Simulations-basierte Programmierung von Industrierobotern

Frank Ortmeier[1], Alwin Hoffmann[1], Uli Huggenberger[2], Wolfgang Reif[1], Dominik Haneberg[1], Gerhard Schellhorn[1], Christian Tarragona[2]

Zusammenfassung

Die Offline-Programmierung von Robotern – das ist die Erstellung von Steuerprogrammen bevor das gesamte Robotersystem installiert ist – gewinnt zunehmend an Bedeutung. Für komplexe Anwendungen ist dabei eine visuelle Simulationsumgebung, die das physikalische Verhalten adäquat wiedergibt, unerlässlich. Mit dem Microsoft Robotics Studio ist im Dezember 2006 eine serviceorientierte Entwicklungsumgebung zur Programmierung von Robotern erschienen. Durch eine integrierte, physikalische Simulation ermöglicht das Robotics Studio die realistische Programmierung von Industrierobotern ohne echte Hardware. Zusätzlich bietet die direkte Integration in eine moderne Standardprogrammiersprache die Möglichkeit sehr schnell und einfach neue Steuerungskonzepte und -paradigmen zu evaluieren. Durch die integrierte Simulationsumgebung können Anwendungen zuerst evaluiert werden, bevor sie am realen Roboter ausgeführt werden. Dadurch wird auch ein iterativer Entwicklungsprozess möglich. Außerdem ermöglicht eine serviceorientierte Architektur die einfache Orchestrierung einzelner Module und Sensoren zu neuen, komplexen Anwendungen.

Der Vortrag illustriert an einer Reihe von Problemen, wie dieser Ansatz bei der Softwareentwicklung von Nutzen sein kann. Es wird ebenfalls gezeigt, wie die in der simulierten Welt erstellten Programme auf realer Hardware eingesetzt werden können. Als Demonstrationshardware wird der EduBot der Firma KUKA Roboter GmbH eingesetzt.

1 Motivation

Während der letzten beiden Dekaden hat die Komplexität automatisierter Produktionsanlagen kontinuierlich zugenommen. Die Konsequenz (und das verfolgte Ziel) ist ein erheblicher Zuwachs bei der Produktivität. Im Gegenzug ist der Anteil der Softwareentwicklungskosten an den gesamten Entwicklungskosten stark gewachsen. Nicht nur bei den Kosten spielt der Softwareentwurf heutzutage eine wesentliche Rolle, sondern auch bei der Entwurfs- und Installationszeit, da in vielen Fällen die Software in weiten Teilen erst endgültig entwickelt werden kann, wenn die physikalische Hardware bereits installiert ist. Besonders ausgeprägt ist dies bei der Programmierung von Industrierobotern.

Derzeit leidet die Softwareentwicklung für Industrieroboter an zwei Schwachstellen im Entwurfsprozess. Zum einen werden die meisten Roboter in herstellerspezifischen Programmiersprachen programmiert und zum anderen können die meisten Anwendungen nur zu einem geringen Anteil „offline“ programmiert werden. Unter Offline-Programmierung versteht man in der Industrierobotik die Programmierung unabhängig vom installierten System, während die Online-

[1] Lehrstuhl für Softwaretechnik und Programmiersprachen, Universität Augsburg, Deutschland

[2] KUKA Roboter GmbH, Augsburg, Deutschland

Programmierung die Programmerstellung an der aufgebauten Anlage bezeichnet. Um die Offline-Programmierung von Industrierobotern verstärkt industriel nutzbar zu machen, ist es notwendig adäquate, aussagekräftige Simulationsumgebungen zu Verfügung zu stellen.

Eine solche Simulationsumgebung ist Microsoft Robotics Studio (MSRS). Die Kernidee dieses Tools ist mit Hilfe einer Standardsimulation – die im Bereich professioneller Computerspiele entwickelt wurde – eine möglichst generische, leistungsfähige und physikalisch korrekte Simulationsumgebung für Industrieroboter zur Verfügung zu stellen. Zur Programmierung stellt MSRS ein service-orientiertes Softwareframework zur Verfügung.

Dieser Beitrag zeigt, wie simulations-getrieben Software für industrielle Anwendungen entwickelt werden kann. Als Simulationsumgebung wird Microsoft Robotics Studio verwendet und als Beispiel die Programmierung eines mobilen Kleinstroboters der Firma KUKA Roboter GmbH vorgestellt. Abschnitt 2 gibt eine Einführung in die Welt des Robotics Studio, in Abschnitt 3 wird die verwendete Hardware – der KUKA EduBot – vorgestellt, während in Abschnitt 4 eine Beispielanwendung vorgestellt wird. Eine Zusammenfassung der Inhalte und ein Ausblick auf zukünftige Probleme ist Inhalt von Abschnitt 5.

2 Microsoft Robotics Studio

Im Dezember 2006 wurde das Microsoft Robotics Studio[3] in der Version 1.0 veröffentlicht. Vorrausgegangen war ein Artikel von Bill Gates im Magazin „Scientific American“ (Gates, 2007), in dem er die heutige Roboterindustrie mit der PC-Industrie vor 30 Jahren vergleicht. Vor allem die Heterogenität der Robotiksysteme in Bezug auf Hardware, Betriebssysteme und Programmiersprachen stellen die Basis für diesen Vergleich dar. Bill Gates ist der Meinung, dass die Heterogenität in der Robotik eine höhere Wiederverwendbarkeit von Programmen verhindert und somit das Innovationspotential der Robotik blockiert. Mit dem Robotics Studio verfolgt Microsoft daher das Ziel, die Programmierung von Robotikanwendungen einfacher und schneller zu gestalten und so die Wiederverwendbarkeit von Anwendungen zu erhöhen. Die Zielgruppe sind akademische, privat interessierte und kommerzielle Entwickler.

Das Microsoft Robotics Studio stellt eine Windows-basierte Entwicklungsumgebung für Roboteranwendungen dar und unterstützt eine Vielzahl bereits existierender Robotikhardware. Es basiert auf einer asynchronen und serviceorientierten Laufzeitumgebung, die aus zwei Kernkomponenten besteht: den Bibliotheken „Concurrency and Coordination Runtime“ (CCR) und „Decentralized Software Services“ (DSS). Zudem beinhaltet das Robotics Studio eine visuelle Programmiersprache, die eine einfache Entwicklung per Drag & Drop unterstützt, und eine virtuelle 3D-Umgebung, die ein realistisches physikalisches Verhalten aufweist.

Die Bibliothek „Concurrency and Coordination Runtime“ ist von jeder Programmiersprache der .NET 2.0 Common Language Runtime benutzbar und bietet ein nebenläufiges, nachrichtenbasiertes Programmiermodell für serviceorientierte Anwendungen. Die CCR verwaltet asynchrone Operationen, behandelt Fehlerfälle und ermöglicht Nebenläufigkeit ohne manuell programmierte Synchronisierung. Die CCR verhindert dadurch u.a. typische Fehler bei der Synchronisierung, z.B. Deadlocks

[3] http://msdn.microsoft.com/robotics/

(Verklemmungen). Softwarekomponenten sind innerhalb der CCR lose gekoppelt und interagieren ausschließlich durch asynchronen Nachrichtenaustausch.

Die zentrale Komponente der CCR ist der *Port*, der als Interaktionspunkt zwischen zwei Softwarekomponenten dient. Dabei handelt es sich um eine getypte FIFO-Warteschlange, die ausschließlich Elemente ihres Typs akzeptiert. Mehrere, unabhängige Ports können zu einem *PortSet* gruppiert werden. Ein PortSet repräsentiert somit die Schnittstelle einer Softwarekomponente innerhalb der CCR.

Üblicherweise werden Elemente, die an einen Port gesendet werden, nicht direkt verarbeitet. Stattdessen wird ein sogenannter *Arbiter* (Verteiler) für jeden Port registriert, um zu entscheiden, wie empfangene Elemente bearbeitet werden. Im einfachsten Fall kann der Arbiter das Element entgegennehmen und an eine Methode zur weiteren Verarbeiten übergeben. Außerdem können Arbiter zusammengesetzt werden, um so komplexe Koordinationsmuster, wie z.B. eine Auswahl oder eine nebenläufige Ausführung, zu repräsentieren. Nachdem die Arbiter für einen Port festgelegt sind, werden diese aktiviert. Anschließend werden an einen Port gesendete Element durch die jeweiligen aktivierten Arbiter entgegengenommen und weiterverarbeitet. Eine ausführliche Beschreibung der Bibliothek ist in der Online-Dokumentation von Robotics Studio oder bei Chrysanthakopoulos und Singh (2005) zu finden.

Die „Decentralized Software Services“ stellen ein verteiltes, serviceorientiertes Anwendungsmodell auf Grundlage von CCR bereit. Infolgedessen bestehen Anwendungen bei DSS aus einer Komposition von einzelnen Services, die jeweils über einen internen Zustand und eine Menge von Operationen verfügen. Durch die Orchestrierung dieser Dienste (z.B. graphisch über die Visual Programming Language) können komplexe Anwendungen erstellt werden, die erneut mit anderen Diensten komponiert werden können. Dadurch können Softwarekomponenten wiederverwendet und zu immer komplexeren Anwendungen zusammensetzt werden. Ferner stellt DSS eine Laufzeitumgebung für die Veröffentlichung und Ausführung von Services bereit. Außerdem stellt diese Umgebung die notwendige Infrastruktur für die Erstellung, Auffindung und Überwachung von Diensten sowie Fehlerbehandlung während der Ausführung dieser Dienste zur Verfügung.

Mit dem Microsoft Robotics Studio wurde auch das auf SOAP basierende „Decentralized Software Services Protocol“ (DSSP) eingeführt (Nielsen & Chrysanthakopoulos, 2007). DSSP wurde entworfen um das Protokoll HTTP zu ergänzen. Dementsprechend erweitern die in DSSP beschriebenen Operationen die aus HTTP bekannten Operationen (z.B. *Delete*, *Get*, *Post* und *Put*) sowohl um die Möglichkeit strukturierter Datenmanipulation als auch um eine Benachrichtigung bei dem Eintritt von Ereignissen. Daher definiert DSSP analog zu HTTP Operationen für die Abfrage des aktuellen Servicezustands (z.B. *Get*) und ergänzend Operationen für die strukturierte Modifikationen eines Zustands (z.B. *Insert* und *Update*). Außerdem kann ein Service Benachrichtigungen über Ereignisse erhalten, indem er sich mit der Operation *Subscribe* bei einem anderen Dienst registriert.

Abbildung 1 zeigt die wesentlichen Bestandteile eines Services im von DSS definierten Anwendungsmodell. Der *Service Identifier* ist eine URI[4] und identifiziert jeden laufenden Dienst eindeutig. Deswegen verweist der Service Identifier ausschließlich auf eine einzelne Instanz eines

[4] Ein Uniform Resource Identifier ist ein einheitlicher Bezeichner und besteht aus einer Zeichenfolge zur Identifizierung von Ressourcen. URIs werden bspw. im Internet zur Bezeichnung von Webseiten oder E-Mail-Empfängern eingesetzt.

Service innerhalb einer spezifischen DSS-Laufzeitumgebung. Der *Contract Identifier* ist ebenfalls eine URI und verweist auf den sogenannten *Contract*. Dieser beschreibt die Schnittstelle des Dienstes, d.h. die von dem Dienst angebotenen Operationen. Der *State* enthält alle Attribute, die für die Beschreibung des Zustands eines Dienstes relevant sind. Durch das Verarbeiten von Nachrichten kann der Zustand im Lauf der Ausführung verändert werden. Daher müssen alle Informationen, die durch die Schnittstelle abgerufen oder manipuliert werden können, im Zustand definiert sein.

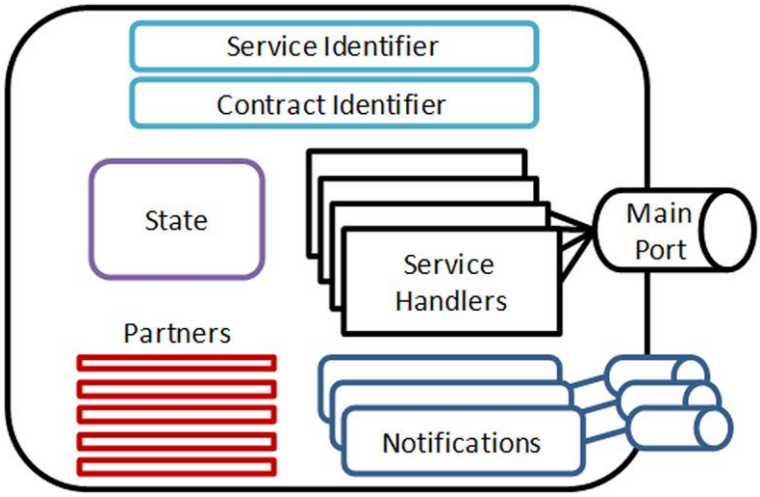

Abbildung 1: Die Bestandteile des von DSS definierten Anwendungsmodells

Um immer komplexere Anwendungen zu komponieren, müssen Dienste in der Lage sein, effizient mit anderen Diensten zu interagieren. Daher bietet DSS die Möglichkeit an, Partnerdienste zu spezifizieren. Dadurch wird für einen Dienst festgelegt, welche weiteren Dienste benötigt werden, damit eine einwandfreie Ausführung sichergestellt werden kann. Partnerdienste werden durch ihre jeweiligen *Service Identifier* spezifiziert und können entweder in der gleichen Laufzeitumgebung ausgeführt werden oder über das Netzwerk angesprochen werden.

Der *Main Port* ist ein PortSet aus der CCR-Bibliothek und nimmt Nachrichten von anderen Diensten entgegen. Die akzeptierten Nachrichten werden durch die Typisierung des PortSets bestimmt und sind entweder Operationen aus dem Protokoll HTTP oder aus dem Protokoll DSSP. Damit die Nachrichten verarbeitet werden können, muss für jeden Nachrichtentyp ein *Service Handler* registriert werden. Dieser kümmert sich um die weitere Verarbeitung der Nachricht. In dem von DSS definierten Anwendungsmodell interagieren Dienste miteinander, indem sie Nachrichten an den Main Port des anderen Dienstes senden. Zusätzlich unterstützt DSS das Publish/Subscribe-Entwurfsmuster. Dabei meldet sich ein Dienst bei einem anderen Dienst an, um über Änderungen informiert zu werden. Beim Eintreten der Änderung wird eine Benachrichtigung an den angemeldeten Service versandt. Diese Benachrichtigungen (*Notifications*) werden bei DSS über separate Ports gesendet.

Ferner unterstützt das Microsoft Robotics Studio die Simulation von realistischen Robotermodellen in einer virtuellen 3D-Umgebung und ermöglicht so die Programmierung komplexer Szenarien ohne die Notwendigkeit der Anschaffung teurer Robotikhardware. Die Simulationsumgebung arbeitet mit der AGEIA PhysX-Technologie und berechnet realitätsnahe physikalische Umgebungsbedingungen für Robotermodelle. Somit vereinfacht Robotics Studio die schnelle prototypische Implementierung von Robotikanwendungen, die zuerst in einer realitätsnahen Simulation getestet werden können, bevor sie auf realer Hardware installiert werden.

Die Simulationsumgebung ist ebenfalls als Service implementiert und kann als Partnerservice von anderen Diensten benutzt werden. Dadurch wird sie automatisch beim Start der Anwendung geöffnet. Objekte in der 3D-Umgebung (z.B. simulierte Robotikhardware) werden Entitäten genannt und haben eine graphische und eine physikalische Repräsentation. Die graphische Repräsentation besteht aus einem Drahtgittermodell und bildet die Grundlage für die detailgetreue Darstellung der Objekte. Die physikalische Repräsentation ist deutlich einfacher und approximiert nur das Drahtgittermodell. Es wird für die komplexen physikalischen Berechnungen und die Kollisionserkennung verwendet. Entitäten können zusätzlich mit Services gekoppelt und so von außerhalb der Simulationsumgebung gesteuert werden. Überlicherweise werden Entitäten, wie bspw. Roboter, mobile Plattformen oder Sensoren, auf diese Weise gesteuert.

3 Der EduBot

Die Firma KUKA Roboter GmbH stellt mit dem EduBot einen speziell für Forschung und Lehre konzipierten mobilen Kleinstroboter vor. Das Besondere an diesem Produkt ist, dass die Hardware im Wesentlichen der Hardware eines echten Industrieroboters entspricht, die Ansteuerung aber direkt über das Robotics Studio erfolgen kann. Außerdem existiert für den EduBot eine realistische, physikalische Simulation innerhalb des Robotics Studio. Dadurch wird eine effiziente, simulations-basierte Programmierung möglich.

Der EduBot ist modular aufgebaut und besteht aus zwei Teilen, einer mobilen, omnidirektional verfahrbaren Plattform und einer 5-Achs-Kinematik. Sowohl die Plattform wie auch die 5-Achs-Kinematik können eigenständig verwendet werden. In der Beispielanwendung soll jedoch die Konfiguration „Arm auf Plattform“ zum Einsatz kommen.

Die mobile Plattform hat eine Größe von ca. 30 cm x 45 cm x 8 cm (Breite, Länge, Höhe) und einen Bodenabstand von 1,5 cm. Die Traglast beträgt ca. 20 kg.

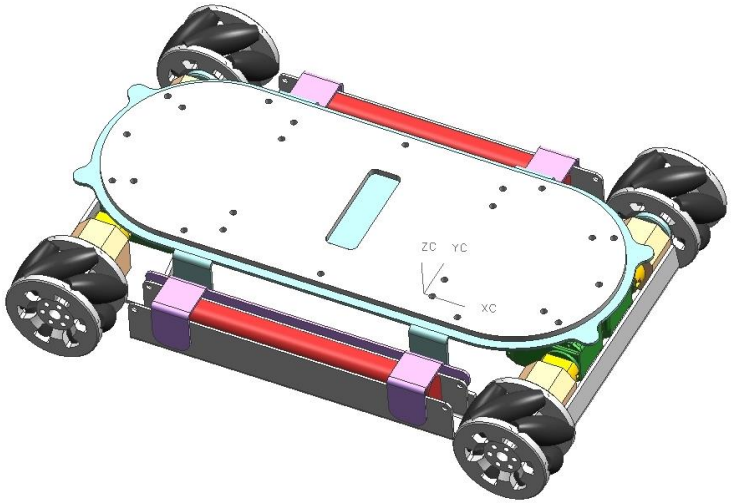

Abbildung 2: Plattform des KUKA EduBots

Um omnidirektionale Bewegungen der Plattform zu ermöglichen, werden vier unabhängig angetriebene, Mecanum-Räder (Ilon, 1975) verwendet. Dieses Rad wurde 1973 von Bengt Ilon, einem Ingenieur der schwedischen Firma Mecanum AB erfunden. Auf dem Rad sind Rollen im Winkel

von 45 Grad angebracht, wobei das Rad und die Rollen auf jedem Rad unabhängig voneinander bewegt werden. Dies ermöglicht eine Bewegung in jede beliebige Richtung. Die Achsen der geneigten Rollen zeigen dabei stets zur Fahrzeugmitte. Durch individuelle Drehzahl- und Drehrichtungswahl der Räder entstehen Fortbewegungskräfte, die sich durch die beweglichen Rollen in zwei Richtungen ausbilden, jedoch in der Summe mit den anderen Rädern zu einer Gesamtbewegungsrichtung des Fahrzeugs führen. Damit ist ein longitudinales Fahren (alle vier Räder drehen in eine Richtung), ein transversales Fahren (diagonal gegenüberliegende Räder drehen in die gleiche Richtung) und Rotation auf der Stelle (Räder auf einer Seite drehen in dieselbe Richtung) möglich. Aus einer abstrakten Sicht bedeutet dies, dass die Plattform eine 3-Achskinematik aus zwei longitudinalen und einer rotatorischen Achse besitzt.

Der Roboterarm, der auf die Plattform montiert wird, ist eine 5-Achs-Kinematik mit einer Traglast von 500g bei einem Eigengewicht von 4kg. Er hat eine Positioniergenauigkeit von 1mm. Als Tool wird ein einfacher Backengreifer mit einem Hub von 20mm verwendet. Die Greiferweite ist von 0 – 70mm stufenweise mechanisch einstellbar. Der Arbeitsraum der 5-Achskinematik ist in Abbildung 3 dargestellt.

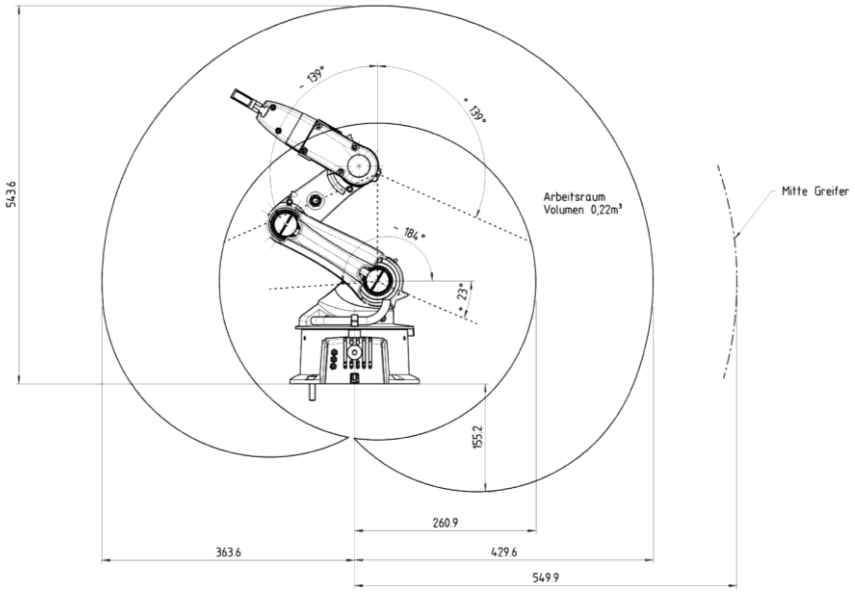

Abbildung 3: Arbeitsraum des EduBot Arms

Der – im Vergleich zu einem 6-Achs-Arm – leicht eingeschränkte Arbeitsraum kann durch Bewegung der Plattform wesentlich erweitert werden. Im Besonderen ist jeder Punkt (bis zu einer bestimmten Höhe über dem Boden) aus beliebiger Richtung durch eine geeignete Kombination von Arm- und Plattformbewegungen für den Greifer erreichbar.

Die Steuerung des Roboterarms erfolgt bei der Konfiguration „Arm auf Plattform“ durch einen in die Plattform integrierten Single Board Computer mit Intel Celeron 1GHz Prozessor, 512MB RAM und 2GB CompactFlash statt. Die Verwendung von Standard PC-Hardware ermöglicht es einfach über LAN, WLAN, USB oder eine serielle Schnittstelle optional Sensoren und Komponenten anzuschließen (z.B. Infrarotsensoren oder Webcams). Die Stromversorgung kann dabei über in der Plattform integrierte Akkus erfolgen. Damit kann der Roboter mobil und autonom ca. 30 Betriebsminuten lang arbeiten. Wahlweise ist auch ein Betrieb über externe Stromversorgung und die Steuerung per USB oder LAN möglich.

Abbildung 4: Entwurfsmuster (links) und Simulationsmodell (rechts) des EduBot

Die Programmierung des EduBot erfolgt in einer Erweiterung von Microsoft Robotics Studio, dem KUKA Educational Framework (Stumpfegger et.al, 2007). Diese Erweiterung enthält bereits eine realitätsnahe und physikalisch korrekte Simulation des Arms und der Plattform sowie Bahnplanungsalgorithmen für diese Einheiten. Dadurch eignet sich der EduBot hervorragend zur Illustration der simulations-basierten Programmierung, da der Übergang von simulierter Hardware zu realer Hardware aus Sicht der Programmierung vollkommen transparent ist.

4 Programmierung

Ein wesentliches Ziel bei der Programmierung von Robotern ist die Vereinfachung der Programmierung. Dazu müssen zwei Ziele erreicht werden. Erstens muss die Programmierung von Robotern möglichst analog zu anderen Programmierproblemen durchgeführt werden. Dies kann erreicht werden, durch den Einsatz von Standardprogrammiersprachen, -entwicklungsumgebungen und -paradigmen. Die Programmierung des EduBot erfüllt diesen Anspruch sehr gut, durch die Programmierung in C# (z.B. mit Visual Studio als Entwicklungsumgebung) und die Verwendung von service-orientierten und/oder objekt-orientierten Programmierparadigmen. Die zweite Voraussetzung ist eine realistische Ausführung und Simulation der Programme. Dieses Ziel wird beim EduBot dadurch erreicht, dass die gesamte Steuerungslogik (wie z.B. Koordinatensystemtransformationen und Bahnplanung) oberhalb der zu steuernden Komponenten angesiedelt ist. Dies wird (im Rahmen der Simulationsungenauigkeit) durch die integrierte Physikengine erlaubt. Besonders für den EduBot ist, dass die Steuerungsinterfaces der realen Hardware und der simulierten Hardware identisch und somit auch austauschbar sind. Dies bedeutet, dass Steuerprogramme ohne Modifikation sowohl für den realen Roboter als auch für die Simulation verwendet werden können.

An dem Beispiel „Türme von Hanoi“, einer bekannten Aufgabe des Mathematikers Edouard Lucas, lässt sich veranschaulichen, wie relativ komplexe Probleme auf diese Weise effizient und einfach gelöst werden können. In dieser Aufgabe geht es darum, dass die Mönche des indischen Tempel zu Benares einen heiligen Turm aus 64 aufeinander liegenden goldenen Scheiben – wobei der

Durchmesser der Scheiben nach oben abnimmt – an einen anderen Ort versetzen müssen. Dabei müssen sie folgende beiden Regeln einhalten:

Ein einzelner Mönch kann immer nur eine Scheibe auf einmal bewegen.

Es darf immer nur eine kleinere auf einer größeren Scheibe liegen (und nicht umgekehrt).

Die Scheiben dürfen nur auf heiligen Boden abgelegt werden (das bedeutet: auf einer anderen (größeren) Scheibe, dem Ausgangsplatz, dem Zielplatz oder einem speziellen Hilfsplatz).

Lucas stellt folgende Lösung vor: Der älteste Mönche befiehlt dem zweitältesten Mönch einen Turm der Höhe 63 auf den Hilfsplatz zu versetzen. Wenn dieser fertig ist, trägt er selbst die 64. Scheibe zum Zielplatz und befiehlt dann dem zweitältesten Mönch den Turm mit 63 Scheiben vom Hilfsplatz zum Zielplatz zu bringen. Der zweitälteste Mönch geht genauso vor und delegiert den Turm der Höhe 62 an den Drittältesten und so weiter. Dieses Problem ist sehr beliebt in der Informatik, da es sich in modernen Programmiersprachen sehr einfach rekursiv lösen lässt.

```
public void doTowersOfHanoi(int i, Tower from, Tower help, Tower to)
{
    if (i > 0)
    {
        doTowersOfHanoi(i - 1, from, to, help);
        Move(from, to);
        doTowersOfHanoi(i - 1, help, from, to);
    }
}
```

Abbildung 5: „Türme von Hanoi“ (exemplarischer C#-Quelltext)

Der rekursive Algorithmus ist in Abbildung 5 als C#-Programm dargestellt. Die Höhe des Turms wird durch den Integer *i* bestimmt. Die beiden Tower-Objekte *from* und *to* repräsentieren die Ausgangs- bzw. Zielposition des Turms. Der oben beschriebene Hilfsplatz wird durch das Tower-Objekt *help* dargestellt. Somit spiegelt der rekursive Algorithmus genau die oben beschriebene Lösung von Lucas wider. Einzig die Teilaufgabe, wie eine Scheibe von einem zum anderen Turm gelangt, ist durch den Algorithmus unspezifiziert und muss für den EduBot angepasst werden. Eine einfache Lösung für den Move-Befehl ist in Abbildung 6 als C#-Programm exemplarisch dargestellt.

In dieser Lösung fährt der Arm des EduBots zuerst eine Position oberhalb des ersten Turms an und bewegt sich anschließend linear zur obersten Scheibe, die er mit seinem Greifer nimmt. Nun wird die Anzahl der Scheiben auf Turm 1 herabsetzt und der Arm bewegt sich zum zweiten Turm. Dieser wird analog zu Turm 1 angefahren. Bevor die Scheibe an ihre Zielposition gebracht werden kann, wird die Anzahl der Scheiben erhöht und damit die Position der obersten, noch nicht abgelegten Scheibe aktualisiert. Zum Schluss wird die Scheibe abgelegt und der Arm in eine Position über dem Turm gebracht. Der Zugriff auf den EduBot wird in der Variable *eduBot* gekapselt.

In dem Programm werden die Positionen der Türme, der Scheiben und der Hilfspunkte in dem Objekt *Tower* gespeichert bzw. berechnet. Hier zeigt sich ein weiterer Vorteil einer modernen, objekt-orientierten Programmiersprache, da die Spezifikation der Aufgabe komplett in Objekten gekapselt wird und so einfach modifiziert werden kann. Dazu gehört beispielsweise eine Veränderung der Position der Türme oder eine Änderung der Scheibendicke oder -anzahl.

```
private void Move(Tower tower1, Tower tower2)
{
    // take a disk from tower 1...
    eduBot.Arm.PTP(tower1.PositionAboveTower);
    eduBot.Arm.LIN(tower1.PositionOfTopDisk);
    eduBot.Gripper.Close();
    tower1.DecreaseSize();
    eduBot.Arm.LIN(tower1.PositionAboveTower);

    // ... and put the disk on top of tower 2
    eduBot.Arm.PTP(tower2.PositionAboveTower);
    tower2.IncreaseSize();
    eduBot.Arm.LIN(tower2.PositionOfTopDisk);
    eduBot.Gripper.Open();
    eduBot.Arm.LIN(tower2.PositionAboveTower);
}
```

Abbildung 6: Bewege eine Scheibe von Turm 1 nach 2 (exemplarischer C#-Quelltext)

Bevor das oben beschriebene Programm auf dem realen EduBot ausgeführt wird, kann es zuerst in der Simulation getestet werden. Dabei kann z.B. festgestellt werden, ob die Hilfspunkte oberhalb der Türme zu niedrig gewählt worden sind oder ob die Berechnung der Scheibenposition fehlerhaft ist. Nach erfolgreichen Tests in der Simulation kann das Programm ohne große Anpassungen auf die reale Hardware überspielt werden. Die Simulation der „Türme von Hanoi“ im Robotics Studio ist in Abbildung 7 dargestellt.

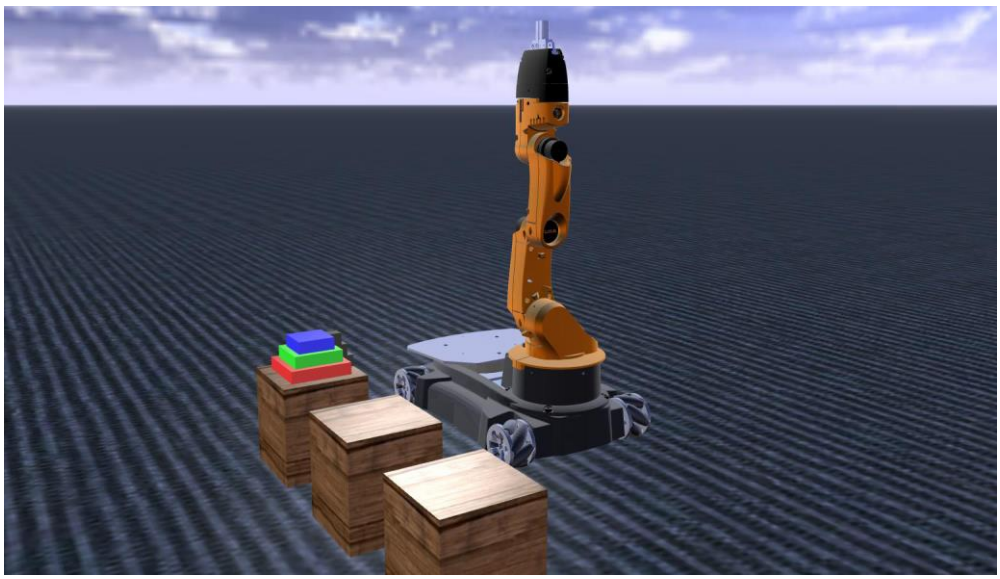

Abbildung 7: Simulation der „Türme von Hanoi“ im Microsoft Robotics Studio unter Verwendung des EduBot

Die Aufgabe „Türme von Hanoi“ ist nur ein Beispiel, wie moderne Programmiersprachen - z.B. durch Rekursion oder Objekt-Orientierung - die Entwicklung und das Testen von Robotikanwendungen vereinfachen. Ein weiteres Beispiel ist das Erreichen eines Zielpunkts innerhalb eines Labyrinths. Falls a priori der Plan des Labyrinths bekannt ist, kann für die Berechnung des kürzesten Pfades ein informierter Suchalgorithmus, wie z.B. der A*-Algorithmus, verwendet werden. Für diesen Algorithmus

existieren in modernen Standard-Programmiersprachen bereits ausgereifte Implementierungen, die eingesetzt werden können. Durch die mögliche Integration von Sensoren kann zudem auf eine Änderung des Labyrinths reagiert werden. Hierbei kann z.B. auf den D*-Algorithmus (Stentz, 1994), der in der Lage ist, neue Informationen über die Umgebung effizient zu verarbeiten, zurückgegriffen werden. Die Integration von Sensoren und der Einsatz des Algorithmus kann in der Simulationsumgebung getestet werden. Anschließend kann die getestete Software auf der realen Hardware eingesetzt werden.

Auch in der industriellen Praxis hat die simulations-basierte Programmierung ein sehr breites Anwendungsfeld. Dies reicht von relativ einfachen Palettieraufgaben, über Planung und Layout von Produktionszellen und -straßen bis hin zu komplexen Aufgabenstellungen für kooperierende Roboter. In all diesen Szenarien verkürzt ein simulations-basierter Entwicklungsprozess die Installationszeit in wesentlichem Umfang und erlaubt es, schon sehr früh Schwachstellen und Engpässe aufzuzeigen.

Es existieren aber auch Beschränkung und Probleme bei dieser Art der Softwareentwicklung. Ein Problem ist die Genauigkeit der Simulation. Es wird immer Fälle geben, in denen sich Simulation und reale Hardware unterschiedlich verhalten. Ein Beispiel – welches fast immer in Simulationen Probleme verursacht – ist das Entstehen ungewünschter Resonanzen. Dieses Problem tritt vor allem bei Robotern mit geringer Steifigkeit und großen Armlängen auf. Im Bereich der kooperativen Robotik besteht zudem der Bedarf an hartechtzeit-synchroner Steuerung von Robotern. Ein Beispiel hierfür ist die Lastverteilung auf zwei Roboter beim Transport einer Glasscheibe. Damit diese Anwendung in der Praxis funktioniert, müssen die Roboter in harter Echtzeit synchronisiert werden, um die Spannungskräfte innerhalb des Glases gering zu halten. Während dies in der Simulation relativ einfach erreicht werden kann, stellen reale Kommunikationsprotokolle und deren Latenzzeiten in der Realität häufig große Probleme dar. Ähnliches gilt für die Sensorintegration bzw. die echtzeitfähige, adaptive Bahnkorrektur mittels Sensoren. Eine Konsequenz dieser Beschränkungen ist, das Systemtest niemals überflüssig werden können.

Trotz all dieser Beschränkungen wird die Zukunft der Roboterprogrammierung auf jeden Fall im Bereich der simulations-basierten Programmierung liegen. Nicht nur weil die Softwareentwicklung dadurch schneller und effizienter wird, sondern auch weil sie dadurch wesentlich einfacher und verständlicher wird. Dazu trägt vor allem die grafische Visualisierung bei, in der auch verschiedene Informationen im Sinne von Augmented Reality eingeblendet werden können (wie z.B. Arbeitsräume, Bahnen und Ähnliches).

5 Zusammenfassung

Die Programmierung von Industrierobotern wird sich in den nächsten Jahren entscheidend ändern. Die bisherigen proprietären Programmiersprachen werden durch Standardprogrammiersprachen und Entwicklungswerkzeuge ersetzt werden. Methodisch wird die Offline-Programmierung weiter an Bedeutung gewinnen. Dazu ist die graphische Visualisierung genauso entscheidend wie eine physikalisch korrekte Simulation. Das Robotics Studio von Microsoft ist ein erstes Entwicklungswerkzeug, welches beide Komponenten enthält. Am Beispiel des EduBot kann sehr illustrativ gezeigt werden, wie sich diese neue Art der Softwareentwicklung anfühlt und auswirken kann. Der Softwareentwickler hat mehr Zeit zur Verfügung, um sich auf die eigentliche Lösung des Problems zu konzentrieren, wodurch nicht nur die Entwicklungszeit sinkt, sondern auch die Qualität der Software zunimmt.

Damit die simulations-basierte Programmierung in der breiten Praxis eingesetzt werden kann, müssen einerseits präzise Simulationsmodelle der Komponenten und deren Beschreibungen (Wo sind diese gültig? Wann treten Abweichungen auf?) erarbeitet werden und andererseits – wie beim EduBot bereits geschehen – die Möglichkeit zum transparenten Umschalten von simulierter auf reale Hardware geschaffen werden.

6 Literatur

Chrysanthakopoulos, G. und S. Singh: An Asynchronous Messaging Library for C#; Proceedings of OOPSLA 2005 Workshop on Synchronization and Concurrency in Object-Oriented Languages (SCOOL), San Diego: Oktober 2005

Gates, B.: A Robot in Every Home; Scientific American Magazine: Januar 2007

Nielsen, H.F. und G. Chrysanthakopoulos: Decentralized Software Services Protocol (DSSP/1.0); Microsoft Corporation: 2007

Ilon, B. E.: Wheels for a course stable selfpropelling vehicle movable in any desired direction on the ground or some other base (United States Patent: 3876255): April 1975

Stentz, A.: Optimal and Efficient Path Planning for Partially-Known Environments; Proceedings of IEEE International Conference on Robotics and Automation: 1994, S. 3310-3317

Stumpfegger, T., A. Tremmel, C. Tarragona und M. Haag: A Virtual Robot Control Using a Service-Based Architecture and a Physics-Based Simulation Environment; Proceedings of IEEE ICRA 2007 Workshop on Software Development and Integration in Robotics (SDIR), Rom: April 2007