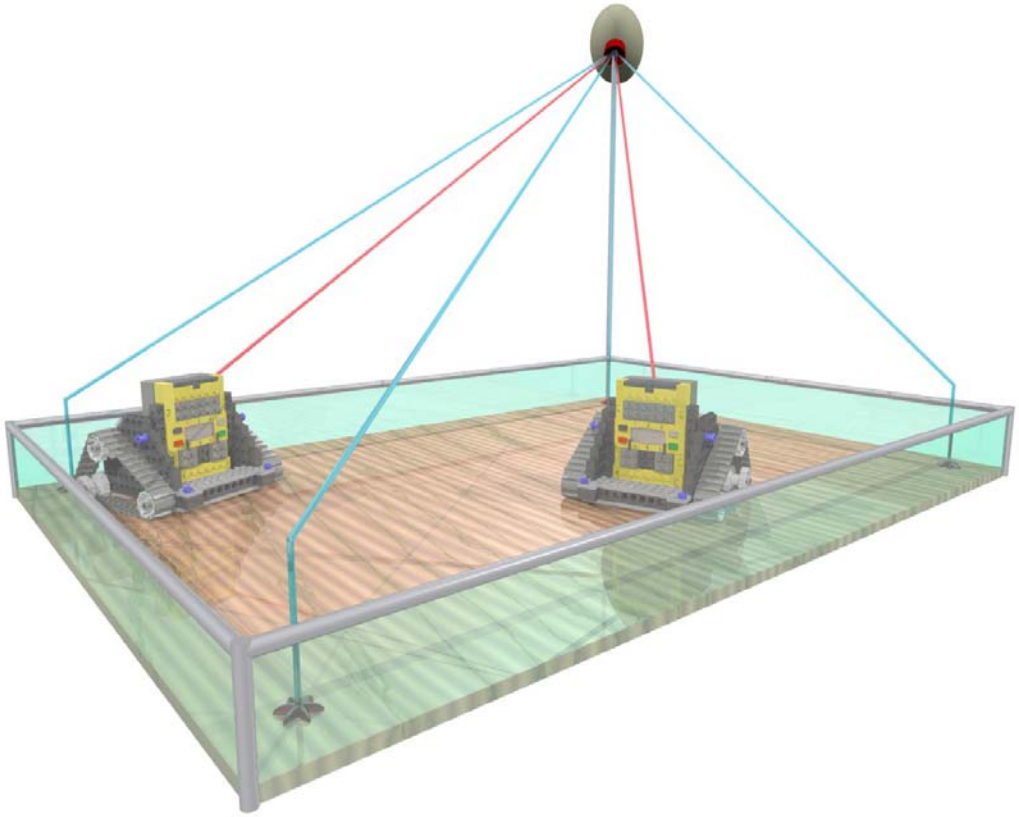
Remote DCL

10.09.2002

Eine komponenten-basierte Umgebung zum Ausführen von Experimenten mit mobilen, eingebetteten Systemen (Robotern) über das Internet



- Stefan Henze, Michael Richter, Helge Issel, Kai Müller, Kai Köhne -

Inhalt

1 Überblick

1.1 Ziel

Ziel von Remote DCL war es, die Erstellung und kontrollierte Ausführung von Programmen für mobile Geräte als Experiment über das Internet zu ermöglichen. Dabei sollte das Experiment in die für DISCOURSE bereits bestehende Infrastruktur integriert werden.

Darüber hinausgehend sollten die im Rahmen dieses Projektes entworfenen und implementierten Software-Komponenten möglichst modular, universell und leicht anpassbar sein, um eine Vielzahl von verschiedenartigen Experimenten mit einer minimalen Anzahl von Änderungen zu ermöglichen. Insbesondere sollten die Komponenten die im Lastenheft (siehe Anhang) geforderten Funktionen und Leistungen bereitstellen.

1.2 Involvierte Projekte

Das Projekt fand als Semesterarbeit im Rahmen der Vorlesung „Komponentenprogrammierung und Middleware“ am Lehrstuhl für Betriebssysteme und Middleware im Hasso-Plattner-Institut für Softwaresystemtechnik statt. Daneben waren aber auch andere Projekte involviert:

1.2.1 DCL

Das Distributed Control Lab (http://www.dcl.hpi.uni-potsdam.de) ist am Lehrstuhl für Betriebssysteme und Middleware angesiedelt und beschäftigt sich mit Software-Paradigmen und Entwurfsmustern, die eine Verbindung von Middleware-basierten Komponenten mit eingebetteten (mobilen) Steuerungssystemen ermöglichen. Im Mittelpunkt des Interesses steht dabei die Frage, wie vorhersagbares Systemverhalten (Zeitverhalten, Fehlertoleranz, Ressourcenverbrauch – CPU/Speicher/Energie – und Sicherheit) gewährleistet werden kann.

Die Bewertung der verschiedenen Ansätze erfolgt anhand von Fallstudien – Steuerungsszenarien, die im DCL aufgebaut werden. Die im Rahmen von Remote DCL entwickelten Komponenten sollen bei diesen Experimenten eingesetzt werden.

1.2.2 DISCOURSE

DISCOURSE (DIStributed & COllaborative University Research & Study Environment, http://www.discourse.de) ist eine Art verteiltes Labor für verteiltes Rechnen mit fortgeschrittener Middleware-Technologie, das an vier Universitäten (der Freien Universität Berlin, der Technischen Universität Berlin, der Humboldt Universität Berlin und dem Hasso-Plattner-Institut der Universität Potsdam) angesiedelt ist und als Testumgebung für die Forschung sowie als Referenzplattform für die Lehre dienen soll. Dieses Projekt bildet sozusagen den "größeren Rahmen“ von Remote DCL.

1.3 Umgebung

Schon vor dem Beginn des Projektes wurde im Rahmen des Distributed Control Lab mit mobilen Robotern und der Infrastruktur, um diese zu programmieren und mit diesen zu kommunizieren, gearbeitet. Auf diese Arbeite sollte bei Remote DCL aufgebaut und die schon vorhandene Soft- und Hardware soweit wie möglich genutzt werden:

1.3.1 RCX

Der RCX™ Microcomputer ist Teil des Lego Mindstorms Robotics Invention System. Er besitzt intern einen Hitachi H8/3292 Microcontroller, 512 Byte RAM sowie 3 Ausgänge für Aktoren (Motoren) und 3 Eingänge für Sensoren. Zur Kommunikation mit PCs/anderen RCX weist er außerdem noch eine Infrarot-Schnittstelle auf.

Er stellt bei den später vorgestellten Experimenten das mobile, zu programmierende Gerät dar.

1.3.2 legOS

LegOS ist ein freies Betriebssystem für den RCX, das es erlaubt, Programme als „native code“ direkt auf dem Microcontroller auszuführen. Es besteht aus einem Kernel als Ersatz für das ursprüngliche Betriebssystem des RCX und mehreren Hilfsprogrammen auf der PC-Seite, die es unter anderem erlauben, Programm für legOS zu kompilieren (makelx) sowie den Kernel (firmdl3) und einzelne, kompilierte Programme (dll) auf die RCX zu laden.

Als Cross-Compiler wird dabei der GCC (GNU C Compiler) eingesetzt, so dass dementsprechend auch C bzw. C++ als Programmiersprachen für den RCX zur Verfügung stehen. Weiter setzen diese Hilfsprogramme eine UNIX-Umgebung voraus, welche unter Windows z.B. durch Cygwin emulierbar ist.

Im Rahmen von Remote DCL wurden unter anderem Wrapper für diese Programme erstellt, um sie für andere .NET Komponenten ohne großen Aufwand nutzbar zu machen.

1.3.3 Cygwin

Die Cygwin-Umgebung ermöglicht es, UNIX-Programme - nach einer Neukompilierung - unter Microsoft Windows auszuführen. Cygwin besteht aus einer Bibliothek, die UNIX-Systemaufrufe unter Windows emuliert, sowie aus zahlreichen mit dieser Bibliothek nach Windows portierten klassischen UNIX-Programmen. Mittels Cygwin lassen sich insbesondere die legOS-Tools auch unter Windows benutzen.

1.3.4 LNPol

LegOS nutzt standardmäßig ein einfaches Packetformat namens LNP (Lego Network Protocol) zur Übermittelung von Daten über die Infrarotschnittstelle eines RCX. Diese LNP-Packete sind in ihren Eigenschaften in etwa vergleichbar mit UDP-Paketen: Sie bestehen aus einem Header mit Ziel- und Quelladresse und der Länge der gesamten Nachricht sowie aus dem Body mit den eigentlich zu übermittelnden Daten und einem Parity-Bit, das sicherstellt, dass einfache Fehler bei der Übertragung

erkannt werden. Es wird nicht sichergestellt, dass abgesendete Nachrichten den Empfänger erreichen.

LNPoI (Lego Network Protocol over the Internet) ist ein Software-Router für diese Pakete. Programme können sich über eine TCP-IP Verbindung anmelden, die Verbindung zu den RCX wird dagegen über Infrarot-Tower ermöglicht. Indem man nun den Programmen bzw. den Infrarot-Towern bestimmte Adressbereiche zuweist, ermöglicht man eine N:N Kommunikation zwischen allen angemeldeten Teilnehmern.

1.3.5 Web-Interface

Das Web-Interface mit integrierter Benutzerverwaltung stand uns bei Beginn des Projektes schon teilweise zur Verfügung. Im wesentlichen besteht es aus Active Server Pages und einem Controller, bei dem sich neue Experimente anmelden können. Das Web-Interface bietet dem Benutzer eine Liste aller angemeldeten Experimente sowie eine für jedes Experiment gleiche Eingabe-Maske.

Das Web-Interface verwaltet neben den Benutzern auch alle Experimente und reguliert den Zugriff auf diese. Die Kommunikation mit den Experimenten erfolgt über eine für alle Experimente gleiche Schnittstelle (siehe dazu das Kapitel 2.2).

1.4 Auswahl des Komponenten-Frameworks

Durch Vorgaben aus dem DISCOURSE-Projekt waren wir auf Microsoft .NET als Komponenten-Framework festgelegt. Eine nähere Erläuterung zu den Vor- und Nachteilen dieser Entscheidung findet sich im Kapitel „Alternative Komponenten-Frameworks“

2 Das System und seine Komponenten

2.1 Überblick über die Komponenten (Grobentwurf)

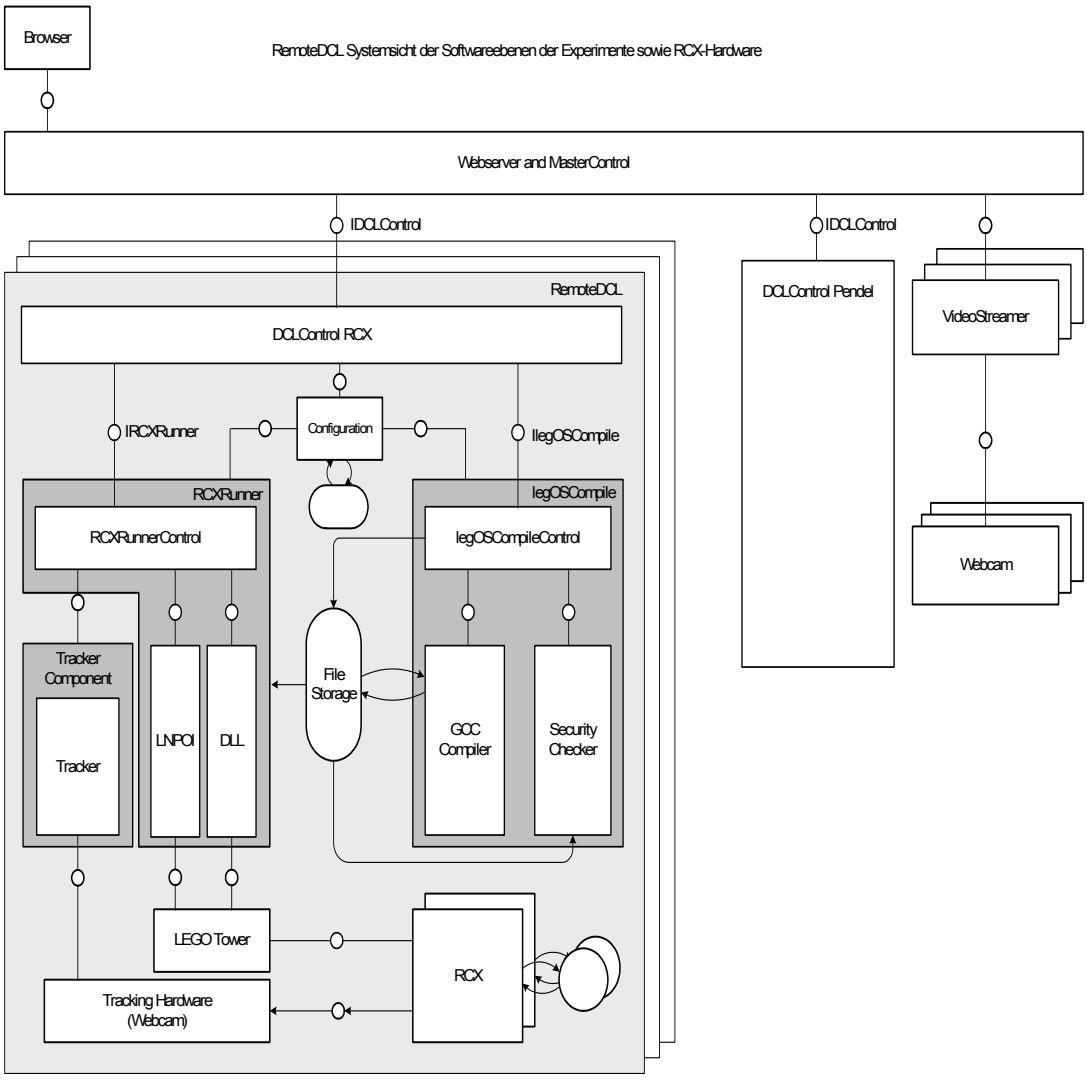


Abbildung 1: Aufbaubild des Gesamtsystems

In Abbildung 1 ist der Gesamtaufbau des Systems als Aufbaubild in FMC-Notation dargestellt: Die im Rahmen von Remote DCL erstellten Komponenten sind hellgrau hinterlegt.

Betrachtet man den Ablauf eines typischen Experiments, werden die Rollen und das Zusammenspiel der einzelnen Komponenten deutlich:

Der Nutzer meldet sich über den Webserver an und erhält einen Überblick über alle am MasterControl angemeldeten Experimente mit ihrem aktuellen Status. Läuft ein Experiment gerade, kann er nur passiv per Videostream den Verlauf des Experiments verfolgen.

Entscheidet er sich hingegen für ein freies Experiment, erhält er eine Maske, in der er typischerweise Programmcode eingeben kann, das dann im Verlaufe des Experimentes ausgeführt wird. Handelt es sich um ein mit Remote DCL erstellten Experimentaufbau, so wird dieser Quellcode vom Experiment-Controller („DCLControlRCX“) an den Compiler („LegOSCompileControl“) weitergeleitet. Dieser analysiert nun das Programm mittels des Security-Checkers auf für die Gesamtintegrität des Systems gefährliche Anweisungen: nur wenn keine gefährlichen Anweisungen gefunden wurden, wird das Programm per GCC kompiliert. Traten bei der Kompilation oder der Sicherheitsanalyse Fehler auf, so erhält der Benutzer eine detaillierte Fehlermeldung und kann sein Programm entsprechend korrigieren.

Ist dagegen alles erfolgreich verlaufen, kann der Benutzer das Experiment starten; daraufhin übergibt der Experiment-Controller das kompilierte Programm der RCXRunner-Komponente. Diese lädt mittels des legOS-Tools DLL das kompilierte Programm auf den RCX und startet mittels LNPoI die mobilen Roboter (es können ggf. mehrere RCX an dem Experiment beteiligt sein). Eine WebCam überträgt während des laufenden Experiments die Daten an die Tracker-Komponente, diese wertet die Bilder aus und errechnet daraus ständig die aktuellen Positionen der Roboter. Diese Positionsdaten werden wiederum ständig von der RCXRunnerControl abgerufen und auf Abbruchbedingungen, wie die Verletzung von Integritätsbedingungen oder das Erreichen des Ziels des Experiments, hin überprüft.

Das Experiment wird beendet, falls eine Abbruchbedingung erfüllt wird, der Timeout zuschlägt oder der Benutzer das Experiment abbricht. Beim Beenden des Experimentes stoppt die RCXRunnerControl-Komponente über LNPoI die Roboter, startet ggf. ein Programm, das die Roboter wieder in die Ausgangsposition zurückfährt und signalisiert dem Experiment-Controller das Ende des Experimentes, der diese Information an den Nutzer weiterleitet.

2.2 Experiment-Controller

2.2.1 Einleitung / Ziele

Für den Webserver, der den Zugriff auf die Experimente verwaltet, sind die Experimente lediglich remoting-fähige Objekte, die dem Interface „IDCLControl“ genügen. Da die Steuerung der Experimente durch den Benutzer ausschließlich über die Methoden dieses Interfaces erfolgen kann, stellt dieses Interface die Anwendungsschnittstelle dar.

Der Experiment-Controller ist innerhalb der Remote DCL-Komponenten also die „Schnittstelle zur Aussenwelt“ und hat damit die Aufgabe, den Ablauf des Experimentes zu steuern. Er muss die Methoden des „IDCLControl“-Interfaces implementieren und die Funktionalität des Experiments durch die Koordination der spezifischen Komponenten realisieren.

2.2.2 Entwurfsentscheidungen

Die Klasse ist verhältnismäßig einfach aufgebaut, da die implementierte Funktionalität lediglich im Verbinden von Objekten (bzw. Komponenten) besteht, die die Hauptfunktionalität des Experiments realisieren. Die aus dieser Kapselung des Ablaufes resultierende Modularisierung ermöglicht es, den Ablauf leichter zu erkennen und zu warten.

Es wäre denkbar, die An- und Abmeldung des Experiments am Server, das Auswählen der Konfiguration und ähnliche grundsätzliche Verwaltungsaufgaben ebenfalls in der Experiment-Controller Klasse zu integrieren, so dass diese ein in sich geschlossene Softwarepaket ergäbe.

Nach der Idee des „Separation of Concerns“ sollte allerdings wenig zusätzliche Funktionalität in den Experiment Controller integriert werden. Dementsprechend enthält der implementierte Experiment-Controller diese administrativen Aufgaben nicht, so dass die Anmeldung des Experiments sowie die Initialisierung der Konfiguration von separaten Klassen übernommen wird.

Dies brachte einige Vorteile: durch die resultierende geringe Komplexität der Klasse war eine sehr einfache Ablaufsteuerung möglich, was besonders das Testen erleichterte. Weiter wurden so auch die beiden Aspekte „Anmeldung“ und „Konfigurationsinitialisierung“ in einzelnen, abgeschlossenen Klassen gekapselt, die größere Flexibilität bieten. Die Art und Weise, wie die Anmeldung am Server durchgeführt wird, ist somit völlig unabhängig vom Experiment Controller. So ist z.B. auch die interaktive Auswahl der Konfiguration oder eines bestimmten Servers leicht realisierbar.

Außerdem ist die „Selbstanmeldung“ ohne Modifikation des Servers auch zur Zeit nicht möglich, da der Webserver ein „server-activated object“ erwartet. Ein solches Objekt wird angelegt, indem dem Hostrechner des Experiments mitgeteilt wird, dass er unter einer bestimmten URI ein Objekt einer bestimmten Klasse zur Verfügung stellen soll. Die entsprechende Instanz erzeugt der Host selbständig, dass heißt, es wird automatisch ein neues Objekt erzeugt.

Die eigentliche Initialisierung des Experiments wird im Konstruktor vorgenommen, da das „IDCLControl“-Interface keine spezielle Initialisierungsroutine vorsieht.

Eine wesentliche Fragestellung war die Art der Realisierung der GetStatus()-Methode. Theoretisch sollte Methode diese zu jedem denkbaren Zeitpunkt eine möglichst spezifische und aktuelle Auskunft über den Status des Experiments geben. Da diese Information jedoch auf Grund der Modularisierung häufig nicht im Experiment-Controller vorliegt, wird das Pattern „Chain of Responsibility“ angewandt, d.h. die eintreffenden Anfragen werden an das zum jeweiligen Zeitpunkt aktive Objekt weitergeleitet, welches diese ggf. seinerseits weiter delegiert, bis ein Objekt gefunden ist, das den aktuellen Status kennt.

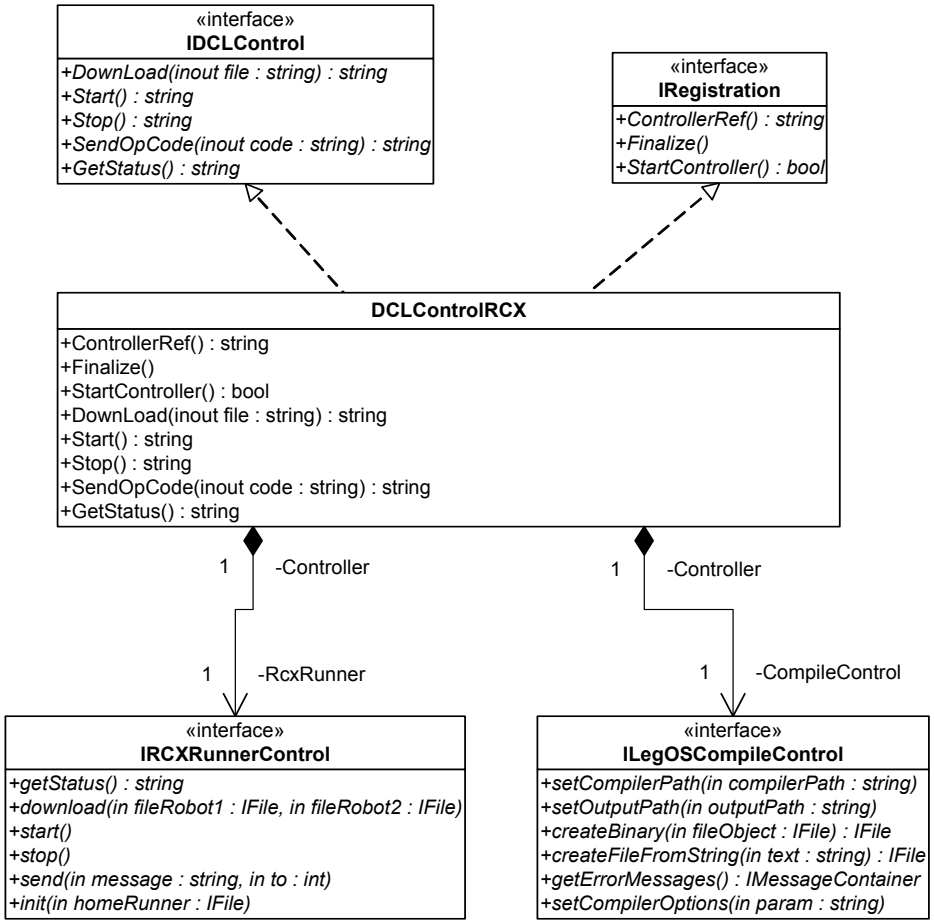


Abbildung 2: DCLControlRCX und seine Umgebung

2.2.3 Implementierung

Das „IDCLControl“-Interface erfordert die Implementierung folgender Methoden: „DownLoad()“, „Start()“, „Stop()“ und „GetStatus()“. Bei der vorliegenden Architektur hat der Experiment-Controller nur die Aufgabe, die entsprechenden Komponenten zur richtigen Zeit zu aktivieren. Aus diesem Grund besteht die Implementierung im wesentlichen aus der Weitergabe der Aufrufe an die Objekte, die die eigentliche Funktionalität implementieren. Dementsprechend wird im folgenden die Realisierung der in diesem Interface definierten Methoden kurz skizziert, um das Zusammenspiel der zentralen Komponenten zu beleuchten.

Die Methode „DownLoad()“ hat eine etwas andere Semantik enthalten als aus ihrem Wortlaut hervor geht, da das vom Benutzer eingegebene Programm nur kompiliert (und statisch analysiert) wird. Hintergrund dieser Entscheidung ist, dass der Transfer eines Programms auf den RCX erst erforderlich ist, wenn dieses wirklich ausgeführt werden soll, also spätestens in „Start()“. Durch diese Verlagerung in die Methode „Start()“ ist eine spätere Erweiterung des Experimentes denkbar, bei der beliebig viele Benutzer Programme schreiben und testweise kompilieren könnten, obwohl nur ein Benutzer Zugriff auf die RCX hat.

In „DownLoad()“ wird mittels des Objekts CompileControl aus dem beim Aufruf übergebenen String eine (Text-)Datei erzeugt, welche im zweiten Schritt in ein ausführbares RCX-Binary umgewandelt wird. Die resultierenden Rückmeldungen für den Benutzer werden aus dem zurückgegebenen Nachrichtencontainer („IMessageContainer“) ausgelesen und an den Webserver zurückgesendet (Details in Kapitel 2.3.2).

Die „Start()“-Methode lädt zuerst die entsprechenden Programme auf die Roboter, anschließend wird dann „RCXRunnerControl“ und damit das Experiment gestartet.

„Stop()“ leitet den entsprechenden Befehl an RCXRunnerControl weiter und stellt so sicher, dass Experiment ordnungsgemäß beendet wird.

In „GetStatus()“ wurden „Delegates“ als C#-Charakteristika zur Implementierung der „Chain of Responsiblity“ verwendet. „Delegates“ sind Objekte, die jeweils einen frei definierbaren Methodenaufruf kapseln. Die in der Signatur der aufzurufenden Methoden definierten Liste der formalen Parameter muss dabei der des Delegate-Typs entsprechen, nicht jedoch die Methodenbezeichnung. Somit stellen Delegates eine Art objektorientierten, typsicheren Funktionszeiger dar. Gegenüber dem sonst üblichen Ansatz, ein „Listener“-Interface zu definieren, ist vorteilhaft, dass kein Interface mehr explizit implementiert werden muss, wodurch sich die Integration von nicht explizit für diese „Chain“ entworfenen Klassen vereinfacht.

2.3 Compiler

2.3.1 Einleitung / Ziele

Um ein Programm auf den RCX ausführen zu können, muss es in entsprechenden Assemblercode umgewandelt werden. Die Entwicklung eines eigenen Compilers (und ggf. eines eigenen Betriebssystems für den RCX) hätte nicht nur den Rahmen dieses Projektes bei weitem gesprengt, vielmehr sicherlich auch seinem Charakter widersprochen. Die Evaluation der verfügbaren Möglichkeiten ergab, dass das legOS Betriebssystem mit seiner Unterstützung für C und den GCC-Compiler am flexibelsten ist. Somit entschieden wir uns, als erste Stufe das Programmieren der Roboter in C zu ermöglichen. Dementsprechend sollte der Zugriff auf den GCC-Compiler so gekapselt werden, dass er wie eine „normale“ .NET-Komponente ansprechbar sein würde. Selbstverständlich sollte die konkrete implementierende Klasse die abstrakten Methoden eines allgemeinen, nicht auf einen speziellen Compiler zugeschnittenen Interfaces realisieren. Somit sollte eigentlich eine leichte Einbindung anderer Compiler gewährleistet sein.

Die Benutzung von legOS als Betriebssystem in Verbindung mit der Programmiersprache C wirft allerdings einige Sicherheitsprobleme auf, deren Minimierung bei der Architektur des Systems angestrebt wurde. Das wesentlichste Problem besteht darin, dass die Integrität des LegOS wegen des nicht vorhandenen Speicherschutzes durch C Programme, die Pointeroperationen enthalten, kompromittiert werden könnte. Um die resultierenden Sicherheitsprobleme zumindest einzuschränken, wird vor Programmausführung durch den „LegOSSecurityChecker“ eine statische Analyse des User-Programms auf C-Quellcode-Ebene durchgeführt.

2.3.2 Entwurfsentscheidungen

Das mittelfristige Ziel besteht darin, die jetzt verwendete Programmiersprache C durch eine Sprache zu ersetzen, die sicherer ist. Dementsprechend wird die statische Analyse des Quellcodes dann nicht mehr notwendig sein. Der spezifische Ablauf der Umwandlung von Quell- in Zielcode (ggf. mit einem oder mehreren statischen Sicherheitsüberprüfungen) wird deshalb von der Klasse „LegOSCompileControl“ gekapselt. Sie enthält wiederum das eigentliche Compiler-Objekt vom Typ „LegOSCompiler“, das den GCC-Compiler kapselt sowie für die Sicherheitsprüfung eine Instanz der Klasse „LegOSSecurityChecker“.

Der Umstieg auf eine andere Programmiersprache oder die Änderung der Sicherheitsanalyse erfordert somit keine Änderung an anderen Komponenten.

Um die Allgemeinheit der einzelnen Klassen – und damit der Komponente – zu wahren, wurde ein spezifisches Nachrichteninterface samt dazugehöriger Nachrichtentransportklasse spezifiziert: ein Nachrichtentransportobjekt (implementiert IMessageContainer) enthält Null oder mehr Objekte, die IMessage implementieren. Jedes dieser Objekte enthält einen Erklärungstext sowie ggf. Zeilen und Spaltennummern, die für die Nachricht relevant sind.

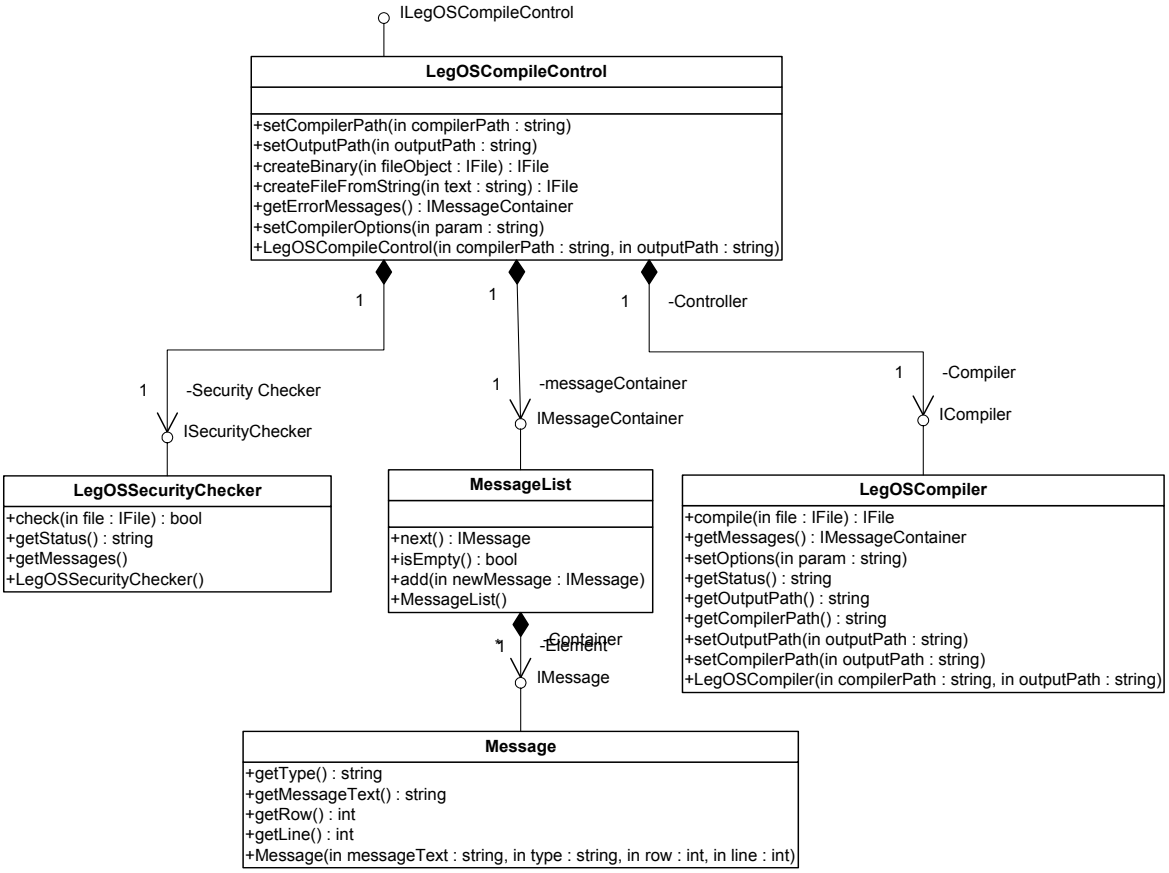


Abbildung 4: Die LegOSCompileControl Komponente

Durch diese Codierung werden alle Nachrichten in einem standardisierten Format an den Aufrufer zurückgegeben, so dass sichergestellt ist, dass compiler-spezifische Codierungen von der kapselnden Klasse abgefangen werden.

Des Weiteren wurde auch ein Interface „IFile“ spezifiziert, das Objekte, die Informationen über eine Datei kapseln, implementieren müssen. Im wesentlichen erlaubt dieses Interface Zugriff auf den Dateityp, Dateinamen sowie den Sicherheitsstatus (erfolgreich geprüft/nicht geprüft). Der Dateityp, der unabhängig von der Dateiendung o.ä. gesetzt werden kann, ist typensicher als Enumeration definiert. Im Zusammenspiel mit dem Sicherheitsattribut kann so leicht sichergestellt werden, dass nur geprüfte LegOS-Binaries auf die RCX geladen werden können.

2.3.3 Implementierung

Die Ansteuerung des Compilers wird in der Klasse „LegOSCompiler“ gekapselt, die den Compiler, der ein externes Programm ist, als separaten Prozess startet. Die entsprechenden Parameter werden dem (externen) Compiler bereits beim Programmstart übergeben, seine Rückmeldungen über die Umleitung von „stdout“ und „stderror“ ausgelesen. In C# ist die gesamte Funktionalität der Win32-API in Bezug auf die Arbeit mit Prozessen in den beiden Klassen „Process“ und „ProcessStartInfo“ gekapselt, was das Prozesshandling deutlich komfortabler gestaltet als die direkte API-Benutzung unter C.

Um die Compilerklasse möglichst flexibel nutzen zu können werden alle Pfade sowie die compiler-spezifischen Optionen aus der Konfiguration ausgelesen.

Da das Erzeugen von Binaries für RCX aus C-Quelldateien ein mehrstufiger Prozess ist, wird statt des Compilers „make“ aufgerufen, welches ein entsprechendes (statisches) makefile verwendet. Diese Konstruktion erlaubt es, den Erstellungsprozess (und damit auch den Compiler) zu ändern, ohne den Komponentenquellcode ändern zu müssen.

Die Kompilation wird immer im Pfad des Compilers ausgeführt, anschließend wird die entstandene Binärdatei in das Ausgabeverzeichnis verschoben. Dieser Entscheidung liegt die Überlegung zu Grunde, dass für das Quellverzeichnis unter Umständen nur Leserechte existieren, während der Zugriff auf das Zielverzeichnis eventuell (z.B. bei Netzwerken) sehr langsam sein kann.

Die Sicherheitsroutine, implementiert in der Klasse „SecurityChecker“, verhindert das Einbinden von Include-Dateien und das Verwenden von Zeigern, um den Zugriff auf LegOS-Betriebsystemstrukturen zu verhindern.

Die Meldungen des Compilations- und Analyseprozesses werden in einem Objekt, das IMessageContainer implementiert, an LegOSCompileControl zurückgegeben, dass dann die relevanten Meldungen weiterleitet. Ist z.B. die Sicherheitsanalyse erfolgreich, so werden nur noch Compilermeldungen angezeigt.

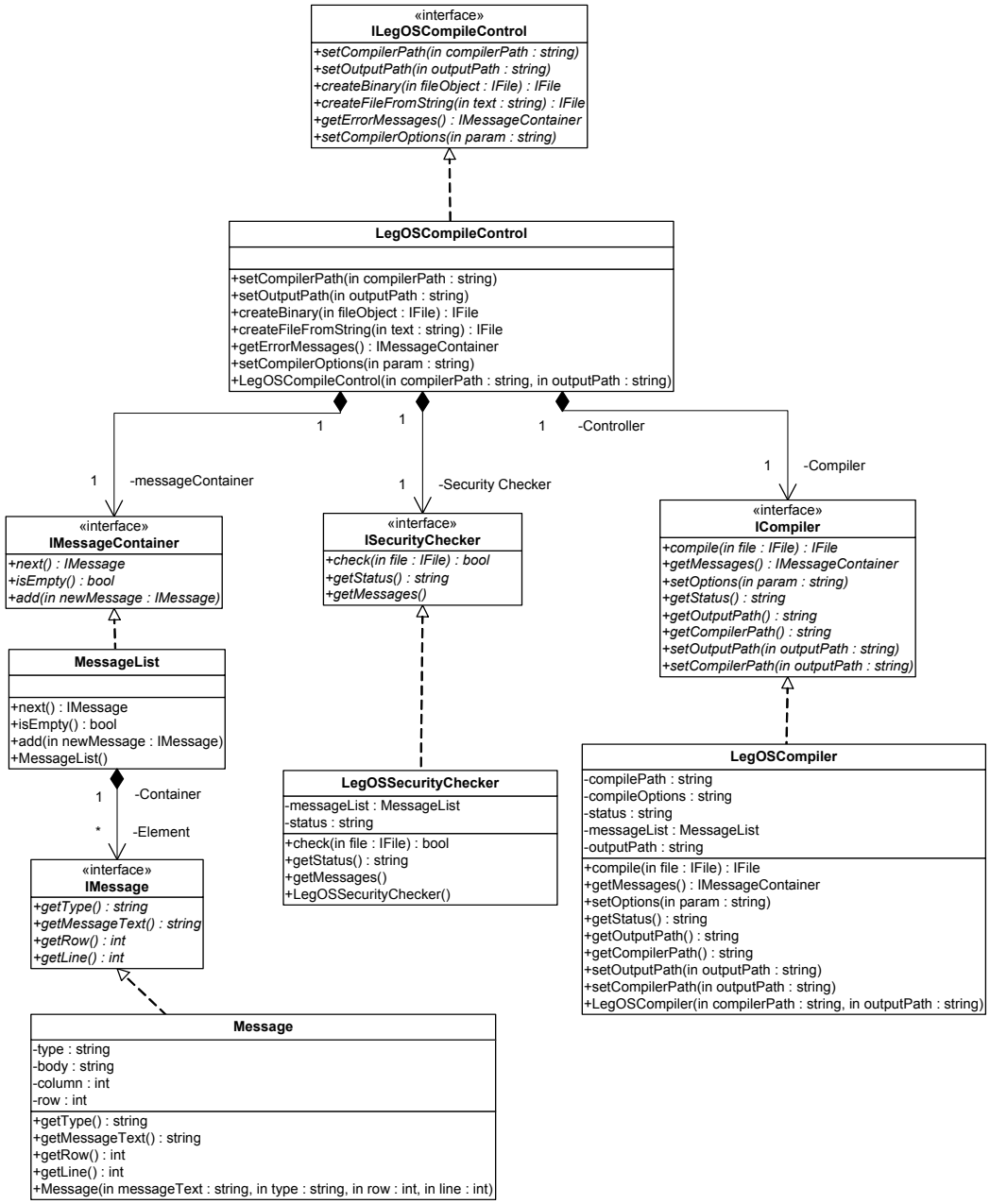


Abbildung 5: LegOSCompileControl Komponente und verwendete Interfaces

2.4 RCX-Runner

2.4.1 Einleitung / Ziele

Die RCX-Runner-Komponente dient zur Steuerung und Überwachung des aktiven Experimentes. Sie bekommt Anweisungen vom Experiment-Controller und gibt Befehle an die RCX-Roboter, erhält, verarbeitet und sendet Tracking-Informationen und stellt die Integrität des Experiments sicher.

Das Wissen um die Position der Roboter ist dabei aus verschiedenen Gründen wichtig. Zum einen lassen sich User-Programme für das Experiment leichter entwickeln, wenn die laufende Position der Roboter bekannt ist, zum anderen lässt sich über die Position ermitteln, ob kritische Situationen eintreten oder eine Endsituation erreicht wurde. Zudem ist ein Zurückführen des Experimentaufbaus in den Anfangszustand leicht möglich, da die beteiligten Roboter lediglich definierte Positionen anfahren müssen.

2.4.2 Entwurfsentscheidungen

Die Aufgabe des RCX-Runners besteht im wesentlichen aus zwei Teilen: Die ständige Ermittlung der Positionen der Roboter und die Überwachung des Experiments bezüglich einiger Integritäts- und Abbruchsbedingungen.

2.4.2.1 Positionsbestimmung

Grundsätzlich sind verschiedene Methoden der Positionsbestimmung von mobilen Robotern denkbar. Möglich sind z.B. Kontakte oder andere Sensoren im Boden oder die relative Positionsbestimmung ausgehend von einem definierten Punkt z.B. mit einer Computermaus. Hier wird hingegen das Prinzip der absoluten Positionsbestimmung mit visueller Identifikation der Objekte benutzt. Diese hat den Vorteil, dass sich etwa Ungenauigkeiten in der Messung (im Gegensatz zur relativen Positionsbestimmung) nicht addieren können; weiter sind die Anforderungen an Hardware gering, es wird nur eine handelsübliche Webcam benötigt.

2.4.2.2 visuelle Positionsbestimmung – theoretische Überlegungen

Die Umgebung, in der das Experiment stattfindet, ist eine Ebene, die von einem Punkt aus komplett überblickt werden kann. Es ist somit möglich, die Positionen aller interessierenden Objekte mithilfe einer einzigen Kamera zu bestimmen. Da je nach Position der Kamera im Verhältnis zur „Welt“ der Roboter aber Verzerrungen auftreten, ist eine Bestimmung der absoluten Koordinaten nur durch eine Entzerrung der durch die Kamera ermittelten Koordinaten (Kamerakoordinaten) möglich.

Kamerakoordinaten und absolute Koordinaten sind im Allgemeinen nicht identisch, aber eindeutig auf einander abbildbar. Vernachlässigt man den „Fischaugen-Effekt“ der Kamera, so lässt sich eine lineare Transformation anwenden, um die Koordinaten umzurechnen. Diese lineare Transformation soll nun im Folgenden hergeleitet werden:

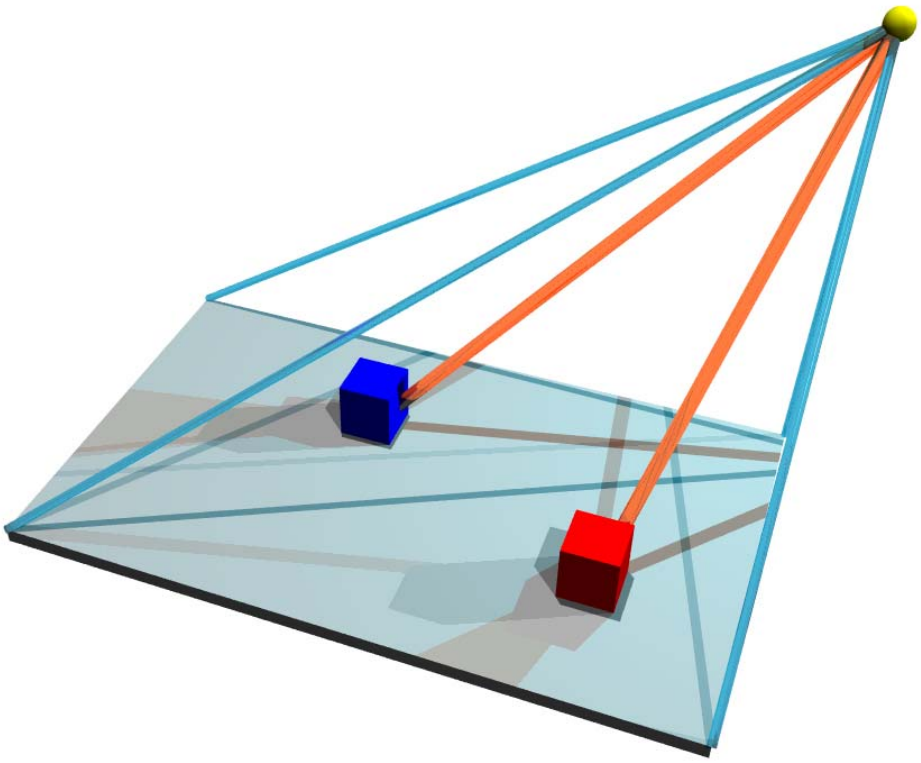


Abbildung 6: Aufbau eines Experiments

Ein grundsätzlicher Aufbau des Experimentes mit zwei Robotern ist in Abbildung 6 zu sehen; die Kamera kann in einer beliebigen Position zur Welt aufgestellt werden. In diesem Beispiel ist das in der Nähe einer der Ecken der Welt. Das resultierende Kamerabild könnte dann aussehen, wie in Abbildung 7 dargestellt.

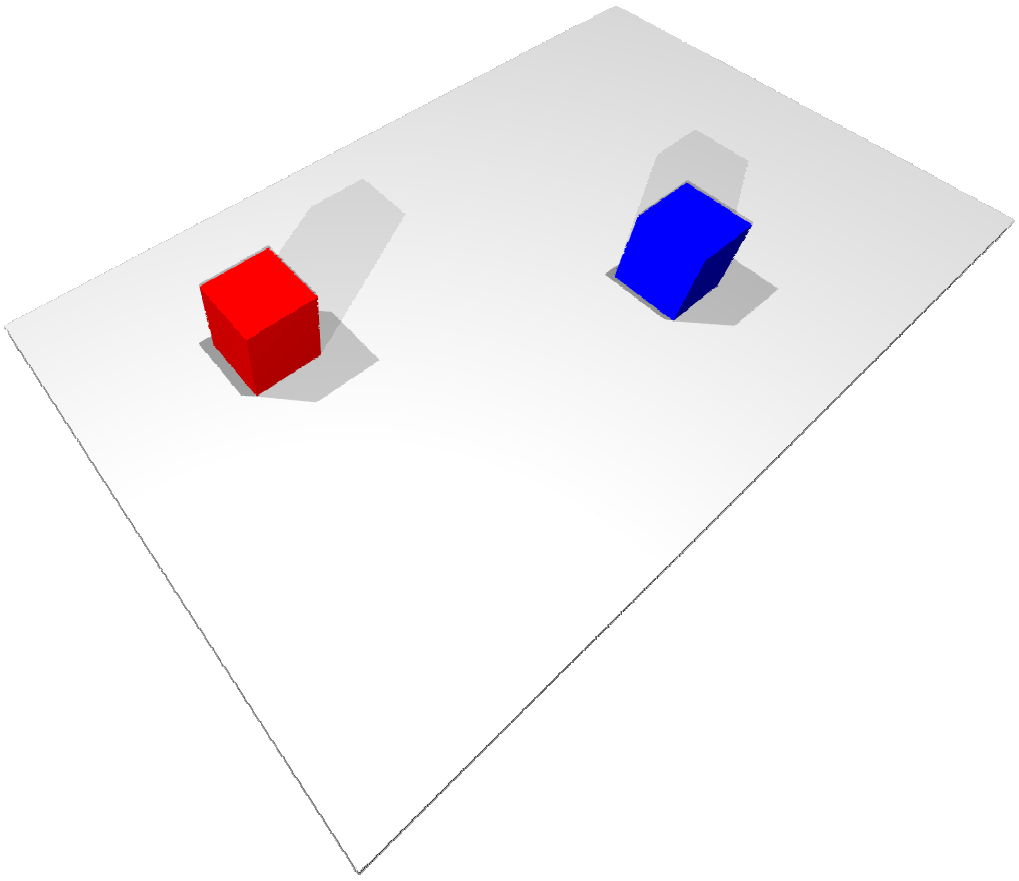


Abbildung 7: Sicht der Kamera auf das Experiment

Das Abbild der Realität ist also im Allgemeinen verzerrt und verdreht. Dies gilt es nun auszugleichen. Zunächst werden hierfür die Koordinaten in die (0,1)-Ebene transformiert. Eine Ecke wird also durch die Koordinate (0,0) beschrieben, die diagonal gegenüberliegende durch (1,1). Die Transformation lässt sich anhand einer Grafik (Abbildung 8) veranschaulichen.

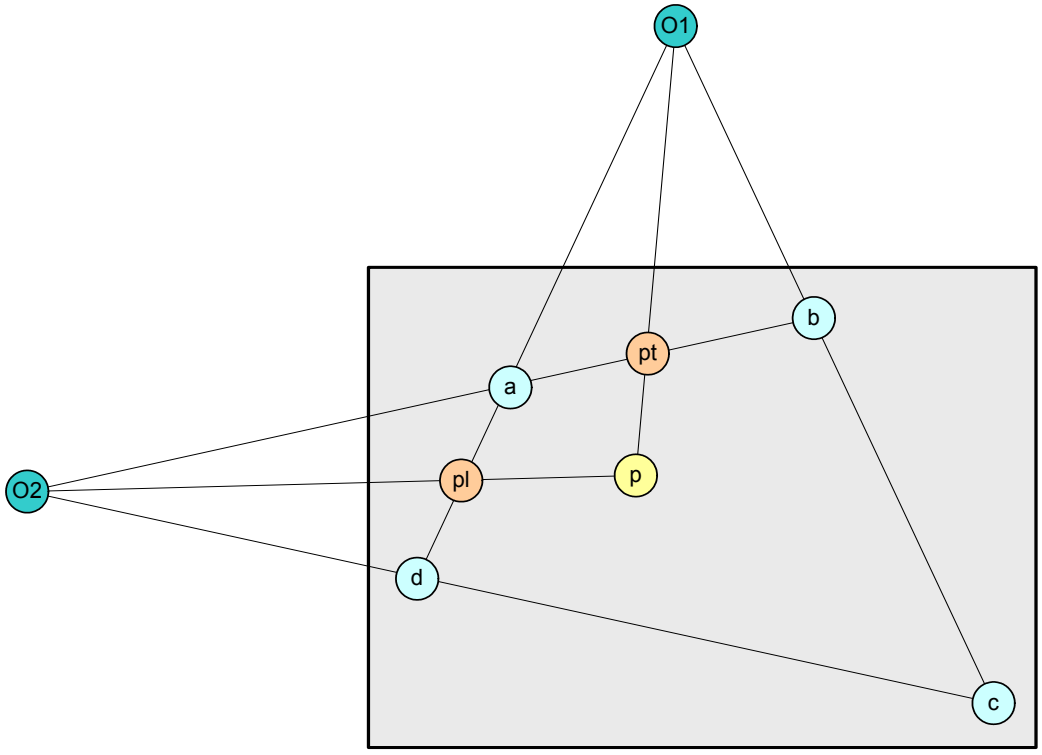


Abbildung 8: Beziehungen zwischen nichtnormalisierten und normalisierten Trackerdaten

Die Ecken der Welt seien durch die Punkte a, b, c und d beschrieben. Ein Punkt p in der Welt soll normalisiert werden. Die beiden Elemente der normalisierten Koordinate entsprechen den dargestellten Verhältnissen von (d,pl) / (d,a) und (a,pt) / (a,b). Die Punkte pl und pt werden als Schnittpunkte von (p,O1) und (a,b) bzw. (p,O2) und (d,a) ermittelt und die Punkte O1 und O2 sind Schnittpunkte der Gerade (d,a) und (c,b) bzw. (a,b) und (d,c).

2.4.2.3 visuelle Positionsbestimmung – Entwurfsentscheidungen

Es war von Beginn an des Projektes klar, dass wir einen Algorithmus für die visuelle Objekterkennung nicht selbst entwickeln bzw. implementieren konnten. Nach der Evaluation von einigen Möglichkeiten entschieden wir uns für den CamShift-Algorithmus der kostenlos im Internet verfügbaren Intel OpenCV-Bibliothek (http://www.intel.com/research/mrl/research/opencv).

Beim CamShift-Algorithmus wird eine Farbe bestimmt und im Bild eine rechteckige Fläche gesucht, die mit dieser (oder ähnlichen) Farbe gefüllt ist. Von nun an wird versucht, diese einmal bestimmte Farbfläche in jedem Kamerabild neu zu identifizieren: In Betracht bezogen wird dabei die Position des Objektes, sowie die Größe der Fläche im letzten Bild, so dass sich dieser Algorithmus insbesondere auch für Tracking von langsam bewegenden Objekten eignet. Eine genauere Beschreibung findet sich in der OpenCV-Referenz (siehe Literaturverzeichnis).

Bei der OpenCV-Bibliothek war schon eine Beispielanwendung des Camshift-Algorithmus mitsamt Quellcode beigelegt, bei der der Algorithmus in einem DirectDraw-Filter implementiert ist, der in eine DirectX-Filterkette hinter das

Kamerabild eingehängt wird: Der DirectDraw-Filter zeichnet dabei das getrackte Rechteck direkt in das Kamerabild ein. Ein Konfigurationsprogramm dient dazu, die Filterkette aufzubauen und das Kamerabild anzuzeigen. Es ermöglicht auch, Einstellungen beim Filter vorzunehmen bzw. die zu trackende Fläche mit der Maus zu markieren.

Wir haben uns entschlossen, diese schon vorhandene Infrastruktur zu nutzen und den Filter bzw. das Konfigurationsprogramm nur entsprechend unseren Bedürfnissen zu erweitern: Insbesondere hofften wir dadurch, um einen allzutiefen Einstieg in die für uns unbekannte Windows- bzw. DirectX- (und damit COM-) Programmierung in C++ herumzukommen. Weiter ist das Prinzip der DirectDraw-Filter und des externen Konfigurators sehr flexibel: Durch den Einsatz von COM ist es z.B. möglich, dass die Kamera an einem Rechner hängt, und der Filter bzw. der Konfigurator auf zwei anderen Rechnern arbeiten. Auch ist der Aufbau höchst modular, und man kann z.B. zur Laufzeit Filter in die Filterkette einfügen und entfernen.

Allerdings mussten wir die Infrastruktur an einigen Punkten erweitern: Da wir auch Experimente mit zwei mobilen Robotern erlauben wollten, mussten wir auch zwei Flächen unabhängig voneinander tracken können; außerdem mussten wir die Trackingdaten ja noch normalisieren (sie dazu die theoretischen Überlegungen weiter oben). Und schließlich mussten die getrackten Daten noch dem Controllerteil der RCXRunner-Komponente zugänglich gemacht werden.

Um mehrere Flächen gleichzeitig zu tracken, lag es nahe, einfach mehrere CamShift-Filter in die Filterkette einzuhängen; dies erforderte einzig und allein Änderungen im Konfigurator. Auch die Normaliserung der Trackingdaten haben wir in den CamShift-Filtern implementiert. Um allerdings die Übermittlung der normalisierten Trackingdaten an den Controllerteil des RCXRunners zu gewährleisten, führten wir ein weiteres COM Objekt ein, den Tracking Data Server.

Die Idee dabei war, die Art des Trackens und die dabei verwendeten Programme möglichst unabhängig vom Rest des RCXRunners zu machen; dies war insofern wichtig, als es genügend alternative Ansätze für das Tracken von Objekten gibt, und wir deshalb diesen Teil möglichst leicht austauschbar machen wollten (zumal wir nicht wussten, ob die Zuverlässigkeit des CamShift-Algorithmus ausreichen würde). Weiter ist der Camshift-Algorithmus relativ rechenintensiv. Es sollte deshalb möglich sein, die Filter und den Konfigurator auf einem anderen Rechner als dem vom Controller benutzten, laufen zu lassen.

Der Tracking Data Server (TDS) ist ein COM-Objekt, dass automatisch vom Konfigurator gestartet wird; an ihm melden sich alle Filter an und schicken beständig die normalisierten Positionen samt einem für jeden Filter einmaligen Bezeichner an den TDS. Auf der Controller-Seite gibt es ein Objekt namens TrackerControl, dessen einzige Aufgabe es ist, immer wieder die aktuellen Daten für jeden Bezeichner vom TDS abzufragen, und sie an den Controller weiterzugeben.

Das Zusammenwirken der oben angesprochenen Komponenten lässt sich in der FMC-Modellierung wie folgt darstellen:

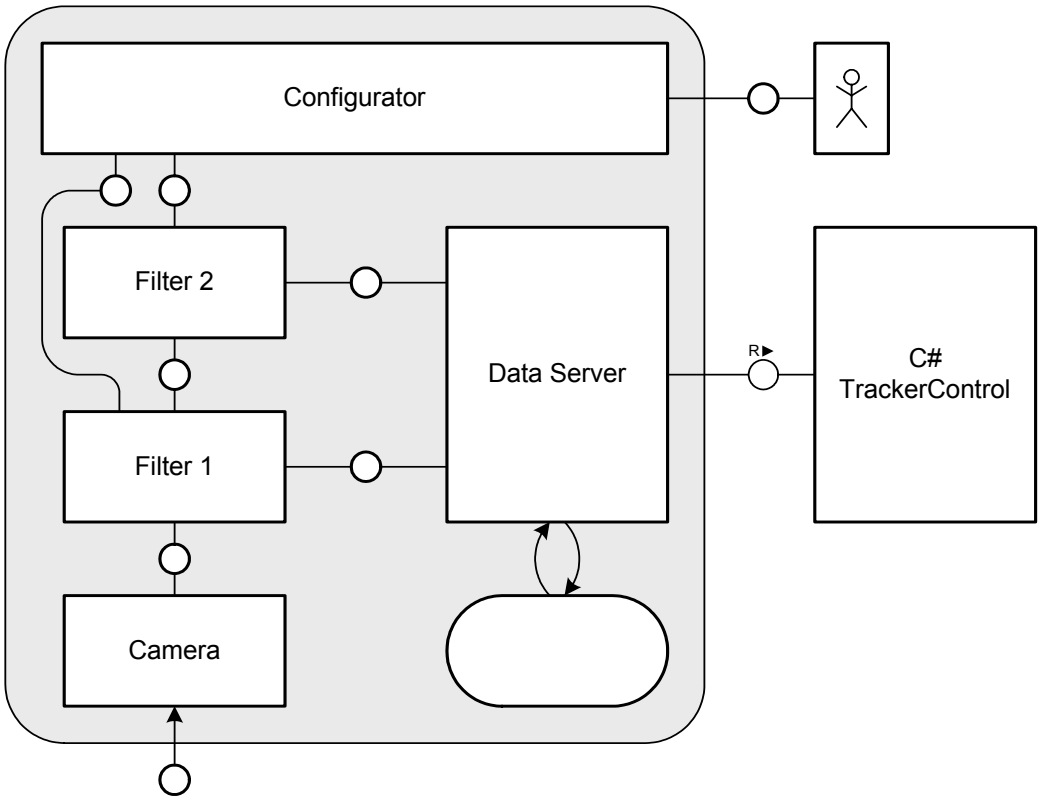


Abbildung 9: Aufbau des Trackers in FMC-Notation

2.4.2.4 Experiment-Kontrolle

Die Experiment-Kontrolle selbst läuft als Automat innerhalb von .NET ab (siehe Abbildung 11). Es wird jeweils die aktuelle Position der Roboter abgefragt, an die RCX übermittelt und schließlich die Situation analysiert. Liegt keine besondere Situation vor, d.h. keine kritische und keine Gewinn-Situation, wird erneut die Position abgefragt. Ist dagegen eine Gewinn-Situation oder eine kritische Situation eingetreten, so wird der Experiment-Controller benachrichtigt, dieser beendet im Folgenden das Experiment (beendet den Thread) und versetzt die Komponenten ggf. in den initialen Zustand zurück. Jetzt kann das Experiment erneut gestartet werden. Das Beenden des Experiments kann natürlich auch so zu jeder Zeit auftreten.

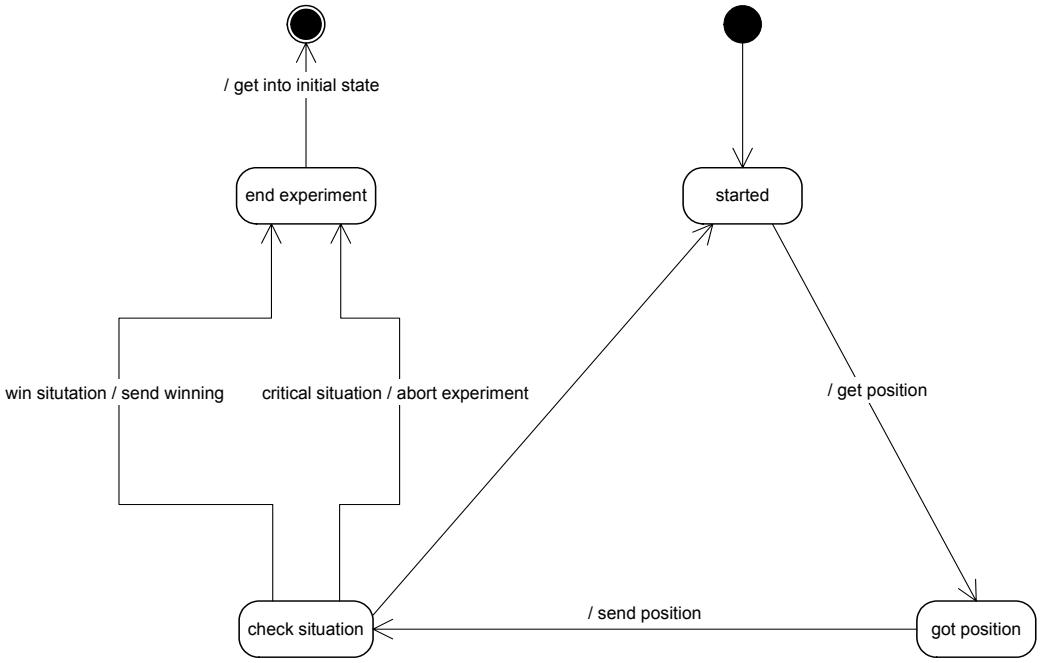


Abbildung 11: Steuerungsautomat

Besonders wichtig ist dabei die Sicherstellung der Rahmenbedingungen sowie die Rückführung des Experimentes in einen definierten Zustand: Es ist unbedingt notwendig, dass die Roboter nicht vom Tisch fallen oder gegen eine Wand fahren. Dies wird durch die einfache Überprüfung der Roboterkoordinaten erreicht: Diese müssen immer zwischen null und eins liegen, da dies die vom Konfigurator vorgegebene „Welt“ ist. Verlässt ein Roboter diesen Bereich (bzw. melden die Trackingdaten dies), wird das Experiment abgebrochen. Auch die Überprüfung auf eine Kollision der Roboter erfolgt analog: Fällt die Distanz der Roboter unter einen in der Konfiguration vorgegebenen Grenzwert, ist das Experiment ebenfalls beendet.

Des Weiteren müssen die Roboter nach Abbruch oder Beendigung eines Experiment-Durchlaufs wieder in den Ursprungszustand zurück versetzt werden. Dazu erhält jeder Roboter ein Programm, dass es ihm ermöglicht, an eine definierte Position „nach Hause“ zu fahren. Die aktuelle Position wird auch dabei ständig an die RCX gesendet, die daraufhin einen Weg zum Heimatpunkt beschreiten.

Da ebenfalls Winkel mit der Position übertragen werden, ist es auch möglich, nach Anfahren der Position, den Roboter in die Ausgangsrichtung zu drehen. Dies ist

allerdings aufgrund fehlender trigonometrischer Funktionen im legOS-Betriebssystem der Roboter zur Zeit noch nicht zuverlässig realisiert.

2.4.3 Implementierung

2.4.3.1 visuelle Positionsbestimmung

Dachten wir anfangs noch, wir könnten durch die Verwendung der Beispiele in der OpenCV-Library um einen tiefen Einstieg in die COM-Programmierung herumkommen, so stellte sich dies leider ziemlich bald als Irrtum heraus: Es kostete uns einiges an Zeit und Nerven, bis wir das oben beschriebene Zusammenspiel der Komponenten realisiert hatten. Insbesondere der Tracking Data Server machte uns Schwierigkeiten: Um zu erreichen, dass er sowohl von den Filtern als auch vom TrackerControl in C# problemlos ansprechbar war, mussten wir ihn als Singleton realisieren, was bedeutet, dass es von diesem Programm automatisch nur eine einzige Instanz auf einem Rechner gibt. Weiter gab es immer wieder Schwierigkeiten mit den Filtern, die nur durch einige Umbenennungen im ActiveX-Verzeichnisbaum behoben werden konnten.

Sehr einfach war hingegen das Ansprechen des TDS von C# aus: Dazu muss nur mittels der Type Library des TDS (die wir allerdings erst einmal erzeugen mussten) ein entsprechender C#-Stub erzeugt werden; dies geschieht mittels eines Kommando-Zeilen-Tools. Auch lassen sich in C# sehr komfortabel Komponenten realisieren, die nach aussen hin als COM-Objekt auftreten. Allerdings waren wir nicht in der Lage, den oben genannten TDS so zu realisieren, da uns die Möglichkeit des Singleton-Patterns unter C# fehlte.

Auch der tiefere Einstieg in die Windows Programmierung mittels C++ war zeitaufwendig, wir mussten schliesslich das Konfigurations-Programm noch erweitern. So wurde z.B. ermöglicht, die vier Eckpunkte der „Welt“ am Anfang des Experiments komfortabel mittels Maus zu markieren und danach die zu trackenden Flächen der Roboter zu selektieren. Einen Screenshot des geänderten Konfigurators sieht man in Abbildung 12.

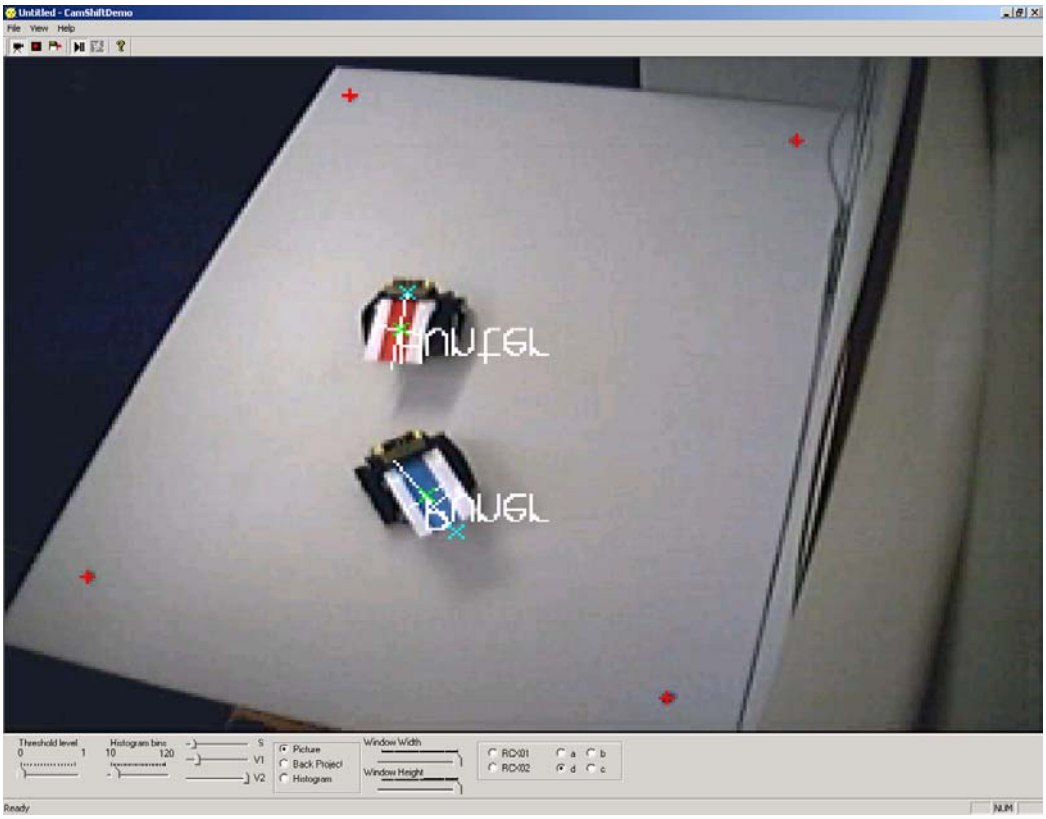


Abbildung 12: Screenshot des Konfigurators

2.4.3.2 Experiment-Kontrolle

Die Experiment-Kontrolle läuft innerhalb von .NET in einem eigenen Thread ab. RCXRunnerControl implementiert das Interface IRCXRunnerControl, das vom Experiment-Controller DCLControlRCX bedient wird. Es ist für eine Experimentumgebung mit zwei Robotern definiert, kann aber auch für einen Roboter verwendet werden.

An RCXRunnerControl sind die Komponenten LNPOI und DLL gekoppelt, die die Kommunikation mit den Robotern übernehmen, sowie die Tracking-Komponente, die ihre Daten über eine WebCam erhält.

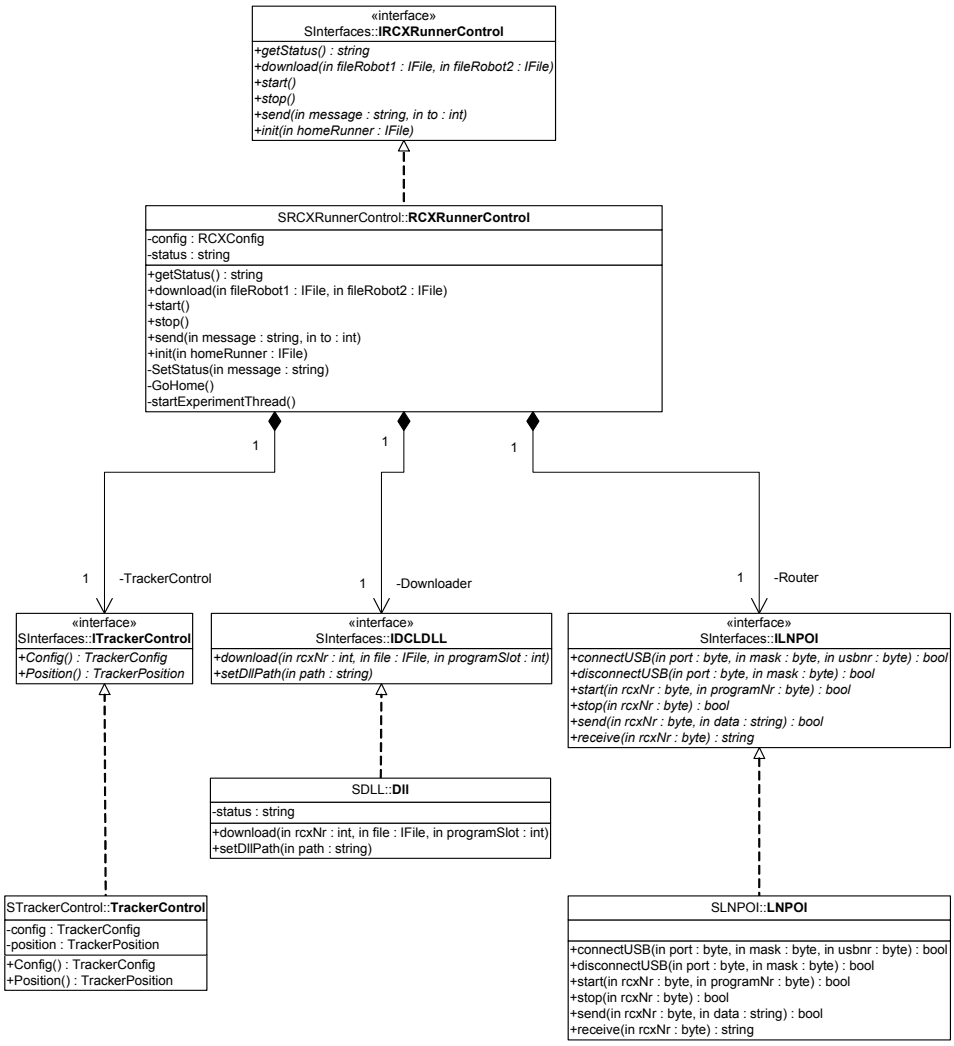


Abbildung 13: UML-Diagramm des RCX-Runner

Die Kommunikation mit den RCX erfolgt über den LEGO-Infrarot-Tower und – je nach Anwendung – mit der legOS-Download-Komponente (dll) oder dem LEGO Network Protocol Over Internet Router (LNPoI). Die beiden Komponenten werden über die Interfaces IDCLDLL und ILNPoI von RCXRunnerControl angesprochen.

Diesen Komonenten liegen eigenständige Programme zugrunde: Beide sind allerdings unter Windows nur in der Cygwin-Umgebung lauffähig und müssen zudem innerhalb einer Cygwin-Bash gestartet werden. Dadurch ist auch die Abfrage von Rückgabeparametern nur über Umwege möglich.

Bei LNPoI hingegen ergab sich noch als weiteres Problem, dass für die Kommunikation mit dem Programm eine TCP/IP Verbindung aufgebaut werden muss, über die dann im Rahmen eines Protokolls Anweisungen an das Programm

gegeben werden können. Es existierte für diese Kommunikation allerdings schon eine C++-Klasse, die es einem ermöglichte, die Feinheiten des Protokolls zu ignorieren.

Zur Integration dieser Klasse in .NET gab es zwei Möglichkeiten: Sie durch Änderungen im Quellcode zu einer managed-C++ Klasse zu machen, oder wiederum eine Wrapper-Klasse zu schreiben, die so den Übergang von managed (.NET) Code zu unmanaged (C++) Code ermöglicht. Wir entschieden uns für letzteres, um flexibel etwa bei möglichen Weiterentwicklungen von LNPoI zu bleiben.

Eine Herausforderung war dabei aber noch das Verbinden der unterschiedlichen Arbeitsweisen in C(++) und C#. So verlangte die C++ Klasse etwa, das die Daten ihr schon in einem einzigen Bytestream codiert übergeben werden, C# unterstützt allerdings die Manipulation von einzelnen Bytes eher schlecht. Auch mussten wir erst herausfinden, wie wir die von C++ verlangten Pointer unter C# erzeugen konnten.

2.5 Konfiguration

2.5.1 Einleitung / Ziele

Ein Ziel des Projektes war es, eine Art Bausteinsystem zu entwickeln, dessen Bestandteile als Basis für die Realisierung weiterer Experimente dienen können. Dementsprechend wichtig war uns auch eine flexible und komfortable Konfiguration der Komponenten.

Es ist im allgemeinen günstiger, Daten zur Laufzeit aus Konfigurationsdateien oder ähnlichem zu gewinnen als Annahmen über das Einsatzumfeld der Komponenten zur Compilezeit zu treffen und statisch zu verankern. Die Verwaltung von Konfigurationsdaten ist also eine häufig wiederkehrende Aufgabe, die dementsprechend in einer separaten Komponente gekapselt werden sollte.

Diese Kapselung bietet eine ganze Reihe von Vorteilen. Der wichtigste Punkt ist sicherlich, dass der Datenzugriff zentral implementiert wird, wodurch eine Entkopplung von der logischen Repräsentation der Daten für die Komponenten und der physischen Speicherung gewährleistet ist. Dementsprechend vereinfacht sich die Entwicklung der Komponenten erheblich, da für die Arbeit mit Konfigurationsdaten ein High-Level Interface zur Verfügung steht. Außerdem ist so – für die Komponenten völlig transparent – die Art der Speicherung der Daten variierbar; es sind sowohl verschiedene Dateiformate denkbar als auch eine dynamische Generierung der Daten zur Laufzeit (vor der ersten Nutzung). Schließlich erlaubt eine zentrale Komponente auch den Einsatz einer zentralen Konfigurations-„Datenbank“ (z.B. eine XML-Datei), was für den Benutzer die Konfiguration deutlich erleichtert.

Um die Unabhängigkeit der Komponenten zu gewährleisten, wollten wir ausserdem verschiedene Namensräume mit beliebigen Namen und beliebiger Komplexität realisiert werden. Eine weitere wesentliche Anforderung war, die Datenintegrität für die jeweiligen Komponenten zu gewährleisten, also sicherzustellen, dass Daten, die von Komponente A angefordert werden, nicht durch eine Komponente B absichtlich oder unabsichtlich verändert werden.

2.5.2 Entwurfsentscheidungen

Die Konfigurationsdaten werden in einer hierarchischen Baumstruktur abgelegt, wobei Komponenten auf einzelne Unterbäume, die direkt unter der Wurzel liegen, über deren Namen zugreifen können. Damit spannen diese „Top-Level“-Unterbäume also konzeptionell Namensräume auf, die dann im Allgemeinen alle für eine bestimmte Komponente notwendigen Daten enthalten. An dieser Stelle setzt auch die erste Stufe des „Information Hiding“ an, da nur auf namentliche bekannte Top-Level-Unterbäume zugegriffen werden kann und es keine Möglichkeit gibt, Namen direkt von der Konfigurationskomponente zu erfahren.

Fordert eine Komponente einen Namensraum (also einen „Top-Level“-Unterbaum) an, so erhält sie eine private Kopie des angeforderten Unterbaumes. Dadurch wird verhindert, dass Manipulationen der Datenstruktur Auswirkungen auf andere Komponenten haben.

Die Organisation der „Top-Level“-Unterbäume orientiert sich an XML: auf jeder Ebene können sich beliebig viele Knoten befinden, wobei jeder Knoten die

Wurzel eines neuen Baumes[1] darstellen kann. Gleichzeitig kann er über eine beliebige Anzahl von Attributen verfügen sowie einen Wert besitzen, z.B. eine Zeichenkette. Die Namen der Knoten müssen nicht eindeutig sein, die Bezeichner der Attribute hingegen schon. Knoten können sowohl über Traversierung des Baumes erreicht werden (mittels „nxtSilbing()“ und „down()“/“up()“), als auch direkt über ihren Namen, wobei bei letzterer Methode zu beachten ist, dass unter Umständen mehrere Knoten mit gleichem Namen existieren.

Das Konfigurationsobjekt (Klasse: „Configuration“)[2] ist ein Singleton, um einen einfachen, globalen Zugriff zu ermöglichen. Eine explizite Übergabe des Konfigurationsobjektes ist unnötig, da durch die verschiedenen Namensräume alle Komponenten ein- und dasselbe Konfigurationsobjekt verwenden können und hätte außerdem bei verschachtelten Komponenten den Nachteil, dass die aufrufende Komponente ein manipuliertes Konfigurationsobjekt übergeben könnte.

Die Datenquelle, die die Konfigurationsdaten definiert, kann nur genau einmal festgelegt werden (write-once), um nachträgliches (böswilliges) „Umbiegen“ der Datenquelle zu verhindern.

Als Format für persistente Speicherung der Daten wurde XML gewählt, da eine 1:1 Abbildung der logischen Baumstruktur auf XML-Elemente möglich ist (schließlich orientierte sich die Organisation unserer Konfiguration ja an XML), XML leicht von Menschen lesbar ist und sich als allgemeiner Standard etabliert hat. Des Weiteren wird die Verarbeitung von XML bereits durch .NET-Basisklassen unterstützt.

[1] Composite-Pattern

[2] im weiteren wird „Konfigurationsobjekt“ als Synonym für „das Singelton-Objekt der Klasse Configuration“ verwendet

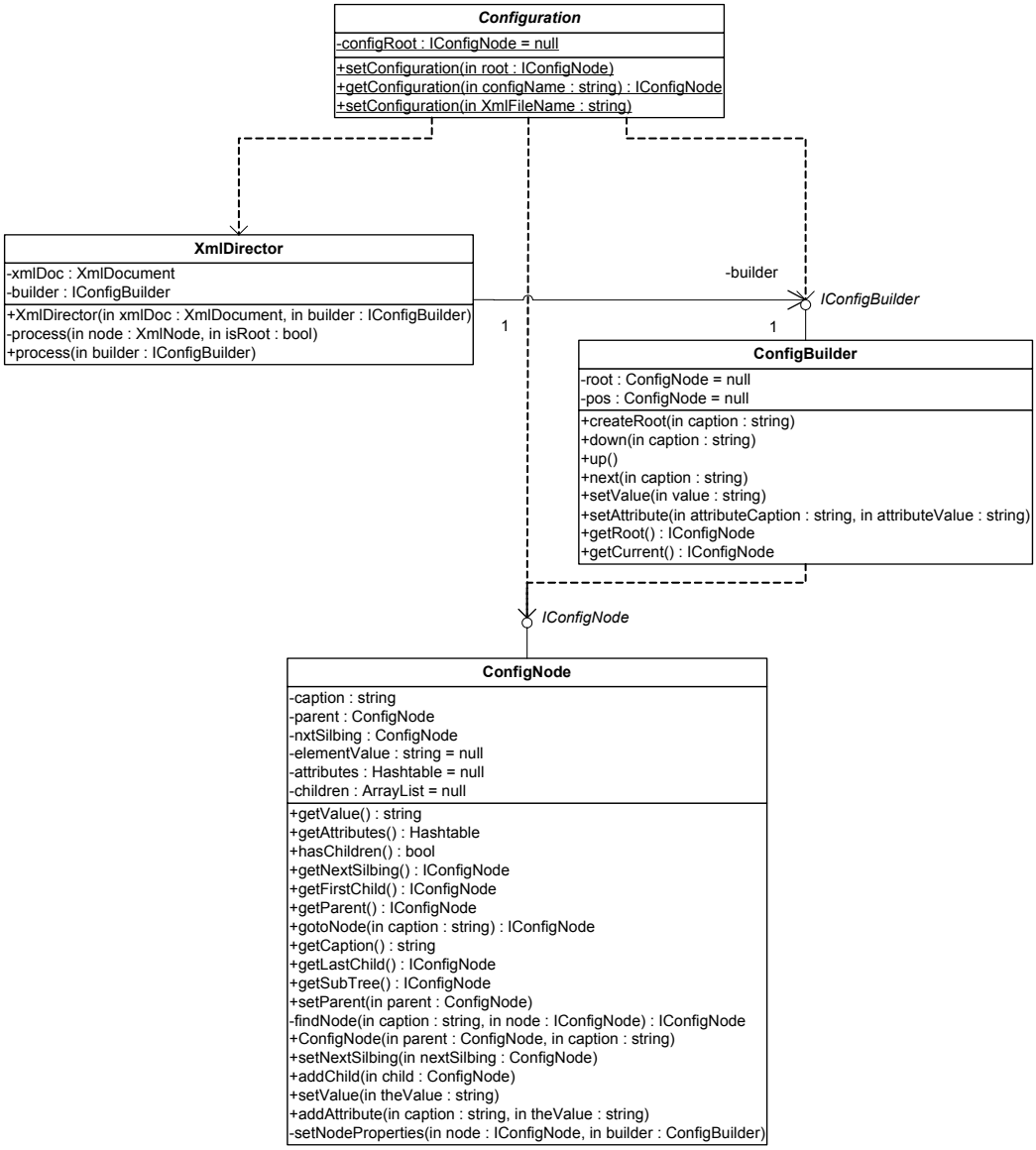


Abbildung 20: Konfigurationskomponente

2.5.3 Implementierung

Wie bereits geschildert, erhält ein Client vom Konfigurationsobjekt auf Anfrage einen Knoten, der die Wurzel eines Top-Level Unterbaumes[3] repräsentiert. Dieses Knotenobjekt stellt dann Methoden zur Verfügung, um zu anderen (Unter-)knoten zu gelangen, welche vom selben Typ sind, also die gleichen Methoden unterstützen.

Die Klasse „Configuration“ erlaubt es, entweder eine XML-Datei als Datenquelle anzugeben oder die Wurzel eines bereits aufgebauten Baumes.

[3] Genaugenommen erhält der Client nicht die Wurzel des Unterbaumes, sondern den ersten Unterknoten des Baumes. Dementsprechend ist der Zugriff auf die direkt unter der Top-Level-Unterbaumwurzel liegenden Knoten über „nxtSibling()“ möglich, da sich diese Knoten mit dem initial zurückgegebenen Knoten in einer Ebene befinden.

Für die Organisation der Klassen, die die Konstruktion des Konfigurationsbaumes realisieren, wurde das „Builder“-Pattern verwendet. Dementsprechend erfolgt der Aufbau des Baumes im Konfigurationsobjekt unter Zuhilfenahme einer Instanz von „XmlDirector“.

Die Konfigurationskomponente wurde also in drei Teile gegliedert: das Konfigurationsobjekt, welches die Festlegung der Datenquelle sowie die Anforderung eines Namensraumes erlaubt, die eigentliche Objektdatenstruktur implementiert in „ConfigNode“ sowie die Konstruktion der Objektdatenstruktur aus einer Datenquelle, die im Allgemeinen durch das Zusammenspiel von „XMLDirector“ und “ConfigBuilder“ realisiert wird.

Die Verwendung des Builder-Patterns ermöglicht es, die Implementierung der Knoten (Interface „IConfigNode“ und „IConfigBuilder") und die der Parser unabhängig voneinander zu variieren. Der Hauptnutzen dieser Flexibilität ist natürlich die verhältnismäßig einfache Erstellung von „Direktoren“ für neue Dateiformate.

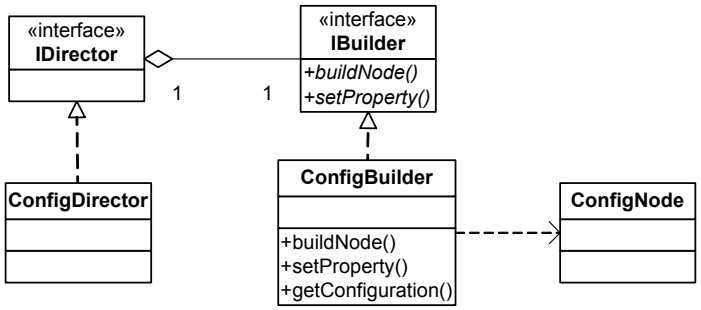


Abbildung 21: Builder-Pattern

Bei der Implementierung des „XmlDirectors“ wurde deutlich, dass die Basisklassen des .NET-Frameworks bereits eine breite Unterstützung des XML-Standards bieten. Insbesondere steht mit „XMLDocument“ bereits eine Klasse zur Verfügung, die ein XML-File in eine im Speicher befindliche Baumstruktur umwandelt (DOM-Modell). Dementsprechend traversiert der „XmlDirector“ lediglich diesen Baum und nutzt die „IBuilder"-Instanz, um einen analogen Baum zu konstruieren. Dementsprechend erscheint es im Nachhinein fraglich, ob sich der Aufwand für die Implementierung einer eigenen Baumstruktur lohnte. Es wäre wahrscheinlich günstiger gewesen, die vorhandenen Klassen zu nutzen. Zum Zeitpunkt der Implementierung erschien dies jedoch schwierig, da von den Knoten, die den XML-Baum bilden, lediglich bekannt ist, dass sie ein bestimmtes Interface realisieren. Damit ist zwar eine Erweiterung durch Vererbung nicht möglich, der Einsatz des „Bridge“-Patterns[4] sollte aber die Anpassung erlauben. Dies würde bedeuten, dass das Objekt, das der Client erhält, alle Methoden implementiert, in dem es die entsprechenden Methoden eines internen „XmlNode“-Objektes nutzt.

Angemerkt sei noch, dass das Konfigurationsobjekt nicht als Singleton im eigentlichen Sinn implementiert ist, sondern dass alle Methoden der Klasse „Configuration“ statisch sind. Dadurch ist natürlich die Bezeichnung „Konfigurationsobjekt“ irreführend.

[4] es lässt sich darüber streiten, ob „Decorator“-Pattern passender wäre

2.5.4 Nachteile des Konzepts

Die wesentlichste Schwäche der verwendeten Konfigurationskomponente liegt darin, dass die konsequente Trennung der Komponenten voneinander und die daraus resultierende einmalige Festlegung der Konfigurationsdaten jegliche dynamische Konfiguration verhindert. Dementsprechend werden im Kapitel „Ausblick/Erweiterungsmöglichkeiten“ einige Überlegungen angestellt, wie eine dynamische Konfiguration realisiert werden könne.

Desweiteren stellt die Verwendung von separaten privaten Kopien für jede Komponente natürlich einen Speicher-Overhead dar.

3 Das Runner-Hunter-Experiment

Das Runner-Hunter-Experiment sollte das erste mittels Remote DCL umgesetzte Experiment sein. Aufgrund von einigen Schwierigkeiten konnten wir es allerdings nur zum Teil realisieren. Dennoch stellte es den ersten „Belastungstest“ für die von uns entwickelten Komponenten dar und ließ uns erste Erfahrungen etwa bezüglich der Systemsicherheit machen, da diese auch von physikalischen Größen (etwa der Trägheit der LNP-Kommunikation) abhängig ist.

3.1 Idee

Der vom Benutzer programmierte Roboter „Hunter“ hat zum Ziel, einem zweiten Roboter, dem „Runner“ mit unbekanntem Programm, innerhalb einer gewissen Zeit möglichst nahe zu kommen[5]. Gelingt ihm dies, wird das Experiment abgebrochen und der Benutzer hat gewonnen.

Das Experiment stellt einen Härtetest dar, da es die Verletzung von Integritätsbedingungen (das wären in diesem Fall: die zwei Roboter kommen sich zu nahe bzw. mindestens einer von verlässt den definierten Bereich) geradezu herausfordert.

3.2 Erfahrungen

Leider konnten wir das komplette Experiment noch nicht durchführen, da sich von völlig unvermuteter Seite im letzten Moment Schwierigkeiten ergaben: die RCX bieten unter LegOS keine Unterstützung von trigonometrischen Funktionen, diese sind jedoch für eine sinnvolle Verarbeitung der Koordinaten durch die RCX erforderlich. Obwohl wir keine für LegOS/RCX geeigneten mathematischen Bibliotheken fanden, ergaben unser Versuche mit Näherungsformeln erste Ergebnisse, so dass diese Hürde in absehbarer Zeit überwindbar gewesen wäre. Dennoch mussten wir uns aufgrund der knappen Zeit in den letzten Tagen vor der Vorführung des Experimentes auf eine Durchführung ohne den „Runner“ beschränken.

Die Ergebnisse sind nichtsdestotrotz aufschlußreich. Probleme ergeben sich vor allem aus der Hardware: So dauert die LNP-Kommunikation zwischen dem Tower und den RCX verhältnismäßig lange (im Sekundenbereich) und ist besonders im Randbereichen der definierten „Welt“ sehr störanfällig. Schon ungünstige Lichtverhältnisse kann den von einem Tower abdeckbaren Bereich drastisch einschränken. Weiterhin kostet die Infrarot-Kommunikation die RCX sehr viel Strom, so dass die Batterien schon nach kurzer Zeit an Leistung verlieren, was wiederum die Reichweite der Kommunikation weiter einschränkt. Hier sind eindeutig noch kreative Lösungen gefragt, um Experimente gegebenenfalls auch über längere Zeit unbeaufsichtigt laufen lassen zu können.

[5] Die RCX-Runner Komponente entscheidet dabei, was nah genug ist

4 Eingesetzte Entwurfsmuster

Hier sollen noch einmal alle Entwurfsmuster zusammenfassend aufgezeigt werden, die wir in den einzelnen Komponenten eingesetzt haben. Nicht einbezogen werden dabei Entwurfsmuster, die sich aus den eingesetzten Frameworks selbst ergeben (so setzt z.B. jedes COM-Objekt das Proxy Pattern ein).

4.1 Creational Patterns[6]

Creational Patterns abstrahieren in irgendeiner Weise den Weg, wie Objekte erzeugt werden. Dabei ist das bei uns wohl am häufigsten überhaupt eingesetzte Entwurfsmuster das Singleton-Pattern: Es sorgt dafür, dass stets nur eine Objektinstanz einer Klasse existieren kann.

Benutzt haben wir dies z.B. bei der Konfiguration: Die Klasse „Configuration“ ist ein Singleton. Somit wird einerseits sichergestellt, dass es immer nur eine zentrale Konfiguration gibt, andererseits muss man eine Referenz auf das Konfigurationsobjekt nicht immer wieder als Parameter in die Schnittstellen mit einbauen oder gar global deklarieren. Ein weiteres Beispiel für den sinnfälligen Einsatz des Singleton Patterns ist der Tracking Data Server, der für das Übermitteln der in den CamShift-Filtern errechneten Koordinaten an die RCXRunner-Komponente zuständig ist. Der TDS ist ein COM-Singleton, das heißt es gibt garantiert nur eine Instanz auf jeder Maschine. Somit können die Filter dem TDS leicht die Daten übermitteln und die RCXRunner-Komponente diese abfragen, ohne dass sie sich (außer auf die Maschine, auf die der TDS laufen soll) auf irgendetwas einigen müssen. Dies garantiert maximale Flexibilität: Es ist so ohne weiteres möglich, andere Methoden der Positionsbestimmung zu entwickeln und einzusetzen, ohne überhaupt etwas an den restlichen Komponenten ändern zu müssen.

Die Konfiguration setzt aber auch mit dem Builder ein weiteres Creational Pattern ein. Dieses sorgt für eine maximale Entkoppelung zwischen der Erzeugung eines (komplexen) Objektes und seiner Repräsentation. Dies wurde aber schon ausführlich im Kapitel 2.5 besprochen.

4.2 Structural Patterns

Structural Patterns beschäftigen sich damit, wie Klassen und Objekte sich zu größeren Strukturen zusammensetzen.

Ein Beispiel dafür ist das Adapter-Entwurfsmuster: Eingesetzt haben wir es bei den Wrappern für dll und LNPoI. Es sorgt dafür, dass ein (externes) Programm im Falle von dll, bzw. eine C++-Schnittstellenklasse bei LNPoI gekapselt wird und sich dem Rest des Systems als normale .NET-Komponente anbietet. Ein Interface wurde also in ein anderes, für den Rest des Systems leichter benutzbares Interface umgewandelt.

Das Composite-Pattern führt hingegen dazu, dass Objekte in Baumhierarchien zusammengesetzt werden, um die einzelnen Objekte und Zusammensetzungen von diesen einheitlich behandeln zu können. Ein Beispiel dafür ist wiederum die

[6] Bei der Benennung der Patterns sowie der Einteilung in Gruppen beziehen wir uns auf das in der Literaturliste angegebene Buch [1].

Konfiguration: Auch hier bilden die einzelnen Konfigurationseinträge einen Baum, den man traversieren kann und bei dem jedes Objekt, ob Blatt oder Zweig, eine einheitliche Schnittstelle („IConfigNode“) aufweist.

Auch das Facade-Pattern, das einzelne Schnittstellen in einem Untersystem zu einer einzigen zusammenfasst, haben wir eingesetzt. So kann man etwa den Experiment-Controller als Facade auffassen, da er nichts anderes macht als die Schnittstellen der beiden Subsysteme „RCXRunner“ und „LegOSCompile“ zur Schnittstelle IDCLControl zu vereinheitlichen. Vernachlässigt man die minimale Steuerungsfunktionalität, dient auch „LegOSCompileControl“ nur dazu, die Schnittstellen des „LegOSSecurityChecker“ und des „LegOSCompile“ zu einer – einfacheren – Schnittstelle nach aussen hin zusammenzufassen.

4.3 Behavioral Patterns

Behavioral Patterns beschäftigen sich schließlich mit Algorithmen und der Aufteilung von Verantwortung zwischen Objekten.

Wie im Kapitel 2.2 schon angeführt, setzen wir dort unter anderem bei der „GetStatus()“ Methode das „Chain of Responsibility“-Pattern ein: Die „GetStatus()“-Anfrage wird in der Objekthierarchie ausgehend vom Experiment-Controller immer weiter propagiert, bis sich ein Objekt findet, das die Frage nach dem momentanen Status des Experiments beantworten kann.

5 Projektverlauf

Viele von uns waren schon vorher als Hilfswissenschaftler am Lehrstuhl für Betriebssysteme und Middleware im Bereich DCL beschäftigt. Unter diesen Voraussetzungen bot es sich an, das Projekt als Semesterprojekt auch für die Vorlesung zu übernehmen.

Die erste Zeit beschäftigten wir uns hauptsächlich damit, ein grobes Systemdesign zu entwerfen und uns mit dem LEGO Mindstorms System anzufreunden. Dazu gehörte u.a. die Evaluation der Reichweite des Infrarot-Empfangs bei den RCX und Überlegungen, wie wir am besten die Positionen der Roboter verfolgen sollten. Auch wurde an einem möglichst robusten Roboterdesign gefeilt. Weiter versuchten wir, uns in die Tiefen und Untiefen des .NET Frameworks einzuarbeiten.

Irgendwann stellten wir mit Schrecken fest, dass wir bis zum Projektvortrag nur noch knapp 1 ½ Monate Zeit hatten und ja auch nebenher noch der „normale“ Universitätsbetrieb weiter lief. Wir entwickelten dann das Konzept zuende und sprachen es mit unserem Dozenten ab – die eigentliche Implementierung der Komponenten konnte schließlich beginnen.

Schnell stellte sich eine klare Trennung in kleinere Gruppen heraus, die Dank „Visual Source Safe“ selbständig arbeiten konnten. Nach den anfänglich zu erwartenden Schwierigkeiten mit der für uns neuen Entwicklungsumgebung nahm der Quellcode dann langsam an Umfang zu und nach und nach wurden immer mehr Komponenten in das System integriert. Besondere Hürden waren dabei, wie schon weiter oben erwähnt, COM bzw. DirectX: Dachten wir anfangs noch, durch einfaches Modifizieren der Komponenten zum Erfolg zu kommen, stellte sich schnell heraus, dass zumindest ein Teil von uns sich in die COM-Programmierung tiefer einarbeiten musste. Auch der Einsatz von Managed C++ kostete einigen von uns viel Zeit und Nerven. Dabei immer wieder sehr hilfreich war die Hilfe von .NET, die auch manchmal über ein Thema zwei verschieden Erklärungen bereit hält, auch wenn nur eine davon funktioniert. Letztendlich müssen wir aber sagen, dass die Teile – einmal erfolgreich zum Laufen gebracht – sich erstaunlich gut zum Gesamtsystem integrieren ließen.

Sehr interessant waren die ersten Tests des Gesamtsystems mit den Robotern, die die zum Teil doch überraschend stark eingeschränkte Funktionalität der RCX deutlich werden ließen. Vor allem die Infrarot-Kommunikation machte uns immer wieder Probleme. Schließlich brachten wir das System aber doch in einen halbwegs stabilen Zustand und der großen Vorführung stand nichts mehr im Wege – außer der Instituts-Firewall. Die Umgebungsbedingungen hatten wir bis dahin noch nicht wirklich in Betracht gezogen, und das rächte sich. Das Institut ist gut geschützt – zu gut für unser System. Doch Dank unserer Administratoren war dies (fast) kein Problem.

Nachdem unsere Vorführung schon gescheitert schien, konnten wir das Hauptproblem – zu leistungsschwache Batterien – doch noch lokalisieren und das Experiment erfolgreich demonstrieren.

6 Testen & Qualitätssicherung

Das Testen stellte uns vor einige Probleme: Das Projekt eignet sich auf Grund seiner Gefahr für Mensch (Nerven ☺) und Maschine (die armen RCX, die vom Tisch fallen) nicht für automatische Tests. Außerdem kamen noch die starken Hardwarebeschränkungen durch die RCX hinzu, wobei uns vor allem die Infrarot-Übertragung uns bis zum Schluss Probleme bereitete. Es boten sich also nur zwei Testarten an: manuelle Tests der fertigen Anwendung und ein Testen des Quellcodes.

Um das Testen zu vereinfachen und zu verbessern setzten wir Log4Net ein. Dieses Tool stellt eine umfangreiche, leicht zu konfigurierende und übersichtliche Protokollierfunktion zur Verfügung. Dies haben wir im ganzen System konsequent eingesetzt, wodurch besonders die Fehlersuche stark vereinfacht wurde. Log4Net half vor allem beim Auffinden von nicht-systemkritischen Fehlern, die aber dennoch zu unerwünschten Ergebnissen führten.

Das Testen des Quellcodes verlief bei uns zwar nicht von Anfang an nach einem formalen Plan, jedoch kristallisierte sich relativ schnell eine Vorgehensweise heraus, die bei allen Programmierteams gleich war. Rückblickend haben wir den klassischen Vier-Stufen-Plan des V-Modells umgesetzt:

Modultest – Integrationstest – Funktionstest – Systemtest.

Im Modultest erwies sich besonders die Definition von eindeutigen Schnittstellen als sehr hilfreich: die einzelnen Gruppen konnte dadurch eigenständig arbeiten und erste Modultests mit Dummy-Schnittstellen durchführen. Es fielen dabei besonders viele Fehler auf, die beim erstmaligen Umgang mit einem neuen Framework und einer neuen Programmiersprache auftreten. Als Beispiel seien hier die umfangreichen Zeichenkettenverarbeitungsfunktionen von C# genannt, die bei näherem Betrachten - und vor allem - Testen, doch nicht so einfach zu benutzen waren wie anfangs gedacht.

Nachdem die grundlegende Funktionalität der Module sichergestellt war, wurden im Integrationstest die Schnittstellen gegeneinander getestet. Hierbei wurden die Schnittstellendummies schrittweise durch die richtigen Funktionen der anderen Module ersetzt. Es stellte sich heraus, dass unsere Definitionen sehr gelungen und nur wenig Anpassungen notwendig waren, die größten Probleme entstanden allerdings bei der Verbindung der einzelnen Programmiersprachen. C#, C++ und managed C++ sind sehr eigen, was Datentypen angeht, insbesondere in die Freizügigkeit, die mit C++ und der Benutzung von Pointern einhergeht, musste viel Zeit investiert werden.

Die ersten Tests des Gesamtsystems verliefen weniger positiv, da besonders die Stabilität noch zu wünschen übrig lies. Hier begannen unsere Funktionstests. Wir teilten uns wieder in Teams auf und testeten nur einzelne Funktionen des Systems, wie die Kommunikation zu den Robotern oder die Verarbeitung der Trackerdaten. Zu den Tests des Quellcodes kamen natürlich noch die Tests der Hardware, besonders die der Roboter, hinzu.

7 Alternative Komponenten-Frameworks

7.1 Einleitung

Eine fundamentale Designentscheidung bei der Implementierung eines Systems besteht in der Auswahl des Komponentensystems. Diese Entscheidung hat Auswirkungen auf die Art und Anzahl der verfügbaren Bibliotheken und Komponenten, der Zielsprache(n), der erreichbaren Performance, die unterstützten Plattformen usw.

In diesem Projekt wurde das zu verwendende Komponenten-Framework bereits durch den „Auftraggeber“ festgelegt, so dass der Vergleich in der Retrospektive angefertigt werden konnte, was sicherlich eher untypisch ist.

In den folgenden Abschnitten wird kein allgemeiner, umfassender Vergleich mit alternativen Frameworks vorgenommen, vielmehr werden Anforderungen, die für die Realisierung dieses Projektes entscheidend waren, untersucht. Auf die .NET Plattform wird dabei nur am Rande eingegangen, da die Realisierung mittels .NET auf den vorhergehenden Seiten ausführlich dargestellt ist.

7.2 Wesentliche Anforderungen

Der Webserver, der zentral den Zugriff auf alle Experimente steuert, erhält nur eine Referenz auf das Experimentobjekt. Durch das Verwenden von Remoting wird dementsprechend der Ort des Experiments völlig vom Standort des Webservers entkoppelt, dementsprechend ist die Unterstützung eines transparenten Verteilungsmodells für eine Realisierung des Projektes erforderlich. Auch wenn für dieses konkrete Projekt durch die Nutzung von .NET-Remoting auf der Seite des Webservers die Wahl des Komponentenframeworks bereits getroffen war, soll dieser Aspekt dennoch beleuchtet werden.

Weiterhin waren für viele Anwendungsfelder, wie z.B. Kompilieren und Kommunikation, nur externe Programme, jedoch keine Bibliotheken verfügbar. Dementsprechend ist auch das Erzeugen von Prozessen sowie die Kommunikation mit diesen (über stdin/stout/error) ein entscheidendes Kriterium.

Im Rahmen der Vorbereitungsphase wurde außerdem deutlich, dass die für die Bildverarbeitung notwendigen Bibliotheken nur als COM-Komponenten vorlagen, so dass auch die Interaktion mit COM wichtig sein würde.

Schließlich war für die Zusammenarbeit mit LNPOI nur eine C++-Klassen (im Quelltext) erhältlich, so dass das entsprechende Framework das Einbinden solcher Klassen möglichst leicht realisierbar machen sollte.

Im folgenden werden Corba und Java im Bezug auf diese Punkte untersucht.

7.3 Corba

Corba erlaubt die Erstellung von objektorientierten verteilten Systemen, wobei Corba selbst ein sprach- und plattformneutraler Standard für die Übertragung von Objektdaten ist. Durch die Definition eines Übertragungsprotokolls sowie eines Mappings von Corba-Datentypen auf Datentypen der entsprechenden Zielsprachen wird sowohl plattform- als auch sprachunabhängige Interaktion zwischen Objekten

möglich. Dementsprechend ist die Realisierung von „Remote-Experimenten“ auch mit Corba machbar.

Da Corba selbst lediglich die Kommunikation zwischen Objekten ermöglicht, jedoch nicht die Implementierung von Funktionalität, ist die Eignung von Corba für unser Projekt nur in Verbindung mit einer spezifischen Programmiersprache sinnvoll.

Wird als Realisierungssprache C++ verwendet, so ist die Verwendung von C/C++ Bibliotheken und -Quellcode möglich, ebenso wie die Kommunikation mit anderen Prozessen. Für die Interaktion mit COM-Komponenten müsste man allerdings auf COM selbst zurückgreifen, so dass letztlich zwei Frameworks benötigt werden würden.

7.4 COM/DCOM

Das Komponentenmodell COM/DCOM ist faktisch mit der Sprache C++ und der Windowsplattform verwoben, womit natürlich die Nutzung von C als auch C++-Bibliotheken bzw. –Klassen sowie die Prozesserzeugung und Interprozesskommunikation möglich ist. Trivialerweise kann COM auch mit anderen COM-Komponenten zusammenarbeiten. Da mit DCOM auch netzwerktransparente Verteilung möglich ist, sind somit alle eingangs erwähnten Anforderungen erfüllt.

Nachteilig an COM/DCOM ist vor allem die recht hohe Komplexität sowohl des Frameworks an sich als auch der unterliegenden Sprache. .NET bietet – neben Garbage Collection und einer sehr großen Klassenbibliothek – sogar einfacheren Zugriff auf COM-Komponenten als COM selbst. Auf Grund der höheren Produktivität, die die Nutzung einer moderneren Sprache wie C# ermöglicht, hätten wir uns sicherlich gegen die Kombination COM/DCOM und C++ entschieden.

7.5 Java

Java ist bekanntermaßen ein sehr mächtiges, plattformunabhängiges Komponentensystem, dessen enorme Beliebtheit zu einem großen Teil auf die große Zahl verfügbarer Bibliotheken und (freier) Tools zurückzuführen ist.

Java unterstützt, ähnlich wie .NET, Remoting bereits mit „Bordmitteln”. Um die Unterschiede für den Entwickler aufzuzeigen, seien hier die für das Erstellen einer remoting-fähigen Klasse notwendigen Schritte kurz skizziert: zuerst muss ein Service-Interface, daß das Interface java.rmi.Remote erweitert, erzeugt werden. Die in diesem Interface deklarierten Methoden sind später die einzigen auch entfernt sichtbaren Methoden der implementierenden Klasse. Anschließend müssen aus dem Quellcode der Klasse mittels eines separaten Compilers (rmic) sowohl „Stub“ als auch „Skeleton“ erzeugt werden. Um ein Objekt auch einem Rechner nach aussen bekannt zu machen, muss desweiteren der „Java RMI naming service“ im Hintergrund laufen (und vorher ggf. manuell gestartet werden). Ein Prozess, der mittels Remoting auf die entsprechende Klasse zugreifen möchte, kann mittels des Java Naming Service eine Interface-Reference erhalten.

Diese Vorgehensweise stellt zwar bereits eine beachtliche Vereinfachung im Vergleich zu Corba dar, ist jedoch im Vergleich zu .NET immer noch recht aufwendig: Bei .NET genügt es, wenn die Klasse, das Remoting unterstützen soll, von einer bestimmten Basisklasse abgeleitet ist (bei C#: MarshalByRefObject).

Außerdem unterstützt Java auch Corba, das heisst mit Java können Klassen erzeugt werden, die mit Corba-Objekten kommunizieren können (bzw. von anderen Corba-Objekten genutzt werden können). Ein interessanter Ansatz ist dabei „RMI over IIOP“, dass das Java-Remoting über ein Corba-kompatibles Protokoll realisiert, so dass mittels Java Corba-Objekte erstellt werden können, ohne in die Untiefen von Corba-IDL absteigen zu müssen.

Die Erzeugung von Prozessen und die Kommunikation mit diesen ist unter Java wie auch unter .NET komfortabel, so dass beide Plattformen für diese Problemstellung geeignet sind.

Problematischer ist unter Java die Zusammenarbeit mit C bzw. C++ Bibliotheken. Die Nutzung von C Bibliotheken wird von Java durch das JNI (Java Native Interface) unterstützt, allerdings erfordert dies Änderungen am Quellcode. Diese Änderungen bestehen zwar nur in der Einführung von Makros vor den zu exportierenden Funktionen, so dass kein Code-Redesign notwendig ist, dennoch benötigt der Entwickler natürlich Zugriff auf den Quellcode. Hat er diesen nicht, so muss er die Funktionen in einer weiteren Dll kapseln (die dann die Originalbibliothek nutzt), dies stellt jedoch unter Umständen einen größeren Aufwand dar, ganz davon abgesehen, dass die Performance darunter leidet.

Es ist möglich, in Java implementierte Klassen von C++ aus zu verwenden, wohingegen der umgekehrte Weg, also die Verwendung von C++-Klassen aus Java heraus nicht unterstützt wird. Auch für die Interaktion mit COM-Objekten konnten wir kein Tool bzw. keine Bibliothek finden. Die einzige uns bekannte Möglichkeit besteht in der Verwendung von Microsofts Java Compiler (und Runtime) aus dem Jahre 1998, deren Weiterentwicklung jedoch von Sun gerichtlich unterbunden wurde.

Zusammenfassend lässt sich also feststellen, dass Java – obwohl eine sehr mächtige Plattform – für unser Projekt nicht geeignet war. Aus der Plattformunabhängigkeit folgt zwangsweise, dass plattformspezifische Komponentenmodelle (COM/COM+) nicht unterstützt werden können. C Bibliotheken werden zwar unterstützt, sind jedoch umständlich zu nutzen, während keinerlei Möglichkeit für die Nutzung von C++-Klassen besteht. Damit ist Java für wesentliche Anforderungen des Projektes nicht geeignet.

8 Ausblick/Erweiterungsmöglichkeiten

Die Komponenten sind in der ersten Version voll funktionsfähig und erfüllen auch die Anforderungen im Lastenheft; dennoch gibt es noch viele Wünsche, die teils kurzfristig umgesetzt werden könnten, teils aber auch mehr Aufwand erfordern:

8.1 Nahziele

Zunächst wird das Hunter-Runner Experiment wohl fest installiert und Schritt für Schritt einer immer größeren Anwendergruppe zugänglich gemacht werden. Hierbei werden die Komponenten erstmals wirklich im Dauereinsatz Betrieb getestet. Weiter sollten die Schnittstellen der Komponenten noch einmal auf eine möglichst große Modularität und Allgemeinheit überprüft und gegebenenfalls überarbeitet werden. Hilfreich wäre dabei auch der Aufbau von möglichst unterschiedlichen, alternativen Experimenten, die dann gegebenenfalls ebenfalls im Rahmen des DISCOURSE-Projektes verfügbar gemacht werden könnten.

8.2 Benutzerschnittstelle

Insbesondere die Schnittstelle zum Benutzer läßt zur Zeit noch eine Vielzahl von Wünschen offen: So würde z.B. eine eigene Scriptsprache für die Programmierung der RCX die Bedienung eines Experiments erheblich vereinfachen und gleichzeitig das System besser vor unerwünschten Manipulationen seitens der Benutzer schützen.

8.3 Alternativen zum RCX

Im Laufe des Projektes stießen wir – mehr als nur einmal – an die Grenzen des Funktionsumfangs der RCX. Deswegen beschäftigten wir uns kurze Zeit mit Alternativen und verwandten Arbeiten, von denen hier der Vollständigkeit halber zwei kurz Erwähnung finden sollen:

Zuerst das Handy Board (www.handyboard.com). Diese vom MIT entwickelte Platine basiert auf einem Motorola MC68HC11. Die größten Vorteile sind einerseits das sehr große Display, daß das Debugging sehr vereinfacht und andererseits die Erweiterbarkeit des Boards, die beim RCX vollständig ausgeschlossen ist. Das Handy Board wird an vielen Universitäten in Kursen zur Robotik eingesetzt und hat im Internet eine entsprechend große Community. Der größte Nachteil war der Preis. Im Vergleich zum relativ günstigen RCX (ca. 250€ inklusive Motoren, Sensoren und Bausteinen) ist das Handy Board sehr teuer (200$ in der Grundausstattung ohne Zubehör). Auch die zusätzlichen Komponenten sind dementsprechend teurer.

Die zweite Konkurrenz zum RCX, die hier erwähnt wird, hat den Hintergrund, dass anfänglich geplant war, dem RCX einen Compag IPAQ „auf den Rücken zu schnallen“. Dies ist leider nicht sinnvoll, da die Kommunikation über die jeweils vorhanden Infrarot-Ports unmöglich ist. Im Internet zu finden ist aber das Palm Pilot Robot Kit (http://www-2.cs.cmu.edu/~reshko/PILOT/). Es wurde von zwei Forschungsgruppen des Carnegie Mellon Robotics Institute ins Leben gerufen. Hierbei wir ein Palm Pilot für die Steuerung des Roboters benutzt. Dadurch hat man eine sehr direkte Schnittstelle zum Roboter - auch per Infrarot. Die größten Nachteile

sind die sehr geringe Erweiterungsmöglichkeiten des Roboters und wieder der hohe Preis.

Bezüglich der RCX sind noch kreative Lösungen gefragt, um diese auch für den Dauereinsatz „fit“ zu machen: Insbesondere die mangelnde Laufzeit der Batterien setzt dem unbewachten Ablauf der Experimente im Augenblick noch enge Grenzen. Auch die Verbesserung der Zuverlässigkeit und Reichweite der Infrarot-Kommunikation wird uns voraussichtlich noch einige Zeit beschäftigen.

8.4 Konfiguration

Das Ziel bei der Entwicklung der Konfigurationskomponente war es, die Konfigurationsdaten zentral zu verwalten und gleichzeitig für die einzelnen Komponenten die Semantik einer exklusiven Nutzung zu erhalten. Aus dieser Zielstellung resultierten dann sowohl die Idee der Namesräume als auch die der privaten Kopien für jede Komponente.

Die explizite Trennung der einzelnen Komponenten hat einige prinzipielle Einschränkungen zur Folge, die im wesentlichen dadurch bedingt sind, dass die Konfiguration nicht dynamisch – also zur Laufzeit - änderbar ist. Damit ist weder die Kommunikation zwischen verschiedenen Komponenten über die Konfigurationsdaten möglich, noch kann die Konfiguration von Komponenten genutzt werden, um Einstellungen zu sichern (z.B. benutzerspezifische Daten).

Die Unveränderbarkeit der Konfigurationsdaten war, wie geschildert, ein Entwurfsziel. Die dementsprechende Entscheidung, Kopien der (Teil-)Bäume zu verteilen, ist mit der Idee einer dynamischen Konfiguration nahezu unvereinbar. Obwohl die Methoden für die Änderung des Baumes prinzipiell bereits vorhanden sind, wirft der Umstand, dass die einzelnen Komponenten nur auf Kopien arbeiten, einige Probleme auf: so muss z.B. die Änderung an einer Kopie zur Änderung aller weiteren Kopien genutzt werden. Wenn dazu jeweils eine Referenz auf die Kopien gehalten wird, wird jedoch die „Garbage-Collection“ verhindert, so dass eine einmal angeforderte Kopie für immer im Speicher verbleibt. Was passiert, wenn bei einer nebenläufigen Applikation der erste Thread einen Unterbaum löscht und kurz darauf – bevor die Löschung abgeschlossen bzw. zu den Kopien propagiert ist – ein zweiter Thread eine weitere Information in den (noch vorhandenen) Baum schreiben will?

Dieses beiden beispielhaft angeführten Probleme sind lösbar, z.B. mit expliziten „destroy()“-Methoden und kopie-übergreifendem wechselseitigem Ausschluss[7], zeigen aber, dass das aktuelle Konzept für eine dynamische Konfiguration nicht besonders geeignet ist.

Eine alternativer Entwurf wäre die Verwendung des „Bridge“-Patterns: die Bridge würde, um den aktuellen Funktionsumfang nachzubilden, die modifizierenden Methoden des unterliegenden Knotens verbergen. Die Erweiterung auf eine dynamische Konfiguration wäre dann relativ problemlos möglich. Denkbar ist beispielsweise, dass jeder Knoten ein Attribut „writeable“ erhält. Wird dieses gesetzt, so erlaubt die „Bridge“ die Modifikation des unterliegenden Knotens. Weniger feingranular, aber für den Komponentenprogrammierer, der die Konfiguration nutzt, vielleicht leichter handhabbar wäre die Einteilung in änderbare und statische Namespaces, die ggf. unterschiedliche Bridge-Klassen zurückliefern.

[7] z.B. in C#: lock() auf Klassenattribut bzw. Klassenobjekt (lock(typeof(this)) { ... }

Schließlich bleibt noch die Frage, wie Komponenten erfahren, dass sich die Konfiguration geändert hat. Hier bietet sich das „Observer“-Pattern an, d.h. Komponenten, die sich für bestimmte Knoten interessieren, melden sich als „Listener“ an.

Es bleibt noch die Frage, ob die Änderungsoperationen an einem Knoten atomar erfolgen sollten. Dies erscheint notwendig, wenn die Konfiguration thread-sicher werden soll, da inkonsistente Informationen auftreten können Dem liegt der Gedanke zugrunde, dass die Attribute eines Knotens gegebenenfalls nur in ihrer Gesamtheit den Informationsgehalt des Knotens darstellen, z.B. ein Attribut als Pfad zum Compiler, ein zweites als Name des „Compiler-Executables“. Schließlich verhindert diese atomare Betrachtung, dass bei Änderungen an einem Knoten für jedes geänderte Attribut alle registrierten „Listener“ aufgerufen werden.

Eine interessante Überlegung ist die Realisierung des Spezialfalles „Austausch einer Klasse“, der z.B. auftreten würde, wenn man zur Laufzeit Komponenten wie z.B. den Tracker austauschen wollte. Ein naheliegender Ansatz ist die Verwendung eines „Decorators“, der den eigentlichen Tracker verbirgt. Dieser „Decorator“ meldet sich als „Listener“ auf dem Knoten an, der die Daten über den einzusetzenden Knoten enthält. Wird dieser Knoten geändert, so ändert der „Decorator“ sein internes Referenzobjekt, an das er die Aufrufe des Clients weiterleitet.

Im Rahmen des Projektes wurde auch an einer Komponente gearbeitet, die es erlauben sollte, aus den Daten eines Konfigurationsknotens ein neues Objekt zu erzeugen, von dem der Aufrufer lediglich weiß, welchem Interface es genügt. Vorraussetzung dafür ist natürlich, dass sich die entsprechenden Komponenten „selbst“ initialisieren, in dem sie die erforderlichen Initialisierungsdaten aus der Konfiguration laden. In Verbindung mit einem Tool zur automatischen, auf „Reflection“ basierenden Codeerzeugung, wie z.B. dem „Wrapper-Assistant“ von Wolfgang Schult, wäre es unter Umständen möglich, die entsprechenden „Decorator“-Klassen automatisch zu erzeugen.

9 Literaturliste

[1] E. Gamma, R. Helm, R. Johnson, J. Vlissides
Design Patterns – Elements of reusable object-oriented software
Addison-Wesley Publishing Company, 1995

[2] A. Polze
Vorlesungsunterlagen Komponentenprogrammierung und Middleware
http://www.dcl.hpi.uni-potsdam.de/LV/Components02
Hasso-Platter-Institut für Softwaresystemtechnik GmbH, 2002

[3] Tom Archer
Inside C#
Microsoft .NET (2001), 1. Auflage

[4] C. Kindel, G. Booch
Essential COM
Addison-Wesley Publishing Company, 1998

[5] G. Eddon, H. Eddon
Inside COM+
Microsoft Press (2000)

[6] Intel OpenCV Computer Vision Library – Reference manual
http://sourceforge.NET/projects/opencvlibrary
Intel Corporation

[7] F. Keller, P. Tabeling, R. Apfelbacher, B. Gröne, A. Knöpfel, R. Kugel, O. Schmidt
Improving Knowledge Transfer at the Architectural Level:
Concepts and Notations
http://www.hpi.uni-potsdam.de/apache/document/documents/
Improving_KnowTrans_ArchLevel.pdf
Hasso Plattner Institute for Software Systems Engineering (2002)

10 Anhang: Lastenheft

Dokumententyp	**Lastenheft**	Vertraulichkeitsgrad **Nur für internen Gebrauch**
Titel	Lastenheft Remote DCL	
Untertitel		
Projektbezeichnung	SST4 / 4-02	
Produktbezeichnung	Remote DCL	

10.1 Zielbestimmung

Es soll ermöglicht werden, Lego-Roboter über das Internet zu programmieren und die Programme zur Ausführung zu bringen. All dies soll möglichst automatisch und ohne Administratoreingriffe geschehen. Insbesondere soll folgendes Szenario realisiert werden:

Ein Roboter verfolgt einen Zweiten auf einem vorgegeben Parcours. Der Verfolger soll nach einem vom Benutzer erstellten Programm autonom agieren. Wenn er der Verfolger den Verfolgten geschnappt hat, soll es bemerkt und das Benutzerprogramm abgebrochen werden.

10.2 Produkteinsatz

1. Phase: Registrierte Benutzer innerhalb des HPI.
2. Phase: Der Service soll weltweit zur Verfügung stehen.

10.3 Produktfunktionen

/LF10/	Bildübertragung der aktuellen Szenerie über Website
/LF20/	Programmierung des RCX über Website (C Sourcecode)
/LF30/	Roboter Koordinaten abfragbar im RCX-Programm
/LF40/	Übertragen des Programms auf den RCX
/LF50/	Kompilieren des Programms durch den Server
/LF60/	Ausführen des Programms auf den RCX
/LF70/	Überwachen der Bewegung der RCX (Verhinderung kritischer Situationen)
/LF80/	Benutzerverwaltung (Authentifizierung und Autorisierung)
/LF90/	Überprüfung von Abbruchbedingungen (z.B. Fehler, Sieg, …)
/LF100/	Überprüfung des User-Programms
/LF110/	Logging wichtiger Daten und Funktionen
/LF120/	RCX ist durch den Benutzer jederzeit kontrollierbar (Start/Stop)
/LF130/	Roboter fahren automatisch zum Ausgangspunkt zurück

10.4 Produktdaten

/LD10/	Benutzerdatenbank
/LD20/	Roboterkoordinaten/Raumkoordinaten
/LD30/	Logfiles
/LD40/	Benutzerprogramme (Sourcen)
/LD50/	Benutzerprogramme (Kompiliert)
/LD60/	Videostream

10.5 Produktleistungen

/LL10/	Benutzung der schon vorhandenen Tools zur RCX-Programmierung
/LL20/	Verwendung von .NET als Framework
/LL30/	Speicherung des zuletzt bearbeiteten RCX-Programmes pro Benutzer
/LL40/	RCX steht einem Benutzer eine max. Zeitspanne zur Verfügung
/LL50/	Mehrere Benutzer können gleichzeitig bedient werden, solange sie RCX-Unabhängige Funktionen ausführen

10.6 Qualitätsanforderungen

Produktqualität	**Sehr gut**	**Gut**	**Normal**	**Nicht relevant**
Funktionalität			X	
Zuverlässigkeit	X			
Benutzbarkeit			X	
Effizienz				X
Änderbarkeit		X		
Robustheit			X	
Wartung		X		
Ergonomie			X	