

Verteilte Software-Entwicklung in der Robotik - ein Integrations- und Testframework

Dipl.-Ing. **Ulrich Reiser**, Fraunhofer IPA, Stuttgart,
Dipl.-Inform. **Christian Plagemann**, Albert-Ludwigs-Universität, Freiburg
Dipl.-Inform. **Christoph Mies**, Fraunhofer IAIS, St. Augustin

Kurzfassung

Eine der größten Herausforderungen innerhalb der Robotik ist die Integration vieler, komplexer Hardware- und Softwarekomponenten zu einem robust funktionierenden Gesamtsystem. Neben den zahlreichen wissenschaftlichen Fragestellungen, die auf Systemebene zu lösen sind, hängt der Integrationserfolg insbesondere von der Lösung praktischer Probleme wie dem Zusammenspiel vieler Entwicklungspartner und der typischerweise stark limitierten Verfügbarkeit von Einzelkomponenten ab. Leistungsfähige Hardwarekomponenten, wie beispielsweise Mehrfingergreifer und Leichtbauarme, sind in der Regel Spezialanfertigungen und stehen daher nur wenigen Partnern innerhalb von Projekten zur Verfügung. In diesem Beitrag wird ein neues Integrations- und Testframework zur räumlich verteilten Forschung und Entwicklung an solchen Komponenten und integrierten Systemen vorgestellt. Entwickler können hierbei Beiträge zu einer Technologieplattform leisten, ohne ständigen, direkten Zugang zur Hardware besitzen zu müssen.

1. Einleitung

Das Thema Integration wurde in den letzten Jahren - insbesondere in der Robotik - immer wichtiger. Meist müssen viele Komponenten zu einem Endsystem zusammengefügt werden, die von einer großen Menge an Projektpartnern hergestellt und systemspezifisch angepasst werden [1]. Dabei sollte die jeweils aktuellste und leistungsfähigste Hardware verwendet werden, um die Unzulänglichkeiten älterer Komponenten nicht durch erhöhten Softwareaufwand kompensieren zu müssen. Die neuesten Hardwarebausteine sind jedoch meist Spezialanfertigungen und somit sehr teuer. In der Regel werden in den Robotik-Forschungslabors deshalb entweder funktional eingeschränkte Eigenentwicklungen zur Evaluierung neuer Algorithmen verwendet oder es wird auf Tests mit Hardware ganz verzichtet. Beide Vorgehensweisen führen typischerweise dazu, dass während der Integration des Gesamtsystems noch viele Probleme auf der Komponentenebene gelöst werden müssen, was wiederum die Entwicklung und den Test von Komponentenschnittstellen stark verzögern kann.

Als Konsequenz entstehen in der Praxis Roboterplattformen, deren Ausstattungen auf den jeweiligen Forschungsschwerpunkt angepasst sind. Viele neu entwickelte Algorithmen können nur schwer miteinander verglichen werden, da sie auf verschiedenen Zielsystemen arbeiten. Eine Portierung auf andere Roboterhardware ist typischerweise mit hohem Aufwand verbunden, so dass eine verlässliche Evaluierung von Alternativen oft ausbleibt. Diese ist jedoch unumgänglich, um Kontinuität und Fortschritt in der Forschung zu erreichen (vgl. beispielsweise die Standardisierungsbestrebungen im EU-Projekt RoSta [2]).

Um dem Ziel näher zu kommen, komplexe Robotersysteme auf flexible Art und Weise aus Einzelkomponenten aufbauen zu können, sind in den letzten Jahren zahlreiche Integrationsframeworks entstanden, z.B. [3,4,5,6]. In diesen Ansätzen werden häufig benötigte Softwarebausteine (z.B. Hardwaretreiber) zur Verfügung gestellt und Basistechnologien, wie die Infrastruktur zur Kommunikation oder Fehlerbehandlung, angeboten. Durchweg wird jedoch davon ausgegangen, dass die zu integrierenden Komponenten lokal zur Verfügung stehen. In dieser Arbeit wird ein Ansatz vorgestellt, um ein *räumlich verteiltes* Gesamtsystem von Robotikkomponenten zu entwickeln und zu testen [1].

2. Konzept

Zentrale Bestandteile des hier beschriebenen Integrationsframeworks sind die Kommunikationsinfrastruktur (siehe Abb. 1), das Architekturkonzept und ein definiertes Ablaufprotokoll für verteilte Systemtests mit mehreren Partnern. Ausgangspunkt der Betrachtung sei die folgende Situation: ein Softwareentwickler möchte seinen Algorithmus, z.B. einen neuen Greifansatz für einen 3-Finger-Greifer oder eine Bahnplanung für einen Roboterarm mit 7 Freiheitsgraden, auf realer Hardware testen, die jedoch nicht lokal zur Verfügung steht. Die Vorgehensweise zum Testen der Komponente besteht aus folgenden Schritten:

- I Einrichtung der lokalen Kommunikationsumgebung (CORBA, VPN, etc.)
- I Anpassung der Komponente an die definierten Vorgaben der Softwarearchitektur
- I Anpassung der Komponente an die speziellen (Hard- und Software-) Schnittstellen der physikalischen Plattform
- I Kommunikation mit der Roboterplattform über eine gesicherte Verbindung (VPN)

Sofern Hardware angesteuert wird, sind ggf. Betreiber der Roboterplattform vor Ort notwendig, um den Vorgang zu überwachen und bei Bedarf einen Notstopp auszulösen. Der Vorteil der Vorgehensweise besteht darin, dass die Integration der neuen Komponente ohne großen Aufwand der Roboterplattform-Betreiber erfolgt.

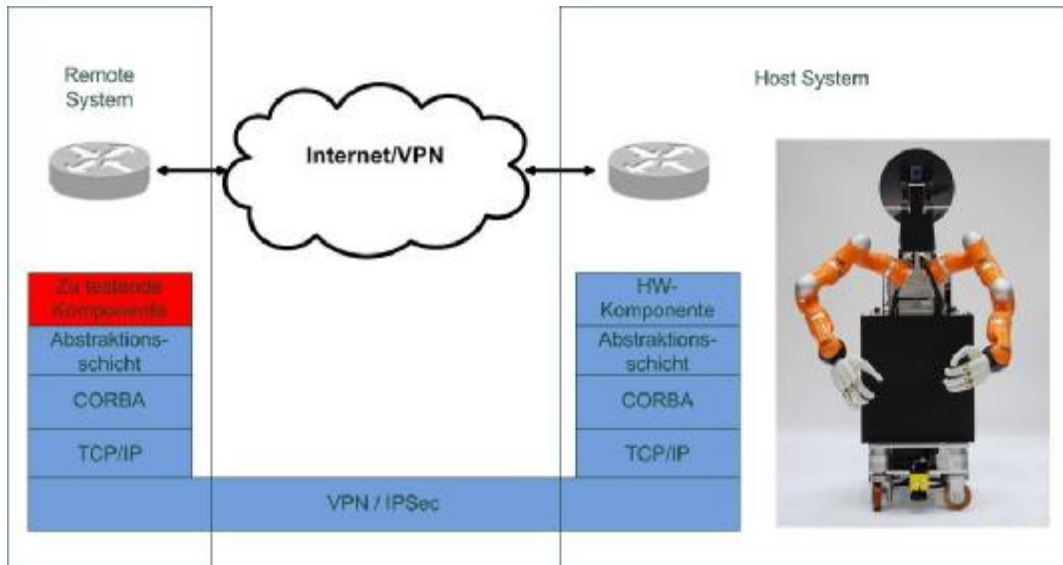


Abb. 1: Verteilte Entwicklung einer neuen Komponente (rot markiert). Die Abstraktions- und Kommunikationsschicht wird durch das Integrationsframework bereitgestellt.

3. Architektur und Middleware

Die Kernaspekte der Architektur, die eine verteilte Entwicklung von Komponenten ermöglicht, beinhalten folgende Punkte, die im weiteren Verlauf dieses Abschnitts erläutert werden:

- I Standardisierte Festlegung von Komponentenschnittstellen
- I Komponentenbasierte Software- und Hardwareentwicklung
- I Autonome Komponenten, die die Expertensysteme für spezifische Aufgaben sind
- I Wishlists zur Optimierung der Arbeitsbedingungen für die Komponente
- I Zentrale Komponenten der Architektur
- I Satz von Standard-Schnittstellen für die Servicerobotik
- I Robustes Laufzeitverhalten

Autonome Komponenten mit definierter Schnittstelle

Die Architektur kombiniert die klassische hybride Kontrolle mit einem komponentenbasierten Softwareentwurfsansatz [7]. Sie definiert das Konzept der *Autonomen Komponenten*. Jede Komponente bietet dem Gesamtsystem nach dem Entwurfsprinzip des *Design-By-Contract* eine Schnittstelle für Dienste (*Services*) an, die mit Vor- und Nachbedingungen angereichert sind. Sind die Vorbedingungen erfüllt, sichert die Komponente zu, dass die Nachbedingungen ebenfalls erfüllt werden. Es werden zwei Typen von Diensten unterschieden: *Operation* und *Kommando*. Operationen stellen einen synchronen Prozeduraufruf der Komponente dar. Kommandos lösen auf der Serverseite langläufige

asynchrone Prozesse aus, die nach ihrer Beendigung den Aufrufer benachrichtigen. Jedem Kommando wird dabei eine eindeutige Identität zugeordnet, die aus Komponentenidentität und Kommandoidentität zusammengesetzt ist. Die Dienste der autonomen Komponenten können auf einer hohen Abstraktionsebene liegen und eine relativ komplexe Funktionalität anbieten, wie beispielsweise die kollisionsfreie Navigation zu einem kartesischen Raumpunkt durch eine Komponente *Fahrwerk*.

In dieser Architektur muss die Koordinierungsebene des Gesamtsystems nur wenig über den internen Zustand der einzelnen Komponenten wissen. Dies ergibt sich direkt aus der losen Kopplung, welche durch die oben beschriebenen Schnittstellendefinitionen realisiert wird. Dadurch wird eine hohe Flexibilität erreicht, die es erlaubt, ganze Komponenten relativ einfach auszutauschen. Eine zentrale Fragestellung in einem solchen System ist die konsistente und sichere Behandlung von Fehlerfällen. Methoden hierzu stellen wir im Folgenden vor.

Wishlists

Das Konzept der *Wishlist* definiert eine Sprache, mit der Komponenten mit dem Gesamtsystem kommunizieren können. Dadurch können die Komponenten einen Antrag auf Änderung des Weltzustands stellen, um eine Optimierung ihrer Leistungsfähigkeit oder die Erfüllung von eigenen Vorbedingungen zu erreichen. Dies ist sinnvoll, da die Komponente als Experte für ihren eigenen Aufgabenbereich am ehesten den Grund für ihr Scheitern oder eigenen Leistungsabfall feststellen kann. Über die Erfüllung der *Wishlist* entscheidet die *Ablaufsteuerung* (s.u.).

Zentrale Komponenten

Um die verteilte Architektur zu koordinieren, dienen im Wesentlichen drei ausgezeichnete Komponenten: die *Ablaufsteuerung (AS)*, der *Planer* und das *Eigenmodell (EM)*. Der Planer trifft die strategischen Entscheidungen des Gesamtsystems und erstellt Pläne zur Erreichung von Zielen bzw. zur Erledigung ihm gestellter Aufgaben. Die geschieht einerseits bei Bedarf während der Laufzeit oder, für wiederkehrende Aufgaben, auf Vorrat. Die Ablaufsteuerung sorgt für die Ausführung der erstellten Pläne, indem sie sie in eine Abfolge von Komponentenaufrufen zerlegt und die Parameterübergabe koordiniert. Das Eigenmodell verwaltet Wissen über den Zustand des Roboters, z.B. den aktuellen Status einzelner Komponenten, die momentane Roboterposition in der Welt oder die aktuelle Gelenkstellungswinkel des Manipulators. Gleichzeitig stellt das EM einen zentralen

Überwachungsdienst des Systemzustands zur Verfügung. Mit diesen Daten wird eine kontinuierliche Eigendiagnose und gegebenenfalls eine Fehlerbehandlung auf Systemebene durchgeführt.

Robustes Laufzeitverhalten

Das in diesem Beitrag vorgestellte Integrations- und Testszenario geht von verteilten Hardwareressourcen und Softwarekomponenten aus, die durch mehrere Partner zeitgleich entwickelt und getestet werden. Es hat sich gezeigt, dass Testläufe von mehreren Komponenten und insbesondere des Gesamtsystems stets zu hoch interaktiven Prozessen werden, in deren Verlauf einzelne Komponenten *während des Betriebs* ab- und zugeschaltet werden müssen. Darüber hinaus soll der unplanmäßige Ausfall einer Komponente nicht zu Fehlverhalten in anderen Systemteilen führen. Um diesen Anforderungen zu entsprechen, wurde das Konzept des *Komponenten-Referenz-Managers* (KRM) entwickelt. Der KRM ist in jede zu testende Komponente integriert und kapselt deren Kommunikation mit dem Gesamtsystem. Konkret beinhaltet der KRM

- 1) einen Hintergrundprozess, der eine aktuelle Liste von Referenzen auf externe Komponenten hält und deren Status überwacht, sowie
- 2) eine Stellvertreterklasse (*ComponentProxy*), die einen transparenten, abgesicherten und bequemen Zugriff auf die externen Komponenten ermöglicht.

Die Implementierung des KRM folgt dem *Singleton* Entwurfsmuster [8], d.h., innerhalb jeder Komponente existiert nur genau eine Instanz des KRM sowie ein zentraler Zugriffspunkt darauf. Das systemzentrale Eigenmodell (siehe oben) wird von den einzelnen KRM-Instanzen der Komponenten automatisch über gescheiterte CORBA-Aufrufe unterrichtet. Es leitet diese Information wiederum als Statusmeldung an alle KRM-Instanzen weiter und verhindert so eine Kaskadierung von Fehlaufrufen. Dem Komponentenentwickler stehen der *Komponenten-Referenz-Manager* sowie die zugehörige Stellvertreterklasse als C++- und Java-Implementierung zur Verfügung, so dass ein Großteil der CORBA-spezifischen Fehlerbehandlung und Statusverwaltung von der eigentlichen Programmlogik getrennt werden kann. Abb. 2 gibt die Schnittstellen des KRM und der Stellvertreterklasse in UML wieder. In C++ lässt sich die Stellvertreterklasse effizient mittels *Templates* implementieren, so dass der Komponentenzugriff schon bei der Programmerstellung automatisch für das konkret verwendete CORBA-Interface optimiert werden kann. Im folgenden Abschnitt wird ein Programmbeispiel für die Verwendung des KRM sowie eines Zugriffs mittels der Stellvertreterklasse gegeben.

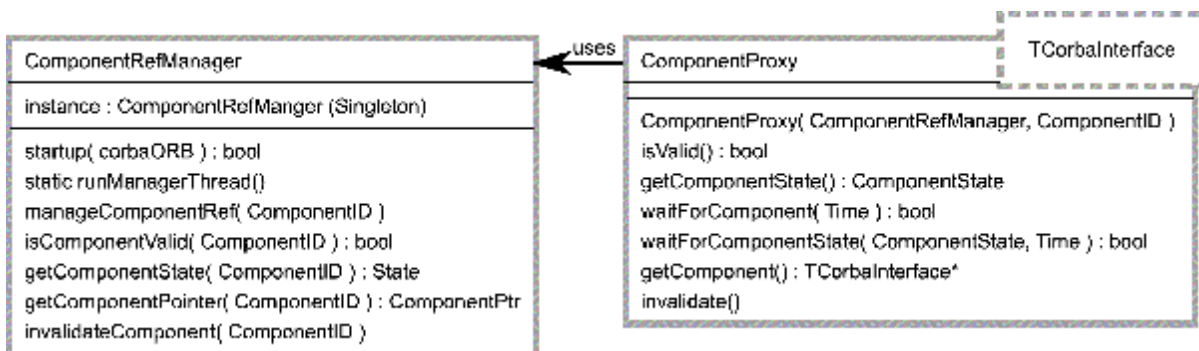


Abb. 2: UML Darstellung des Komponenten-Referenz-Managers sowie der Stellvertreterklasse für Komponenten.

4. Erstellung und Integration neuer Komponenten – ein Fallbeispiel

Am Beispiel einer Komponente mit einfacher Funktionalität, der Ansteuerung einer Pan-Tilt Einheit zur Ausrichtung eines Sensorkopfes, soll im Folgenden die Vorgehensweise zur Definition, Implementierung und Integration einer neuen Komponente veranschaulicht werden. Diese Komponente sei hier als *Kopfsteuerung* bezeichnet

Erstellung einer architekturkonformen Komponente

Architekturkonforme Komponenten besitzen einen minimalen Satz an Operationen, die implementiert werden müssen:

```

|   ReturnCode startComponent();
|   ReturnCode breakComponent();
|   ReturnCode continueComponent();
|   ReturnCode stopComponent();
|   ReturnCode aliveCheck();
|   ReturnCode externalComponentRenewed( in eComponentId componentId );
|   ReturnCode exitComponent();
|   ReturnCode getCommandTaskState( in CommandTaskID id,
|                                   out CommandTaskInfo info );
|   ReturnCode GetComponentState( out eComponentState componentState );
|   ReturnCode getRunningCommands( out CommandTaskIDlist commands );
  
```

Diese Schnittstellendefinition richtet sich nach der *CORBA-Interface Definition Language* (IDL), mittels der Schnittstellen sprachunabhängig modelliert werden können. Durch den IDL-Compiler kann daraus nativer Code in verschiedenen Programmiersprachen automatisch generiert werden.

In unserem konkreten Beispiel, der *Kopfsteuerung*, werden zusätzlich die folgenden komponentenspezifischen Funktionen mittels der Schnittstelle *IHeadControl* zur Verfügung gestellt:

```
| CommandReturnCode lookAtWorldPose( in Pose pose, in boolean Track );  
| ReturnCode getAngles(inout double pan, inout double tilt);
```

Damit die Komponente nach dem Aufruf eines Kommandos den Erfolg zurückliefern kann, muss der Aufrufer das Interface

```
interface IObserverOfHeadControl {  
    void updateCalledByHeadControl(in CommandTaskInfo commandTaskInfo);  
};
```

implementieren. Damit ist das Interface für die Kopfsteuerung vollständig.

Integration der Komponente in das Gesamtsystem

Nach der Implementierung der Komponente (z.B. in C++ oder JAVA) sieht der Integrationsprozess die folgenden Schritte vor:

- 1) Voraussetzung für die Integration der neuen Komponente in das Gesamtsystem ist die Anbindung an das Eigenmodell, welches auf dem Zentralrechner des Roboters läuft. Bei der Überprüfung der Kommunikation mit dem Eigenmodell kann die Komponente isoliert betrachtet werden und der Test kann durch den Entwickler via VPN erfolgen. Zusätzlich sollten die Aufrufe sämtlicher Dienste aus anderen Komponenten, wie beispielsweise der Ablaufsteuerung (s.u.), getestet werden, um die systemkonforme Kommunikation der Kopfsteuerung zu gewährleisten. Dies kann ebenfalls per VPN geschehen.
- 2) Danach werden einzelne Abläufe in einem größeren Zusammenhang getestet. Dabei werden mehrere Komponenten benötigt, die jeweils den Status der Kopfsteuerung im Eigenmodell auslesen können. Diese Abfrage wird durch den oben beschriebenen KRM erleichtert (s. auch das Beispiel unten).
- 3) Der endgültige *Abnahmetest* für einen bestimmten Ablauf bzw. für ein vollständiges Szenario wird durchgeführt. Dazu müssen selbstverständlich alle teilnehmenden Komponenten vor Ort aktiv sein.

Die Anbindung an das Eigenmodell erfolgt durch eine Registrierung während der Initialisierungsphase jeder Komponente. Sobald diese erfolgreich abgeschlossen ist, können die anderen Komponenten des Systems auf die Funktionalität der Kopfsteuerung zugreifen. In dem folgenden Programmbeispiel (C++) zeigen wir beispielhaft, wie die Kopfsteuerung von einer anderen Komponente aus benutzt werden kann:

```
[001] ComponentRefManager m_componentRefManager();
[002] m_componentRefManager.startup( m_orb );
[003]
[004] DesireComponentProxy< IHeadControl >
[005]     m_headControl ( m_componentRefManager,
[006]         desire::HEADCONTROL_ID );
[007]
[008] m_headControl.waitForComponent();
```

In den Zeilen [001] und [002] der Benutzerkomponente wird der Komponenten-Referenz-Manager instantiiert und gestartet. Danach wird ein Stellvertreterobjekt (*ComponentProxy*) für die Kopfsteuerung (ausgeprägt auf die zu verwendende Schnittstelle *IHeadControl*) erstellt und auf die Verfügbarkeit der Kopfsteuerung gewartet.

```
[009] if (m_headControl.isValid()) {
[010]     try {
[011]
[012]         m_headControl.getComp()->getAngles( thePose, false );
[013]
[014]     } catch (CORBA::Exception &ex) {
[015]         m_headControl.invalidate();
[016]     }
```

Die Zeilen [009] bis [016] derselben Benutzerkomponente überprüfen, ob die Kopfsteuerung noch verfügbar ist und lesen beispielhaft deren Stellungswinkel aus. Man beachte, dass *m_headControl* in diesem Beispiel eine Instanz der Stellvertreterklasse für die Kopfsteuerung ist. Diese verwendet intern den KRM, um auf den aktuellen Komponentenzustand und die aktuelle CORBA-Referenz zuzugreifen. An diesem Beispiel wird anschaulich deutlich, dass ein Großteil der tatsächlich durchgeführten Aktionen (sowohl innerhalb der Komponente als auch bezüglich der Kommunikation über die Middleware) vor dem Benutzer verborgen bleibt. Neben der konsistenten Fehlerbehandlung steigert dies ebenfalls die Robustheit des Systems, da Programmierfehler auf Benutzerseite unwahrscheinlicher werden.

Interaktiver Komponententest

Der Aufruf von einer Methode der *Kopfsteuerung* kann interaktiv mittels einer graphischen Oberfläche für die *Ablaufsteuerung* getestet werden. Abb. 3 zeigt die Kontrolloberfläche für die Ablaufsteuerung. Im oberen Teil wird der Status der einzelnen Komponenten farblich dargestellt. Die obere Gruppe der Steuerungsknöpfe ruft Unit-Tests für die jeweiligen Komponenten auf, um die Kommunikation zu testen. Im mittleren Bereich können einzelne ausgewählte Abläufe ausgelöst werden, an denen in der Regel mehrere Komponenten beteiligt sind. Der untere Bereich der Steuerungselemente dient der Steuerung der Ablaufsteuerung und damit des gesamten Systems.

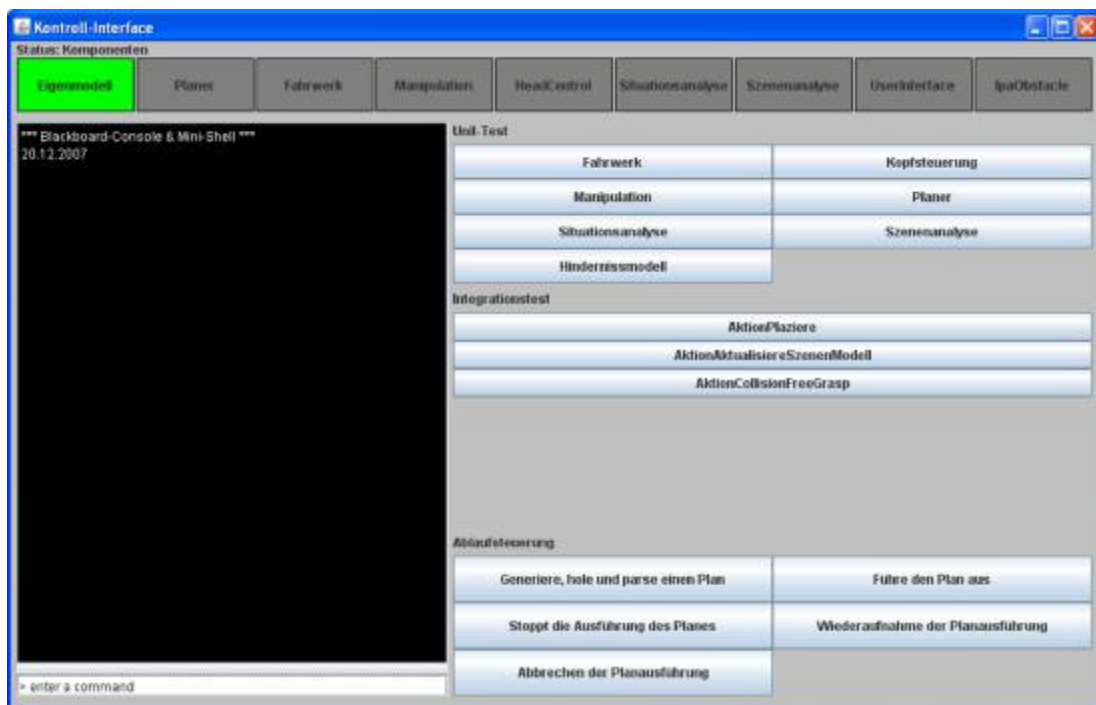


Abb. 3: Kontrolloberfläche der Ablaufsteuerung

Es sei an dieser Stelle noch einmal darauf hingewiesen, dass Tests mit der Ablaufsteuerung aus der Ferne transparent via VPN durchgeführt werden können.

5. Fazit

Das in diesem Beitrag vorgestellte Integrations- und Testframework ermöglicht die verteilte Entwicklung von Komponenten innerhalb eines komplexen Robotersystems. Die beschriebene Architektur unterstützt das Konzept der komponentenbasierten Softwareentwicklung und stellt integrative Kernkomponenten zur Verfügung. Wichtige Bestandteile des Frameworks sind außerdem standardisierte Systemschnittstellen und eine leistungsfähige und flexible Kommunikationsschicht.

Die im Verbundprojekt DESIRE [1] gesammelten Erfahrungen haben gezeigt, dass der in Abschnitt 4 beschriebene, verteilte Integrationsprozess zu einer starken Effizienzsteigerung im Vergleich zu traditionellen Herangehensweisen führen kann. Durch die vorgestellten Ferntests können bi- und multilaterale Komponententests einfacher und effizienter durchgeführt werden, so dass der Integrationsprozess kontinuierlich fortschreiten kann.

6. Referenz

Die diesem Bericht zugrundeliegenden Arbeiten wurden teilweise im Rahmen des Projekts DESIRE mit Mitteln des Bundesministeriums für Bildung und Forschung (BMBF) unter dem Förderkennzeichen 01IME01 gefördert.

7. Literatur

- [1] "DESIRE - Deutsche Service Robotik Initiative", www.service-robotik-initiative.de, 2007
- [2] "RoSta - The main international contact point for robot standards and reference architectures in service robotics", <http://www.robot-standards.org>, 2007.
- [3] Gerkey, B.P.; Vaughan, R.T.; Støy, K.; Howard, A.; Sukhtame, G.S.; Mataric; M.: Most Valuable Player: A Robot Device Server for Distributed Control. In Proc. of the IEEE/RSJ Intl. Conf. on Intelligent Robots and Systems IROS, pp. 1226–1231, Wailea, Hawaii, 2001.
- [4] Vaughan, R.T.; Gerkey, B.P.; Howard, A.: On device abstractions for portable, reusable robot code. In Proc. of the 2003 IEEE/RSJ Intl. Conference on Intelligent Robots and Systems IROS, Las Vegas, Nevada, 2003.
- [5] Côté, C.; Létourneau, D.; Michaud, F.; Brosseau, Y.: Robotics System Integration Frameworks: MARIE's Approach to Software Development and Integration. In ser. Springer Tracts in Advanced Robotics: Software Engineering for Experimental Robotics, Springer-Verlag Heidelberg, vol. 30, 2007.
- [6] Makarenko, A.; Brooks, A.; Kaupp, T.: Orca: Components for Robotics. In IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 2006). Workshop on Robotic Standardization.
- [7] Szyperski, S.; Gruntz, D.; Murer, S.: Component Software – Beyond Object-Oriented Programming. Addison-Wesley Longman, Amsterdam, 2002.
- [8] Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.: Design Patterns. Addison-Wesley, Boston, MA, USA, 1995.