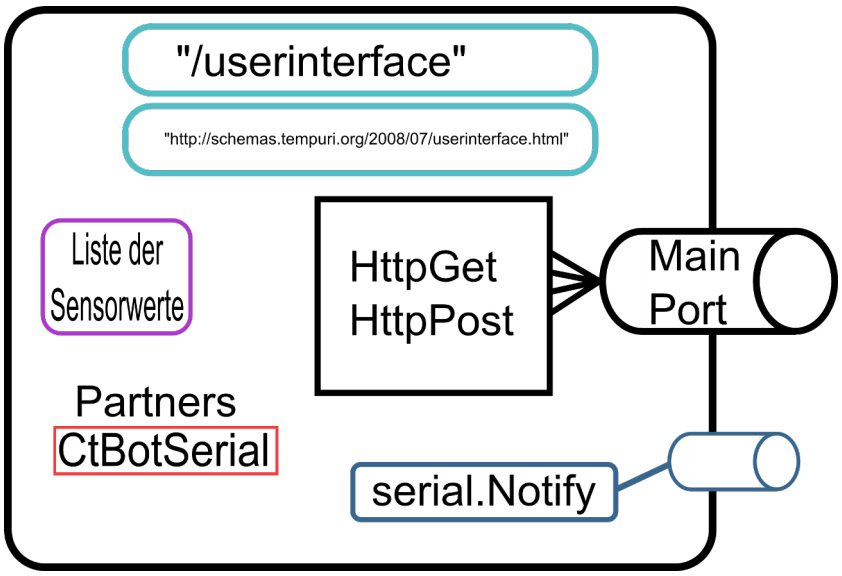


Abbildung 5.3: Vereinfachtes Service-Modell

Der UserInterface-Service stellt eine Weboberfläche zur Verfügung, über die die Sensorwerte abgelesen werden können und mit deren Hilfe der c't-Bot über Knöpfe ferngesteuert werden kann.

Grundsätzlich stehen verschiedene Methoden zur Verfügung dem Benutzer die Interaktion mit einem Service zu ermöglichen. Neben eher exotischen Varianten wie der Systemkonsole gibt es die klassischen Varianten einer grafischen Benutzeroberfläche oder einer Weboberfläche.
Eine grafische Benutzeroberfläche in C# ist mit der Klassenbibliothek WinForms erstellbar. Damit lassen sich typische fensterbasierte Oberflächen gestalten. Der Nachteil von WinForm ist, dass die Integration in einen Service ein paar Workarounds benötigt.
Eleganter ist es, dem Benutzer eine Weboberfläche zur Verfügung zu stellen. Da Services über das Netzwerk erreichbar sind und die Standardfunktionen der Webtechnologien unterstützen, lassen sich mit einfachen Mitteln schnell brauchbare Ergebnisse erzielen.

Standardmäßig werden im Webbrowser lediglich Informationen über den Service angezeigt, die kaum von Interesse sind. Mit wenigen Handgriffen kann aber bereits der momentane Status des Services als XML-Stream angezeigt werden. Ist eine benutzerfreundliche Ansicht oder aber gar eine Interaktion mit dem Service gewünscht, lässt sich mittels Extensible Stylesheet Language Transformations (XSLT) eine Weboberfläche gestalten, die neben den normalen HTML-Strukturen auch ein Design mittels CSS und sogar Scriptsprachen bietet.

Dieser Service stellt eine Weboberfläche zur Verfügung, auf der die aktuellen Sensordaten des c't-Bots angezeigt werden und dieser über Richtungsknöpfe gesteuert werden kann.

Der Service-Port dieses Services sieht folgendermaßen aus:

```
[ServicePort()]
public class UserInterfaceOperations :
  PortSet<DsspDefaultLookup,
          DsspDefaultDrop,
          Get,
          HttpGet,
          HttpQuery,
          HttpPost,
          serial.Notify>
  {}
```

Listing 5.4: UserInterface-Service-Operation-Port

Die Operation *Get* ist dafür zuständig, den momentanen Status des Services zurückzugeben. Im einfachsten Fall sieht der Handler dazu wie folgt aus:

```
[ServiceHandler(ServiceHandlerBehavior.Concurrent)]
  public virtual IEnumerator<ITask> GetHandler(Get get)
  {
    get.ResponsePort.Post(_state);
    yield break;
  }
```

Listing 5.5: Get-Handler

Ports sind ein freies und billiges Gut in der DSS. Jeder Nachricht kann ein Port angehängt werden, auf dem eine Antwort geposted werden kann. Im Grunde entspricht dieser Mechanismus einem Brief, dem ein frankierter und an sich selbst adressierter Rückumschlag beiliegt. Der Empfänger hat somit die Möglichkeit, auf eine Anfrage eine Antwort zu geben. Ohne einen Response-Port gibt es keine Möglichkeit für den Get-Handler, herauszufinden, wer die Get-Nachricht versendet hat.

Die Operationen HttpGet, HttpQuery und HttpPost sind notwendig, wenn der Service z. B. mit einem Webbrowser interagieren soll. Die HttpGet-Operation gibt dabei lediglich den aktuellen Status des Services aus, bzw. die mittels XSLT verknüpfte Weboberfläche.
Mit der HttpQuery-Operation können dem Service noch beliebige Daten übergeben werden. Wird z. B. im Adressfeld des Webbrowsers die Zeile

```
http://localhost:50000/userinterface?key=value
```

eingegeben, wird eine Nachricht erzeugt, die im HttpQuery-Handler aufläuft. Die Schlüssel-Werte-Paare sind als Arrays in der Nachricht enthalten und können von dort aus bequem weiterverarbeitet werden.
Mit der HttpPost-Operation können dem Service Daten aus einer HTML-Form übergeben werden. Die Schlüssel-Werte-Paare aus der HTML-Form lassen sich aus der Nachricht extrahieren und zur weiteren Verarbeitung in eine *NameValueCollection* speichern.

Mittels XSLT lassen sich nahezu beliebig umfangreiche Webseiten für einen Service gestalten, die eine Interaktion mit dem Benutzer erlauben. Der Vorteil von XSLT ist, dass sich Benutzeroberflächen gestalten lassen, die nicht nur auf dem Rechner benutzbar sind, auf dem der Service läuft, sondern überall dort, wo ein Zugang zum Internet zur Verfügung steht.

Um das zu erreichen, sind mehrere Schritte notwendig.
Zuerst muss eine XSL-Datei angelegt werden und diese in den Service integriert werden. Am einfachsten ist es, die Datei als eingebettete Ressource zu verwenden. Dadurch erspart man es sich die Datei immer mit kopieren zu müssen. Dazu muss Folgendes dem Service Code hinzugefügt werden:

```
[EmbeddedResource("Robotics.Services.CtBot.CtBotSerial.transform.
    xslt")]
string _transform = null;
```

Die XSL-Datei selber ist nach folgendem Schema aufgebaut.

```
1 <?xml version="1.0" encoding="utf-8" ?>
2 <xsl:stylesheet version="1.0"
3   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
4   xmlns:gui="http://schemas.tempuri.org/2008/07/userinterface.html"
      >
5
6   <!-- Hier Code einfügen -->
7
8 </xsl:stylesheet>
```

Listing 5.6: XSLT-Datei

Zeile 4 im Listing 5.6 macht den Service-State innerhalb der XSL-Datei unter dem Name *gui* verfügbar. Der Teil "http://schemas.tempuri.org/2008/07/userinterface.html" ist der Contract-Identifier, wie er in der Manifest-Datei angegeben wurde.
Möchte man nun die einzelnen Sensorwerte in einer Tabelle darstellen, kann dies mit einer HTML-Tabelle geschehen. Die einzelnen Sensoren sind dann mit *gui:SensorName* zu erreichen.

```
<table border="1">
  <tr class="odd">
    <th>Sensor</th>
    <th>Links</th>
    <th>Rechts</th>
  </tr>
  <tr class="even">
    <th>Border</th>
    <td align="right" >
      <xsl:value-of select="gui:BorderSensor/gui:Left"/>
    </td>
    <td align="right" >
      <xsl:value-of select="gui:BorderSensor/gui:Right"/>
    </td>
  </tr>
  <!-- usw. -->
</table>
```

Listing 5.7: XSLT-Datei

Möchte man nun die Weboberfläche benutzen, um den c't-Bot zu steuern, bzw. Daten an den Service zu schicken, kann dies einfach über eine HTML-Form geschehen.

```
<form id="Formvor" action="" method="post">
  <input name="button" type="hidden" value="vor" />
  <input type="submit" value="Vorwärts"/>
</form>
```

Listing 5.8: HTML-Form

Im Listing 5.8 wird ein Knopf erzeugt. Wenn dieser Knopf gedrückt wird, wird per HttpPost ein Schlüssel-Werte-Paar mit dem Schlüssel *button* und dem Wert *vor* an den Service gesendet.

5.5 Visual Programming Language

Mit der VPL und dem VPL-Editor hat man Zugriff auf alle erstellten Services, sofern die Standardeinstellungen für das jeweilige Service-Projekt nicht verändert wurden. Ein bestehender Service kann einfach in das Diagramm-Fenster gezogen werden und steht dort mit allen seinen Operationen zur Verfügung. Im Auslieferungszustand des Robotic-Studio stehen im VPL-Editor bereits für diverse Roboter fertige Services zur Verfügung. Weiterhin kann auf unterschiedliche Hilfs-Services zugegriffen werden, die z. B. Mathematische Funktionen oder grafische Benutzeroberflächen bereithalten.

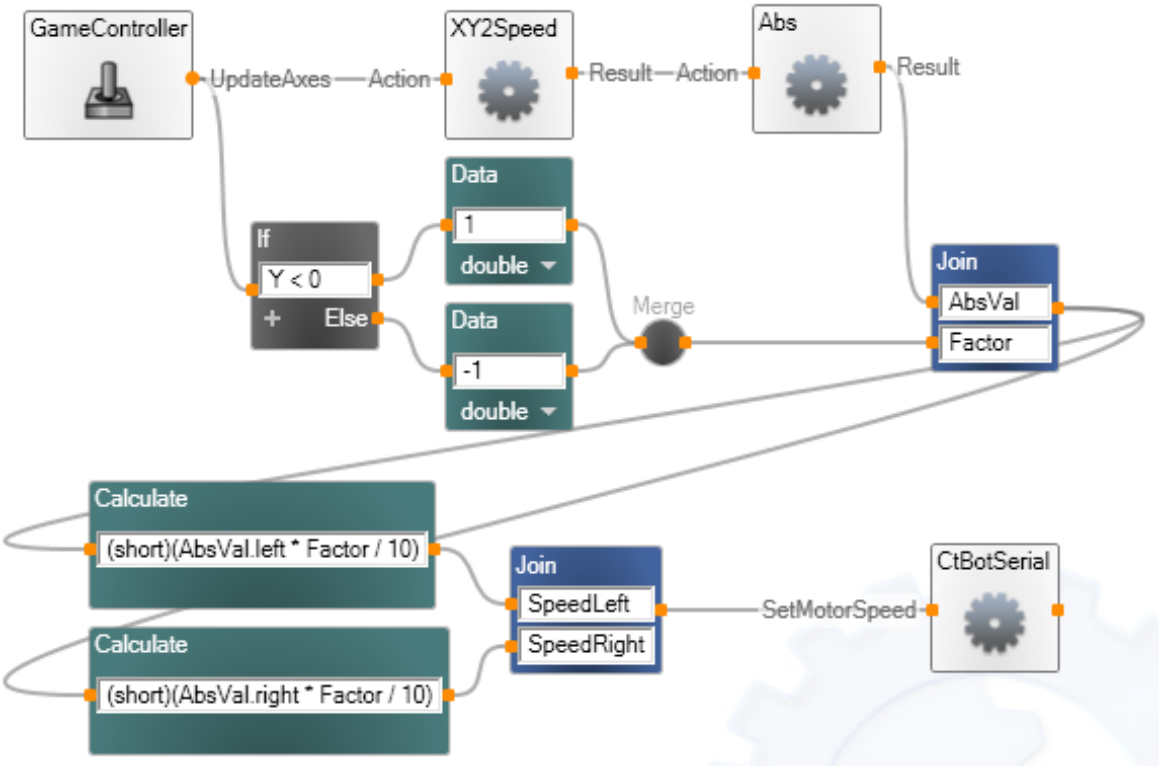


Abbildung 5.4: Hauptprogramm

Als Beispiel-Programm für die Verwendung der VPL wurde die Fernsteuerung des c't-Bots mit Hilfe eines Joysticks gewählt. Die nötigen Voraussetzungen für dieses Beispiel wurden in den vorangegangenen Kapiteln 5.3 und 5.4 beschrieben. Sowohl das Robotic-Studio als auch der c't-Bot senden und empfangen Kommandos.
In diesem Beispiel werden zwei fertige Services verwendet, die bereits im Robotic-Studio enthalten sind.

GameController Der *GameController* unterstützt DirectInput fähige Eingabegeräte wie Joysticks oder Gamepads. Diese Geräte müssen nur an den Computer angeschlossen

werden und werden dann automatisch von dem *GameController*-Service erkannt. Der Service benachrichtigt angeschlossene Services über Veränderungen am Eingabegerät und liefert die aktuellen Werte mit dieser Benachrichtigung mit. Für dieses Beispiel werden nur Veränderungen an den Achsen berücksichtigt.

MathFunctions Die Activity Calculate unterstützt nur die grundlegenden Rechenoperationen wie Addition, Subtraktion, Multiplikation, usw. Der Service *MathFunctions* dagegen stellt eine Reihe an gebräuchlichen mathematischen Funktionen zur Verfügung wie die trigonometrischen Funktionen, Potenzrechnung, Pi, usw.
Dazu werden am Eingang des Services ein oder mehrere Werte angegeben. Da das Ergebnis der Berechnung sofort zur Verfügung steht, verfügt der *MathFunction*-Service über keinen Notification-Port. Stattdessen wir das Ergebnis direkt über den Result-Port ausgegeben.

In Abbildung 5.4 ist das Diagramm zu sehen, welches die gesamte Joysticksteuerung beschreibt. Am Anfang des Datenflusses steht der Service *GameController*. Der GameController-Service informiert über Veränderungen, die an den Achsen oder den Knöpfen passieren.
In diesem Fall wird auf Veränderungen auf den Achsen gewartet. Bei einer Veränderung an den Achsen wird eine Nachricht erzeugt. Diese Nachricht wird durch eine Verzweigung einmal an die Activity *XY2Speed* und einmal an ein if-Statement weitergeleitet.

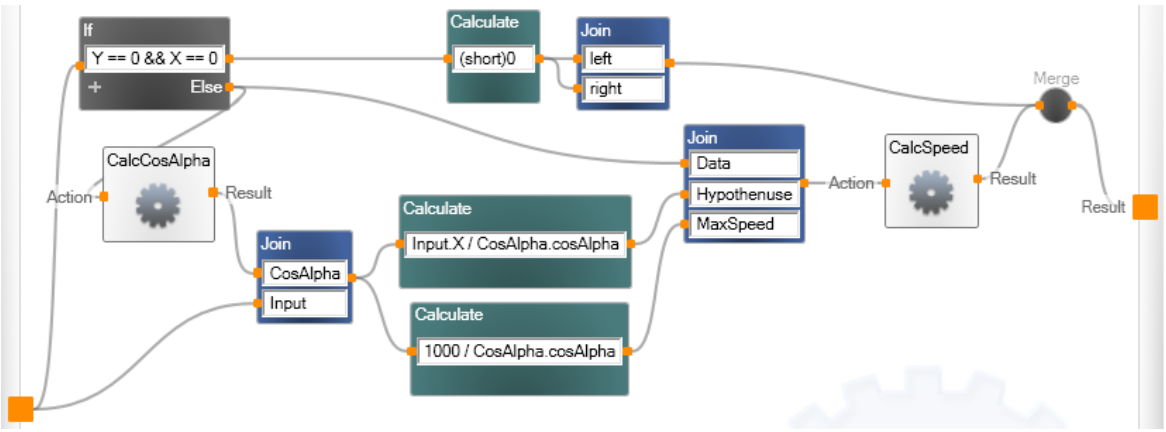


Abbildung 5.5: XY2Speed-Activity

In der Activity *XY2Speed* werden aus den X- und Y-Koordinaten des Joysticks entsprechende Werte für die Motoren des c't-Bots berechnet. Wie genau, ist weiter unten beschrieben. Die XY2Speed-Activity liefert in ihren Ergebnissen falsche Vorzeichen. Von den Ergebnissen für die Motorenwerte werden daher die Absolutwerte gebildet. Um aber nun auch das Rückwärtsfahren zu ermöglichen, wird mit der unteren Verzweigung, vom *GameController* Service aus, die Y-Achse des Joysticks abgefragt und in einen positiven oder negativen Faktor

umgewandelt. Mit diesem Faktor ist auch der Geschwindigkeitsbereich des c't-Bots einstellbar. Möchte man z. B. aus Sicherheitsgründen den Roboter maximal mit halber Kraft fahren lassen, kann einfach als Faktor 0.5 eingetragen werden. Der Joystick kann dann immer noch bis zum Anschlag bewegt werden, der c't-Bot fährt aber dennoch höchstens mit der halben Maximalgeschwindigkeit.
Der Absolutwert und der Faktor werden nun mittels einer Join Activity zu einer Nachricht zusammengefasst. Anschließend wird jeweils in den linken und den rechten Motorenwert der Faktor eingerechnet und durch 10 geteilt. Die Division durch 10 ist deshalb notwendig, weil der *GameController*-Service die Werte für die X- und Y-Achsen im Bereich von -1000 bis +1000 ausgibt. Dadurch lässt sich recht einfach der Motorenwert als Prozentangabe an den c't-Bot weitergeben.
Zum Schluss werden die faktorisierten Werte mit der *SetMotorSpeed*-Operation dem *CtBotSerial*-Service übergeben, welcher sie dann, in ein Kommando verpackt, an den c't-Bot sendet.

In Abbildung 5.5 ist das Innere der *XY2Speed*-Activity dargestellt. Die Activity errechnet aus den X- und Y-Koordinaten der Joystick-Achsen passende Werte für die c't-Bot Motoren. Mittels trigonometrischer Funktionen in der *CalcCosAlpha*-Activity wird der Winkel zwischen der X-Achse und der gedachten Linie, der Hypotenuse, vom Ursprung zum Koordinatenpunkt berechnet (h in Abb 5.7).

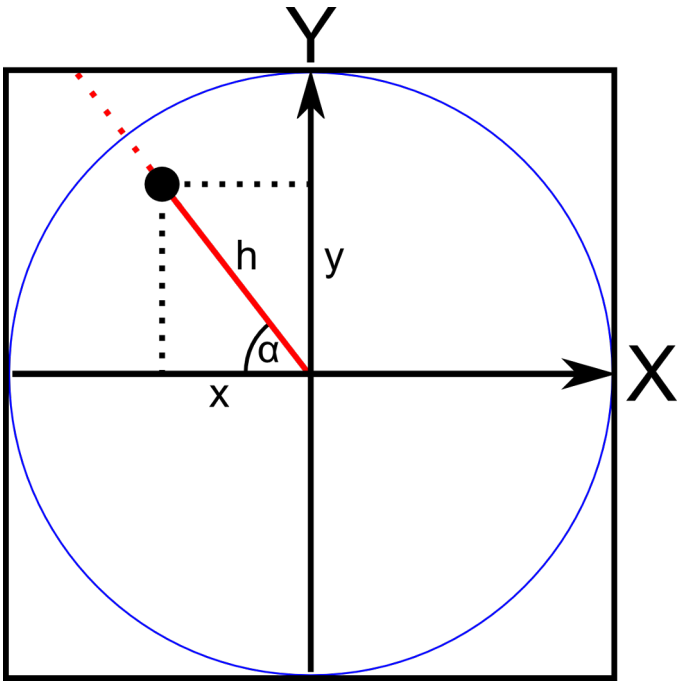


Abbildung 5.6: Joystick-Koordinatensystem

Mit dem berechneten Winkel kann nun die Hypotenuse selber berechnet werden. Idealerweise würde der Wert für die Hypotenuse in alle Richtungen 1000 betragen, in Abbildung 5.7 durch einen blauen Kreis dargestellt. Das Koordinatensystem des GameController-Services geht aber über diesen Bereich hinaus.
Deshalb wird der maximale Wert für die Hypotenuse berechnet, der in diesem Winkel möglich ist. Nun kann der verzerrte Hypotenusenwert zu dem maximalen Wert ins Verhältnis gesetzt werden und so der richtige Wert für die Motoren errechnet werden.

Die eigentlichen Motorenwerte werden nun in der Activity *CalcSpeed* berechnet. Abhängig

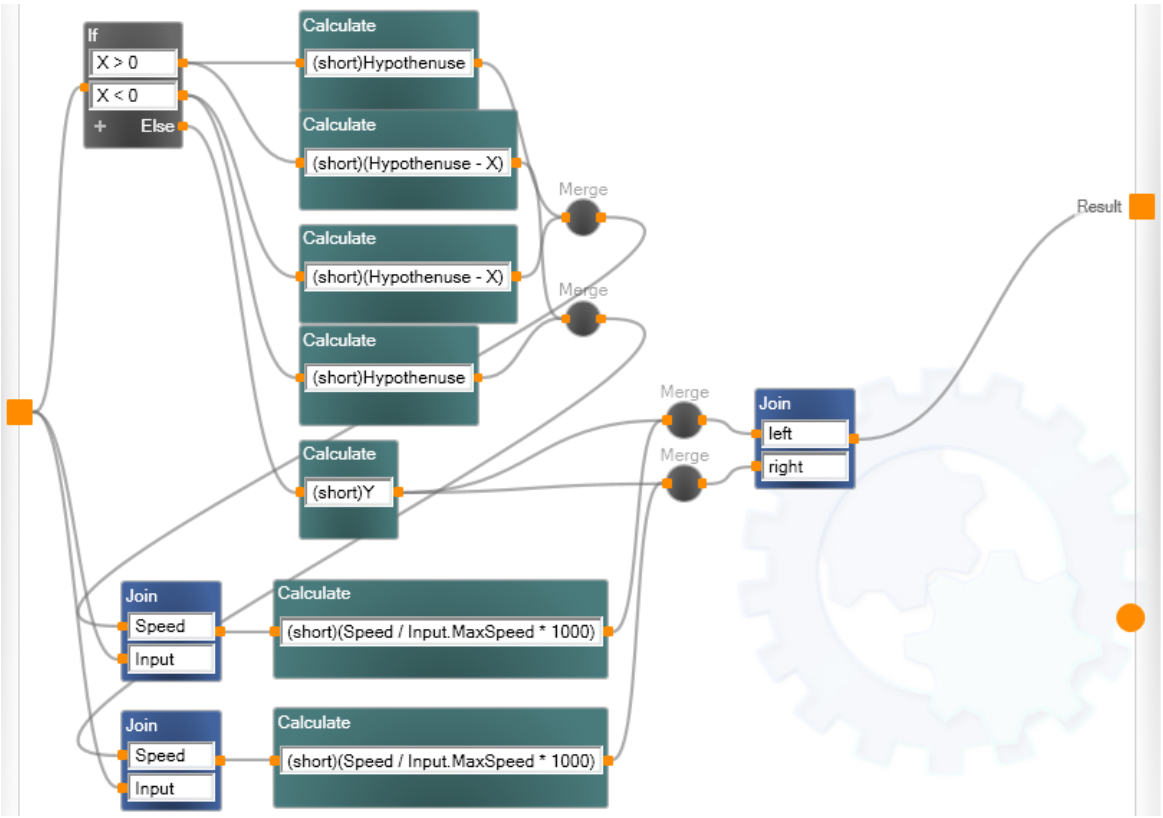


Abbildung 5.7: CalcSpeed-Activity

von der Position des Joysticks und der Hypotenuse werden die Werte für den linken und den rechten Motor berechnet.

5.6 Erfahrungen

5.6.1 c't-Bot

Wenn man den c't-Bot das erste Mal in den Händen hält muss er erst einmal zusammen gebaut werden. Der c't-Bot wird ausschließlich als Bausatz verkauft.
Mit einem Lötkolben, einem Seitenschneider und Schraubendrehern ist die Montage des c't-Bots auch für Laien möglich. Für den Roboter ist eine umfangreiche und bebilderte Aufbauanleitung vorhanden. Die komplette Montage mit einem ersten Testlauf ist somit an einem Nachmittag zu schaffen.
Die Erfahrungen mit dem Aufbau und der Wartung der diversen c't-Bots im Roboter-Labor der Hochschule für Angewandte Wissenschaften Hamburg zeigen, dass kein Roboter dem anderen gleicht. Die Sensoren und Motoren unterliegen anscheinend großen Schwankungen bei der Fertigung. Dadurch kann es vorkommen, dass ein c't-Bot wie auf Schienen geradeaus fährt, ein anderer c't-Bot aber stark zu einer Seite abdriftet. Auch die Sensoren unterliegen diesen Schwankungen.
Zum Glück gibt es eine Community für den c't-Bot, die einem bei Problemen weiterhelfen kann. Für viele Probleme wurden bereits Lösungen gefunden.

5.6.2 Robotic-Studio

Das Robotic-Studio lässt kaum Wünsche offen. Durch die Integration in die .NET-Umgebung stehen dem Programmierer nahezu unbegrenzte Möglichkeiten zur Verfügung, um Programme zu entwickeln. Durch die CCR und die DSS ist das Erstellen von nebenläufigem und unabhängigem Code ohne großen Aufwand möglich.

Vorteilhaft ist dabei die Nähe zu gängigen Webtechnologien und die Entlastung des Programmierers.
Das Robotic-Studio ist sehr komplex, sobald aber die grundlegenden Mechanismen verstanden wurden, können leistungsfähige Programme entwickelt werden.

In dem Robotic-Studio sind viele Tools integriert, die die Arbeit mit den Services erleichtern. Die meist verwendeten Tools sind *dssnewservice.exe* und *dsshost.exe*.
Mit der *dssnewservice.exe* lassen sich schnell und einfach neue Services erstellen. Neben dem gewünschten Zielordner, Namen und der Programmiersprache lassen sich über entsprechende Parameter viele zusätzliche Optionen für den neuen Service einstellen.
Das Erstellen neuer Services ist auch mit Visual Studio ohne Probleme möglich. In dem Assistenten zur Erzeugung neuer Projekte kann ein neues Roboter-Projekt ausgewählt werden. In den folgenden Schritten können dann die verschiedenen Optionen, wie Name, alternative Kontrakte oder Partner-Service angegeben werden.

Die *dsshost.exe* ist nötig, um Services auszuführen.

```
dsshost /p:50000 /t:50001 /m:"ServiceTutorial1.manifest.xml"
```

Damit wird ein neuer DSS-Node erzeugt auf dem der Service *ServiceTutorial1* läuft. Der Parameter /p:50000 gibt an, dass hereinkommende HTTP-Pakete auf Port 50000 erwartet werden. /t:50000 ist der für ankommende SOAP-Pakete zuständige Port.

Der Umfang des Robotic-Studio ist aber auch ein Nachteil. Dadurch, dass selbst für den einfachsten Service viele unterschiedliche Einstellungen vorgenommen und Konventionen eingehalten werden müssen, ist ein Einstieg in das Robotic-Studio keine einfache Angelegenheit.
Der Mechanismus Subscription ist ein sehr gutes Beispiel für die Komplexität des Robotic Studio. Im Grunde ist Subscription die Umsetzung des Observer-Design-Pattern für die Services. Nun gibt es aber von vornherein zwei Varianten um Subscription umzusetzen, Selective-Subscription und Event-Notification-Filtering. Bei Selective-Subscription kann der Beobachter-Service angeben, für welche Arten der Änderung er sich interessiert. Für jede Art muss er selbst einen Empfangs-Handler implementieren. Beim Event-Notification-Filtering kann der Beobachter-Service neben der Art der Veränderung auch noch die genaue Unterart angeben und in welchem Bereich die Veränderungen liegen darf.
Die genauen Mechanismen die den beiden Varianten zu Grunde liegen, sind auf den ersten Blick schwer nachzuvollziehen. Diese hohen Grundvoraussetzungen, die vom Programmierer erwartet werden, gelten für weite Bereiche des Robotic-Studio. Ohne intensive Recherche in Büchern oder anderen Quellen ist es sehr schwierig, die Mechanismen zu begreifen und auszunutzen.

Ansonsten ist die Arbeit mit den Services klar strukturiert. Für jede Funktion, die der Service anbieten soll, wird in dem Main-Port eine Operation eingefügt. Für jede dieser Operationen wird ein passender Handler implementiert. Dieser Handler erhält die für ihn passende Nachricht und stellt die eigentliche Funktion zur Verfügung.
In der Dokumentation des Robotic-Studio sind für die meisten Aspekte des Robotic-Studio ausführliche Tutorien vorhanden. Diese Tutorien geben Aufschluss darüber, wie die einzelnen Funktionen des Robotic-Studio eingesetzt werden können.

5.6.3 Visual Programming Language

Im Umgang mit dem VPL-Editor wird vom Benutzer viel Disziplin abverlangt. Für den erfahrenen Programmierer ist das Konzept einer Datenfluss-orientierten Programmiersprache zunächst etwas ungewohnt und verlangt ein Umdenken beim Programmieren. Möchte man im Verlauf eines Programms auf Daten zugreifen, so ist es notwendig, diese Daten in Form einer Nachricht dem entsprechenden Block zuzuführen. Eine Activity bzw. ein Service hat

immer nur Zugriff auf die Daten, die ihm im Eingang als Nachricht zur Verfügung stehen. Möchte man z. B. mit einer Activity auf einen Wert einer Variablen zugreifen, muss dieser Wert als Nachricht empfangen worden sein. Abbildung 5.8 enthält ein Beispiel, das diesen

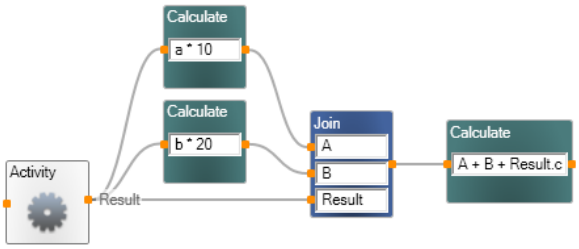


Abbildung 5.8: VPL Datenfluss

Umstand verdeutlicht: Die Activity liefert als Ergebnis drei Werte a, b und c. Der Wert a wird mit 10 multipliziert, der Wert b wird mit 20 multipliziert. Möchte man nun die Ergebnisse der Multiplikationen mit dem Wert c addieren, ist es notwendig die einzelnen Ergebnisse mit einem Join zu einer Nachricht zu verbinden um sie dann anschließend miteinander addieren zu können.
Das führt dazu, dass umfangreiche Programme von Anfang an sehr gut geplant sein müssen. Zwar ist das Verschieben von Programmblöcken ohne weiteres möglich, aber sollte man z. B. *Merge* mit *Join* verwechselt haben, erfordert dieses einen ziemlichen aufwändigen Programmumbau.

5.6.4 Arbeitsverteilung auf Robotic-Studio und c't-Bot

Wie im Kapitel 4.4 beschrieben, ist die genaue Aufteilung der verschiedenen Aufgaben auf das Robotic-Studio und den c't-Bot von entscheidender Bedeutung.
Je nach Aufgabenstellung ist es notwendig bestimmte Aufgaben entweder vom c't-Bot oder vom Robotic-Studio übernehmen zu lassen.

5.6.4.1 Autonomer c't-Bot

Diese Variante entspricht im Wesentlichen dem Grundzustand des c't-Bots. Einfache Aufgaben lassen sich recht einfach in passende Verhalten für den c't-Bot umsetzen. Je umfangreicher die Aufgabenstellung wird, desto schwieriger wird es, die verfügbaren Ressourcen des c't-Bots effektiv zu nutzen. Da der c't-Bot nur über eine einfache arithmetische Recheneinheit verfügt, die Operationen wie Multiplikation und Division in mehreren Rechenschritten verarbeiten muss, ist es notwendig, mit den verfügbaren Ressourcen sparsam umzugehen.

5.6.4.2 Fernsteuerung durch Robotic-Studio

Das Robotic-Studio läuft auf einem Computer, der über ein Vielfaches der Ressourcen des c't-Bots verfügt. Theoretisch sollte das Robotic-Studio in der Lage sein alle Aufgaben, die der c't-Bot selber schaffen kann, zu übernehmen, ohne dass irgendwelche Probleme für den c't-Bot oder das Robotic-Studio entstehen sollten.
In der Praxis zeigt sich aber, dass schon relativ simple Aufgaben nicht vom Robotic-Studio alleine bewältigt werden. Der Hauptgrund hierfür ist die Art, wie der c't-Bot und das Robotic-Studio untereinander Daten austauschen.
Die Sensordaten, die der c't-Bot ermittelt müssen in ein Kommando verpackt werden, welches anschließend über die RS232-Schnittstelle des c't-Bots versendet wird. Das Kommando wird nun von dem WiPort-Modul auf dem Erweiterungsboard empfangen und für den Transport in einem TCP/IP-Netzwerk vorbereitet. Nach der Übertragung über das Netzwerk läuft das Paket auf einem emulierten COM-Port auf und wird dort wieder in einen einfachen seriellen Datenstrom umgewandelt. Jetzt stehen die Daten dem Robotic-Studio zur Verfügung.
Versendet das Robotic-Studio einen Befehl an den c't-Bot nimmt das Kommando den entgegengesetzten Weg.

Ein einfaches Beispiel ist die Verfolgung einer Linie. Wichtig hierfür sind nur die Liniensensoren des c't-Bots. Die aktuellen Werte werden an das Robotic-Studio gesendet. Hier werden die Daten ausgewertet und die Stellwerte der Motoren berechnet. Die Werte werden anschließend an den c't-Bot gesendet, welcher sie sofort umsetzt.
In der Praxis zeigt sich aber, dass der c't-Bot Schwierigkeiten hat, der Linie zu folgen. Die Daten brauchen relativ lange, bis sie an der Gegenstelle angekommen sind. Sind die Daten für die Motoren berechnet, ist der c't-Bot meistens bereits so weit von der Linie abgekommen, dass er sie nicht mehr finden kann.

In Kapitel 4.4.2 wurde ausgerechnet, dass für eine sinnvolle Linienverfolgung der c't-Bot alle 10 ms einen neuen Sensorwert an das Robotic-Studio senden muss und das Robotic-Studio innerhalb von 10 ms aus diesen Werten neue Stellwerte für die Motoren an den c't-Bot senden muss.

Wenn man das Framework des c't-Bots genauer betrachtet, erkennt man, dass der c't-Bot die Sensorwerte ausreichend oft aktualisiert. Allerdings werden die Sensorwerte nur alle 100 ms versendet. Damit erreicht der c't-Bot lediglich ein Zehntel der benötigten Sendefrequenz. Den c't-Bot die Daten einfach häufiger senden zu lassen, hilft aber auch nicht weiter.
Die RS232-Schnittstelle ist auf 38400 Baud eingestellt. Das bedeutet, dass ein Byte in 0,26 ms übertragen wird. Ein Kommando besteht aus 11 Bytes, woraus folgt, dass ein Kommando in $11\ Bytes * 0,26\ ms = 2,86\ ms$ übertragen wird. Standardmäßig werden insgesamt 11 Sensor-Kommandos übertragen. D.h., dass alle 11 Kommandos in $11 * 2,86\ ms =$

$31,46\ ms$ übertragen sind.
Für andere Aufgaben bleiben $100\ ms - 31,46\ ms = 68,54\ ms$ übrig.

Eine Überwachung der Datenpakete im Netzwerk zeigt, dass der emulierte COM-Port alle 20 ms die Daten mit dem WiPort-Modul austauscht. Dabei kann sich ein Kommando auf zwei dieser Pakete aufteilen. Im schlimmsten Fall entsteht somit eine Verzögerung der Daten von 40 ms pro Senderichtung.

In praktischen Versuchen hat sich herausgestellt, dass eine Linienverfolgung mit dem Robotic-Studio kaum möglich ist. Bei einer Geschwindigkeit von 0,1 m/s fährt der c't-Bot ständig von der Linie und findet nicht wieder zurück auf die Spur. Um der Linie zuverlässig folgen zu können, muss die Geschwindigkeit des c't-Bots sehr stark reduziert werden. Die Geschwindigkeit ist dann aber so niedrig, dass die Motoren kaum die Kraft besitzen, den c't-Bot vorwärts zu bewegen.

5.6.4.3 Kooperation von c't-Bot und Robotic-Studio

Sinnvoll ist es, die Leistungsfähigkeit des c't-Bots mit der Leistungsfähigkeit des Robotic-Studio zu kombinieren. Der c't-Bot übernimmt mindestens die Aufgaben, die eine schnelle Reaktion erfordern. Zu diesen Aufgaben gehören u. a. Abgründe erkennen, Hindernissen ausweichen oder Linien verfolgen.

Als Beispiel sei hier die Joysticksteuerung des c't-Bots genannt, die in der VPL realisiert wurde (s. Kapitel 5.5).
Das Robotic-Studio ist für die komplette Verarbeitung der Joystickdaten zuständig. Das Studio empfängt die Joystickdaten und rechnet die X- und Y-Koordinaten der Joystickachsen in Werte für den Motor um. Für die Umrechnung der Koordinaten in Motorwerte sind Operationen nötig, die der c't-Bot nicht unterstützt, bzw. nur sehr schwer umzusetzen sind. Die Operationen müssten dann in Rechenschritte zerlegt werden, die einen sehr hohen Rechenaufwand bedeuten und die Kapazitäten des c't-Bots übersteigen.
Der c't-Bot übernimmt die Aufgaben Daten zu erfassen, Motoren zu stellen und Abgründe zu erkennen. Diese Aufgaben stellen für den c't-Bot keine große Belastung dar.

Die Geschwindigkeit und Verzögerung der Datenübertragung sorgt dafür, dass die neuen Motorenwerte erst mit einer kleinen Verzögerung im c't-Bot ankommen. Allerdings ist die Trägheit des Menschen um ein Vielfaches höher, sodass die Datenübertragung lediglich einen kaum spürbaren Einfluss auf die Steuerung nimmt.

5.6.5 Kommunikation

5.6.5.1 Hardware

Die Entscheidung, für die Kommunikation zwischen c't-Bot und Robotic-Studio auf die RS232-Schnittstelle und WLAN zurückzugreifen, stellt sich im Einsatz als sehr praktikabel heraus.
Die RS232-Schnittstelle ist durch ihr einfaches Design ideal, um auf einem Micro-Controller eingesetzt zu werden. Mit zwei Anschlüssen, die für das Senden und Empfangen von Daten benötigt werden, wird kaum Platz verschwendet. Die Geschwindigkeit der seriellen Schnittstelle ist prinzipiell ein Schwachpunkt, fällt aber bei den hier vorgestellten Beispielen nicht weiter ins Gewicht.
Das WiPort-Modul auf dem Erweiterungsboard des c't-Bots ist in seiner Handhabung sehr benutzerfreundlich. Um das Modul zu konfigurieren, kann eine der RS232-Schnittstellen verwendet werden. Das Menü ist aber auch über das Netzwerk verfügbar und kann über WLAN oder ein Netzwerkkabel erreicht werden.
Im Betrieb haben sich aber zwei Schwachstellen herausgestellt. Die eine ist die Stromversorgung. Wird der c't-Bot über Batterien mit Strom versorgt, reduziert das WiPort-Modul die Einsatzbereitschaft des c't-Bots erheblich.
Die zweite Schwachstelle ist die Hitzeentwicklung des WiPort-Moduls. Das Display des c't-Bots ist direkt über dem WiPort-Modul angebracht. Dadurch staut sich die erwärmte Luft zwischen WiPort und Display. Das hat zur Folge, dass sich der WiPort bei zu großer Hitze automatisch abschaltet. Der c't-Bot unterbricht dadurch sein Programm, bis das WiPort Modul ausgeschaltet wurde.

5.6.5.2 Software

Das verwendete Protokoll (s. 5.2.2) ist ursprünglich für den Datenaustausch zwischen dem c't-Bot und dem Simulator c't-Sim entwickelt worden. Das Robotic-Studio stellt einen ähnliche Einsatzbereich für das Protokoll dar.
Eine Implementierung des Protokolls im Robotic-Studio ist ohne großen Aufwand möglich und lässt sich einfach in eine Service-Struktur einfügen.

6 Fazit

6.1 Zusammenfassung

Diese Bachelorarbeit befasst sich mit der Integration des c't-Bots in das Robotic-Studio.
Zuerst wurden der c't-Bot und das Robotic-Studio vorgestellt. Beim c't-Bot wurden die einzelnen Bauteile und ihre Funktionen erläutert. Zudem wurde auf das Framework des c't-Bots eingegangen und die Subsumption-Architektur vorgestellt.
Für das Robotic-Studio wurden die unterschiedlichen Bestandteile der CCR und der DSS erklärt sowie Einsatzmöglichkeiten für den c't-Bot beschrieben.

Für die Zusammenarbeit von c't-Bot und Robotic-Studio wurden drei grundsätzliche Wege vorgestellt, die sich für unterschiedliche Herangehensweisen für die Integration eignen.
Der erste Weg beschreibt einen autonom arbeitenden c't-Bot, der ohne Robotic-Studio auskommt. Das Robotic-Studio wird in dieser Variante dazu verwendet, die Sensordaten für den Benutzer übersichtlich darzustellen. Ansonsten werden alle Aufgaben vom c't-Bot autonom gelöst.
Im zweiten Weg wird der c't-Bot vom Robotic-Studio ferngesteuert. Der c't-Bot muss lediglich die Sensordaten ermitteln und an das Robotic-Studio senden. Das Robotic-Studio berechnet dann aus den Sensordaten Befehle für den c't-Bot. Diese Befehle werden an den c't-Bot gesendet und ausgewertet.
Als dritter Weg wurde die Kooperation von c't-Bot und Robotic-Studio vorgestellt. Hier werden die Vorteile des c't-Bots und des Robotic-Studio miteinander verbunden. Die Nachteile, die durch die beiden anderen Wege entstehen, können so fast vollständig vermieden werden.

Zusätzlich wurde ein Einblick in die Visual Programming Language gegeben. Mit der VPL sollen Programme mit intuitiven visuellen Mitteln schnell und einfach umgesetzt werden können.

6.2 Bewertung

6.2.1 c't-Bot

Der c't-Bot ist ein idealer Roboter für den Einstieg in die Robotik. Mit seinen zahlreichen Sensoren und Aktoren und dem günstigen Preis ist er den meisten anderen Robotern aus dem Hobbybereich überlegen.
Allerdings hat der c't-Bot auch Nachteile. So neigen die verwendeten Bauteile dazu, nicht immer das zu machen, was von ihnen gefordert wird. Auch müssen die Sensoren für jeden c't-Bot separart kallibriert werden. Ebenso laufen die Motoren selten gleichmäßig, sodass eine Fahrt geradeaus meistens dem Zufall unterliegt.

Der größte Vorteil des c't-Bots ist der Umstand, dass sämtliche Komponenten, sei es Hard- oder Software, genau dokumentiert und offen zugänglich sind. Somit kann der gesamte c't-Bot den eigenen Wünschen angepasst werden.
Im Roboterlabor der Hochschule für Angewandte Wissenschaften Hamburg wurde der c't-Bot bereits mit einem ZigBee-Funkmodul und einem selbst entwickelten Atmel-Evaluationsboard ausgestattet. Weiter Modifikationen werden nur von den Möglichkeiten bzw. der Fantasie des Anwenders beschränkt.

Der größte Schwachpunkt des c't-Bots im Einsatz mit dem Robotic-Studio ist die Kommunikation. Dadurch, dass sämtliche IO-Pins des Atmels belegt sind und nur die RS232-Schnittstelle für die Kommunikation übrig bleibt, sind die Möglichkeiten sehr begrenzt. Umso wichtiger ist es die vorhandenen Ressourcen so effektiv wie möglich einzusetzen.
Eine Möglichkeit wäre, die Kommando-Struktur, die im c't-Bot Framework enthalten ist zu verschlanken und dadurch ein besseres Verhältnis der Nutzdaten zum Overhead zu erreichen. Eine nützliche Funktion wäre auch, wenn das Robotic-Studio nur die Daten empfängt, die es auch tatsächlich benötigt. Momentan werden alle Sensordaten übertragen, aber nicht unbedingt benötigt. Hier könnte ein spezielles Kommando vom Robotic-Studio die benötigten Sensoren an-, bzw. ausschalten.

6.2.2 Robotic-Studio

Das Robotic-Studio ist eine sehr leistungsfähige Programmierumgebung. Durch die CCR und DSS stehen Datenstrukturen und Operationen zur Verfügung, die die Programmierung der kritischen Abschnitte erleichtern. Kritische Abschnitte, die sonst umständlich mit z. B. Semaphoren behandelt werden müssen, werden im Robotic-Studio automatisch geschützt. Ein Handler, der einen kritischen Abschnitt betritt, muss nur mit dem Attribut *[ServiceHandler(ServiceHandlerBehavior.Exclusive)]* versehen werden. Dadurch stellt die CCR automatisch sicher, dass einer Task exklusiver Zutritt zu einem kritischen Abschnitt gewährt wird.

Das Robotic-Studio ist sehr komplex. Jeder Aspekt des Studios kann mehrere Ausprägungen haben und ist oft mit diversen Einstellmöglichkeiten versehen.

6.2.3 Visual Programming Language

Die VPL ist Fluch und Segen zugleich. Das Versprechen, mit grafischen Elementen eine intuitive Programmierumgebung zu bieten, kann die VPL nur bedingt einhalten. Es ist durchaus möglich, einfache Programmstrukturen so darzustellen, dass sie auf einen Blick verständlich sind. Aber bei etwas komplexeren Programmen geht der Überblick schnell verloren. Zwar können einzelne Programmabschnitte in Activities eingekapselt werden, allerdings wird das Navigieren durch die einzelnen Programmelemente stark erschwert.

Ein verständliches Programm mit der VPL setzt voraus, dass die Services an sich sehr leistungsstark sind und einen großen Teil der Logik in sich aufnehmen. Dadurch kann ein schlankeres Design erreicht werden. Allerdings ist dann innerhalb der VPL kaum etwas von dem eigentlichen Programm ersichtlich. Das kann für den Einsteiger ein Vorteil sein, reduziert aber auch das Verständnis des Programms.

Wünschenswert wäre es, wenn für die VPL-Diagramme eine Benutzeroberfläche innerhalb des Editors gestaltet werden könnte. Die wenigen Oberflächen die für den VPL-Editor zur Verfügung stehen sind oft nicht ausreichend. Einfache Anzeigen, wie Statusbalken fehlen ebenso wie Kontrollelemente, wie Schieberegler oder Eingabefelder.
Sind solche Ein- und Ausgabeelemente im Programm nötig, müssen sie als Services implementiert werden.

6.3 Aussichten

Im c't-Bot sind die einzelnen Aufgaben in Verhalten realisiert, die an das Subsumption-Modell angelehnt sind. Im Robotic-Studio sind alle Services gleichberechtigt. Es ist kein Mechanismus vorgesehen, Services oder einzelne Aufgaben zu priorisieren.
Es wäre aber eine interessante Aufgabe, die Subsumption Architektur auch auf das Robotic-Studio umzusetzen, bzw. die Machbarkeit zu untersuchen.

Für den c't-Bot existiert bereits ein Simulator (c't-Sim), der eine einfache Umwelt simulieren kann, in der sich der c't-Bot bewegt. Hier kann es sich lohnen zu untersuchen, inwieweit sich der c't-Sim und das Robotic-Studio kombinieren lassen bzw. ob sich der c't-Sim durch dem Simulator des Robotic-Studios sinnvoll ersetzen lässt.

Bisher unterstützt der Programmcode für das Robotic-Studio nur einen einzigen c't-Bot. Gerade die hohe Skalierfähigkeit der Services lässt aber ein Szenario zu, in dem mehrere

c't-Bots eine gemeinsame Aufgabe lösen und zentral koordiniert werden.
Hierfür wird ein zentraler Service benötigt, der alle c't-Bots koordiniert.

Die Kommunikation zwischen c't-Bot und Robotic-Studio ist bereits sehr effizient. Aber gerade unter Berücksichtigung der knappen Ressourcen des c't-Bots könnte eine Überarbeitung sinnvoll sein.
Eine einfache Verbesserung wäre das Robotic-Studio auswählen zu lassen, welche Sensorwerte überhaupt übertragen werden sollen. Bei der Joysticksteuerung werden überhaupt keine Sensorwerte des c't-Bots benötigt. Bei der Kartografierung der Umgebung werden z. B. nur die Abstandssensoren benötigt. Die anderen Sensorwerte übertragen zu lassen, bedeutet nur einen unnötigen Mehraufwand für den c't-Bot. Um das zu vermeiden, könnte sich der *CtBotSerialService* (s. 5.4.2) merken, welche Services welche Sensorwerte benötigen und einen Befehl an den c't-Bot senden, der die nicht benötigten Sensoren abschaltet.
Dadurch ließe sich auch Strom sparen und die Einsatzbereitschaft des c't-Bots zeitlich verlängern.

Im November 2008 hat Microsoft gleich zwei große Veröffentlichungen im Rahmen des Robotic-Studio gemacht.
Die erste Veröffentlichung war eine neue Version des Robotic-Studio unter dem Namen *Microsoft Robotics Developer Studio 2008*. Eine abgespeckte Express-Version steht kostenlos zum Download[1] zur Verfügung. Die Vollversion ist entweder käuflich zu erwerben oder aber über das MSDNAA-Netzwerk erhältlich.
Die zweite Veröffentlichung ist das *CCR and DSS Toolkit 2008*. Da sich die CCR und die DSS auch außerhalb von Roboterprojekten einsetzen lassen, hat sich Microsoft entschlossen, diese beiden Komponenten des Robotic-Studios als eigenständiges Toolkit zu veröffentlichen.

[1] http://go.microsoft.com/fwlink/?LinkID=134518
Stand 07.12.2008

Literaturverzeichnis

[Bachfeld 2006] BACHFELD, Daniel: Steuermann. In: *c't* 9 (2006), S. 222–226

[Benz 2006] BENZ, Benjamin: Kammerjäger — Fehlersuche in elektronischen Schaltungen. In: *c't* 12 (2006), S. 240–244

[Benz und König 2006] BENZ, Benjamin ; KÖNIG, Peter: Virtuelle Spielgefährten — Simulator für c't-Bots. In: *c't* 3 (2006), S. 186–191

[Birk 2006] BIRK, Andreas: An der nächsten Ecke links ... — Karten bauen (nicht nur) mit dem c't-Bot. In: *c't* 19 (2006), S. 198–205

[Brooks 1986] BROOKS, R. A.: A Robust Layered Control System for a Mobile Robot. In: *IEEE Journal of Robotics and Automation* 2 (1986), März, Nr. 1, S. 14–23

[Buschmann] BUSCHMANN, Frank: *Pattern-orientierte Softwarearchitektur : ein Pattern-System*. Addison-Wesley, year =

[Dustdar u. a. 2003] DUSTDAR, Schahram ; GALL, Harald ; HAUSWIRTH, Manfred: *Software-Architekturen für Verteilte Systeme*. Springer, 2003. – ISBN 3-540-43088-1

[Evers 2006a] EVERS, Torsten: Ausgang gesucht! — Komplexe Verhalten für den c't-Bot selbst entwickelt. In: *c't* 10 (2006), S. 236–239

[Evers 2006b] EVERS, Torsten: Wo bin ich? — Positionsbestimmung für den c't-Bot. In: *c't* 13 (2006), S. 226–229

[Fielding 2000] FIELDING, Roy T.: *Architectural Styles and the Design of Network-based Software Architectures*. 2000. – URL http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm. – [Online; Stand 12. November 2008]

[Grimmer 2006] GRIMMER, Christoph: Hohe Schule. In: *c't* 7 (2006), S. 218–223

[heise 2008] HEISE: *c't Projekte - c't-Bot und c't-Sim*. 2008. – URL http://www.heise.de/ct/projekte/ct-bot/. – [Online; Stand 10. Dezember 2008]

[Johns und Taylor 2008] JOHNS, Kyle ; TAYLOR, Trevor: *Professional Microsoft Robotics Developer Studio*. Wiley Publishing, Inc, 2008. – ISBN 978-0-470-14107-6

[Melzer 2007] MELZER, Ingo: *Service-orientierte Architekturen mit Web Services*. Elsevier, Spektrum Akademischer Verlag, 2007. – ISBN 978-3-8274-1885-2

[Microsoft 2008a] MICROSOFT: *DSS Service Components*. 2008. – URL http://msdn.microsoft.com/en-us/library/bb648760.aspx. – [Online; Stand 13. Dezember 2008]

[Microsoft 2008b] MICROSOFT: *Microsoft Robotics*. 2008. – URL http://msdn.microsoft.com/en-us/library/bb881626.aspx. – [Online; Stand 15. Oktober 2008]

[Microsoft 2008c] MICROSOFT: *SerialPort Class*. 2008. – URL http://www.swe.uni-linz.ac.at/people/schiffer/se-96-19/se-96-19.htm. – [Online; Stand 14. Oktober 2008]

[Microsoft 2008d] MICROSOFT: *Welcome to Robotics Developer Studio*. 2008. – URL http://msdn.microsoft.com/en-us/library/bb648760.aspx. – [Online; Stand 13. Dezember 2008]

[Morgan 2008] MORGAN, Sara: *Programing Microsoft Robotics Studio*. Microsoft Press, 2008. – ISBN 0-7356-2432-1

[Nehmzow 2002] NEHMZOW, Ulrich: *Mobile Robotik — Eine praktische Einführung*. Springer, 2002. – ISBN 3-540-42858-5

[Nielsen und Chrysanthakopoulos] NIELSEN, Henrik F. ; CHRYSANTHAKOPOULOS, George: *Decentralized Software Services Protocol DSSP 1.0*. – URL http://purl.org/msrs/dssp.pdf. – [Online; Stand 13. November 2008]

[Prescod 2008] PRESCOD, Paul: *Roots of the REST/SOAP Debate*. 2008. – URL http://www.prescod.net/rest/rest_vs_soap_overview/. – [Online; Stand 12. November 2008]

[Rosseburg 2007] ROSSEBURG, Kai: *Entwicklung einer Programmierumgebung für roboterbasierten Informatikunterricht an Schulen*, Hochschule für Angewandte Wissenschaften Hamburg, Bachelor, 2007

[Schiffer 1996] SCHIFFER, Stefan: *Visuelle Programmierung - Potential und Grenzen*. 1996. – URL http://msdn.microsoft.com/en-us/library/system.io.ports.serialport.aspx. – [Online; Stand 26. Mai 2008]

[Schiffer 1998] SCHIFFER, Stefan: *Visuelle Programmierung*. Addison-Wesley, 1998. – ISBN 3-8273-1271-x

[Tanenbaum und van Steen 2008] TANENBAUM, Andrew S. ; STEEN, Maarten van: *Verteilte Systeme — Prinzipien und Paradigmen*. Pearson Studium, 2008. – ISBN 978-3-8273-7293-2

[Thiele 2006] THIELE, Thorsten: An der Leine. In: *c't* 7 (2006), S. 223

[Thiele und Thiede 2006a] THIELE, Thorsten ; THIEDE, Carl: Hallo Welt! — Aufbau und Inbetriebnahme des c't-Bot. In: *c't* 4 (2006), S. 208–213

[Thiele und Thiede 2006b] THIELE, Thorsten ; THIEDE, Carl: Spielgefährten — Roboter für Löter, Simulator für Soft-Werker. In: *c't* 2 (2006), S. 130–135

[Weerawarana u. a. 2005] WEERAWARANA, Sanjiva ; CURBERA, Francisco ; LEYMANN, Frank ; STOREY, Tony ; FERGUSON, Donald F.: *Web Services Platform Architecture*. Prentice Hall, 2005. – ISBN 0-13-148874-0

[Wikipedia 2008a] WIKIPEDIA: *Protokoll (Informatik) — Wikipedia, Die freie Enzyklopädie*. 2008. – URL http://de.wikipedia.org/w/index.php?title=Protokoll_(Informatik)&oldid=51175962. – [Online; Stand 31. Oktober 2008]

[Wikipedia 2008b] WIKIPEDIA: *Representational State Transfer — Wikipedia, Die freie Enzyklopädie*. 2008. – URL http://de.wikipedia.org/w/index.php?title=Representational_State_Transfer&oldid=45689372. – [Online; Stand 16. Mai 2008]

Glossar

Kürzel	Beschreibung
API	Programmierschnittstelle
CCR	Concurrency and Coordination Runtime
CLR	Common Language Runtime; Laufzeitumgebung des .NET Frameworks
COM-Port	Communication Port; RS232
CPU	Hauptprozessor
DLL	Dynamic Link Library
DSS	Decentralized Software Services
FIFO	First In First, First Out
GUID	Globally Unique Identifier
HTTP	Hypertext Transfer Protocol
IO	Input Output
LCD	Flüssigkristallanzeige
MMC	Multimedia Card; digitales Speichermedium
PWM	Pulsweitenmodulation
REST	Representational State Transfer
SD	Secure Digital; digitales Speichermedium
SPI	Serial Peripheral Interface
TWI	Two Wire Interface; auch I^2C genannt

Kürzel	**Beschreibung**
URI	Uniform Resource Identifier
VPL	Visual Programming Language
XML	Extensible Markup Language
XNA	XNA is Not Acronymed
XSL	Extensible Stylesheet Language
XSLT	Extensible Stylesheet Language Transformation

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §22(4) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 18. Dezember 2008	
Ort, Datum	Unterschrift