

Diplomarbeit

Implementierung einer graphischen Oberfläche zur Unterstützung bei der Roboter-Softwareentwicklung in ROS

von
Filip Müllers

Bonn, 16. Februar 2012

Erstgutachter: Prof. Dr. Sven Behnke
Zweitgutachter: Prof. Dr. Armin B. Cremers



Eidesstattliche Erklärung

Ich versichere, dass ich die vorliegende Diplomarbeit "Implementierung einer grafischen Oberfläche zur Unterstützung bei der Roboter-Softwareentwicklung in ROS" selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtliche und sinngemäße Wiedergaben aus anderen Quellen sind kenntlich gemacht und durch Zitate belegt.

Filip Müllers

Bonn, 16. Februar 2012

Kurzfassung

Zu den Aufgaben eines Roboterprogrammierers für komponentenbasierte Softwaresysteme gehört das Zusammensetzen und Wiederverwenden von existierenden Komponenten sowie deren Konfiguration, um sie an eine vorgesehene Aufgabe und die verwendete Hardware anzupassen. Für das Roboterframework ROS existieren bereits viele gut getestete Komponenten und es bietet Techniken, um diese zu einem Softwaresystem zu verbinden.

In dieser Diplomarbeit wird eine graphische Kompositionsumgebung vorgestellt, die auf ROS basiert und sich dessen Techniken zunutze macht. Das Ziel ist es, Entwickler bei der Erstellung und Verwendung von komponentenorientierten Software zu unterstützen. Zu diesem Zweck wird eine Schnittstellendefinitionssprache entwickelt, mit der vorhandene Komponenten und ihren Eigenschaften beschrieben werden können. In der graphischen Benutzeroberfläche werden Spezifikationsdateien mit diesen Beschreibungen verwendet, um Komponenten typgerecht zu verknüpfen und leicht zu konfigurieren.

Durch einen engen Austausch mit der Entwickler- und Benutzergemeinschaft von ROS, werden die Anforderungen der späteren Nutzer, während des gesamten Entwicklungsprozesses des Programms, berücksichtigt. Im Anschluss an die Implementierung werden die Leistungsfähigkeit sowie weitere zuvor definierte Anforderungen an die Benutzerumgebung evaluiert.

Inhaltsverzeichnis

1. Einleitung	1
1.1. Motivation	3
1.2. Aufgabenstellung	6
1.3. Struktur der Arbeit	9
2. Grundlagen	11
2.1. Relevante Programmierkonzepte	11
2.1.1. Manuelle und automatische Programmierung	13
2.1.2. Graphische Programmierung	13
2.1.3. Komponentenbasierte Softwareentwicklung	16
2.2. Komponentenbasierte Roboterframeworks und -betriebssysteme	19
2.3. Konzepte und Techniken von ROS	25
2.3.1. Nodes	25
2.3.2. Master Service	26
2.3.3. Parameter	27
2.3.4. Messages	27
2.3.5. Topics	29
2.3.6. Services	29
2.3.7. Names und Remapping	30
2.4. Launchfiles	31
2.4.1. Launchfile-Elemente	32
2.4.2. Launchfile-Syntax	37
2.4.3. Beispiele	39
3. Verwandte Arbeiten	41
3.1. Roboter-Entwicklungsumgebungen	41
3.2. Debugging in ROS	50
3.3. Fazit	51
4. Implementierung des Programms	53
4.1. Anforderungsanalyse	53
4.1.1. Anfangsbefragung der ROS-Entwickler	54

4.1.2. Vorausgehende Befragung der ROS-Community	54
4.1.3. Anforderungen an das Programm	58
4.2. Nodespezifikationsdateien	61
4.2.1. Syntax und Semantik der Spezifikationsdateien	62
4.2.2. Name und Pfad der Spezifikationsdateien	65
4.2.3. Erstellung der Spezifikationsdateien	66
4.3. rxDeveloper	67
4.3.1. Implementierungsdetails	67
4.3.2. Ausführung des Programms	68
4.3.3. Funktionalität	68
4.3.4. Der Component Connector	70
4.3.5. Element-Editoren und Assistenten	76
4.3.6. Testing/Debugging	80
4.3.7. Der Component Creator	81
4.3.8. Specfile-Editor	83
5. Evaluation und Bewertung	85
5.1. Evaluation der Vollständigkeit	85
5.2. Nutzung der Spezifikationsfiles	89
5.3. Plattformunabhängigkeit und Verfügbarkeit	90
5.4. Bewertung	91
6. Zusammenfassung und Ausblick	93
6.1. Zusammenfassung	93
6.2. Ausblick	95
7. Verzeichnisse	97
7.1. Definitionen und verwendete Abkürzungen	97
7.2. Abbildungen	99
7.3. Tabellen	102
7.4. Listings	102
7.5. Literaturverzeichnis	103
A. Anhang	A-107
A.1. Umfragedetails	A-107
A.2. Projektseite, Quellcode und Testfiles	A-123

1

Kapitel 1

Einleitung

Roboter werden seit vielen Jahren genutzt, um dem Menschen gefährliche, lästige oder schwierige Arbeiten zu erleichtern bzw. ganz abzunehmen. Das Einsatzfeld für Robotik hat sich in dieser Zeit immer weiter vergrößert und Roboter werden heutzutage nicht nur in der Industrie, sondern auch in Forschungseinrichtungen und zunehmend im Haushalts- und Servicebereich verwendet. Abbildung 1.1 zeigt das weltweite Wachstum von Service- und Industrierobotern zwischen 2006 und 2008.

Die Anforderungen an die Roboter unterscheiden sich hierbei je nach Einsatzgebiet. Industrieroboter agieren in streng kontrollierten Umgebungen und erledigen monotone Arbeiten, die Geschwindigkeit, Präzision und Stärke verlangen. Die autonomen Fahrzeuge bei der DARPA Urban Challenge¹, die sich selbstständig im Straßenverkehr zurechtfinden oder die Marsrover², die eigenständig Teile der Marsoberfläche erkunden, agieren in Umgebungen, die sich stark unterscheiden. Serviceroboter, wie Reinigungsroboter oder Pflegeroboter wiederum müssen sich in einer verändernden Umwelt zurecht finden und zum Teil mit dem Menschen interagieren können. Alle diese Aufgabenfelder setzen unterschiedliche Hardware- und Softwarekonfigurationen voraus. Verschiedenen Sensoren, Aktuatoren und Motoren benötigen unterschiedliche und spezialisierte Treiber.

Für andere Softwarekomponenten existieren hingegen Lösungen, die sich auf mehreren Robotern wiederverwenden lassen. Diese Komponenten könnten zum Beispiel die Navigation oder die Kartografie betreffen, für die bereits Algorithmen existieren, die robust und konfigurierbar sind und nicht für jeden Gebrauch neu implementiert werden müssen. Roboter-Entwicklungsumgebungen sollten in der Lage sein, Softwarekomponenten wiederzuverwenden und aus diesen Komponenten Robotersoftware zusammen zusetzen und an die spezielle Aufgabe anzupassen, anstatt bereits vorhandene und getestete Algorithmen für jedes Robotiksystem neu zu entwickeln.

¹ <http://archive.darpa.mil/grandchallenge/> [Stand: 07.02.2012]

² <http://marsrover.nasa.gov/home/index.html> [Stand: 07.02.2012]

1. Einleitung

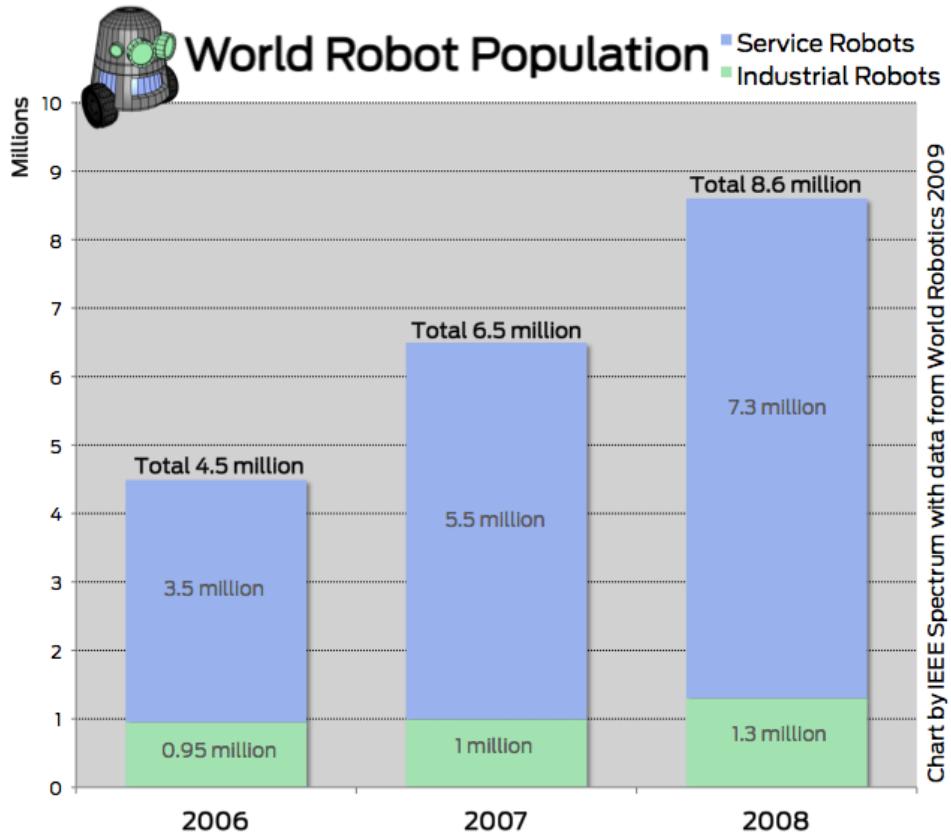


Abbildung 1.1.: Anstieg der Roboter-Population 2006-2008. (Quelle: <http://spectrum.ieee.org>, Artikel: „World Robot Population Reaches 8.6 Million“ [Stand: 07.02.2012])

Diese Diplomarbeit beschäftigt sich mit der Konzeption und der Entwicklung einer graphischen Benutzeroberfläche, die den Anwender beim Prozess der Erstellung von Robotersoftware unterstützt. Im Speziellen soll das Zielprodukt dem Softwareentwickler beim Verbinden einzelner Softwarekomponenten (Softwaremodule) in einer komponentenbasierten Softwarearchitektur helfen und hierbei mögliche Fehler reduzieren und auch frühzeitig verhindern können. Die Benutzeroberfläche soll einfach zu benutzen und plattformunabhängig sein.

In diesem Kapitel wird die Motivation (Kapitel 1.1) für diese Arbeit erläutert, die konkrete Aufgabenstellung (Kapitel 1.2) definiert sowie die weitere Gliederung der Arbeit (Kapitel 1.3) vorgestellt. Einige relevante Begriffsdefinitionen sowie verwendete Abkürzungen, sind, zusätzlich zu ihrer Erklärung in der Arbeit, in Kapitel 7.1 zum Nachschlagen zusammengefasst.



Abbildung 1.2.: Hierarchische Roboterprogrammierstruktur, nach Han *et al.* (2010). Die jeweils untere Ebene bildet die Grundlage für die darüber liegende Ebene.

1.1. Motivation

Robotik

Zunehmend gewinnt die Robotik auch im Haushaltsbereich und bei Technik-enthusiasten durch Serviceroboter und günstigere Hardwarekomponenten an Bedeutung. Eine Konsequenz daraus ist, dass es auch Menschen mit geringen technischen Kenntnissen zukünftig möglich gemacht werden muss, Roboter zu programmieren. Auf der anderen Seite werden die verwendeten Roboter immer vielseitiger. Folglich bedarf es zunehmend leichter und flexibler Programmiersysteme, um die Komplexität der Robotersysteme zu reduzieren (Biggs und Macdonald, 2003).

Die Robotik besteht aus vielen Teilgebieten. Abgesehen von der Kombinationsvielfalt bei den Hardwarekomponenten eines Robotiksystems, besteht die eigentliche Komplexität darin, Software so zu gestalten, dass alle Hardwarekomponenten zusammenwirken. Einen Roboter zu entwickeln bedeutet seine Hardwarekomponenten von Sensorik, über Aktuatoren und Motoren, bis hin zum Prozessor zusammenzufügen und anschließend softwareseitige Grundprobleme, wie etwa die Treiber für die ausgewählten Komponenten, die Navigation, die Lokalisierung oder Bilderkennung zu lösen. Erst dann kann dem Roboter eine Funktion gegeben werden. Es ist also erst möglich, die dem Roboter angedachte Aufgabe zu programmieren, wenn diese zugrunde liegenden Teilprobleme gelöst sind. Wäre man mit der Konstruktion und Programmierung eines Robotersystems auf sich allein gestellt, so wäre es ein aussichtsloses Unterfangen. Abbildung 1.2 zeigt die hierarchische Program-

1. Einleitung

mierstruktur, in der Robotersoftware entsteht. Selbst funktional einfache Roboter benötigen eine Vielzahl an Softwaremodulen. Das richtige Zusammenfügen dieser Module, von der Erstellung abgesehen, ist zeitintensiv und fehleranfällig (Namoshe *et al.*, 2008) und steigert die Komplexität der Aufgabe um ein Vielfaches (Cote *et al.*, 2004). Die Zahl der Softwaremodule, die bei vielseitigeren Robotern benötigt werden, steigt mit den Aufgaben und den benötigten Hardwarekomponenten.

Roboterbetriebssystem/Roboterkontrollarchitektur

Wie auch in anderen Bereichen der Softwaretechnologie existieren in der Robotik eine Vielzahl von Konzepten und Sprachen zur Entwicklung von und der Fehlersuche in Software. Laut Macdonald *et al.* (2003) entsprechen viele verfügbare Entwicklungsumgebungen für Roboter jedoch nicht dem Stand der Technik in der allgemeine Softwareentwicklung. Gründe dafür sind eine Vielzahl verschiedenen Roboterkonfigurationen, das Nichtwiederverwenden von Quelltext und eine hohe Fragmentierung der Roboter Middlewares (Shakhimardanov und Prassler, 2007). Bei einigen Roboterframeworks oder -kontrollarchitekturen führt der Versuch einer Portierung der Software von einem Roboter auf einen anderen zu einer vollständigen Neuentwicklung dieser Software, da kein offener Standard für Kollaboration und Wiederverwendung von Quelltext existiert (Macdonald *et al.*, 2003).

Einige Roboterarchitekturen wurden entwickelt, die diese Probleme verhindern wollen. Die entstandenen komponentenbasierten Roboterkontrollarchitekturen besitzen alle Vor- und Nachteile, auf die im Kapitel 2.2 eingegangen wird. Eine momentan sehr verbreitete Kontrollarchitektur ist das *Robot Operating System*, ROS (Quigley *et al.*, 2009). Auch ROS versucht einige der aufgezeigten Probleme zu vermeiden. Es wurde von der Firma Willow Garage³ initiiert und wird von einer weltweiten Gemeinschaft als Open-Source-Projekt weiterentwickelt. Das Grundkonzept von ROS stützt sich auf die Wiederverwendung von Quelltext sowie auf Kollaboration zur Schaffung einer gemeinschaftlichen Plattform für Roboteranwendungen (Cousins, 2010).

Mit der Entwicklung von ROS wurde bereits ein wichtiger Schritt in Richtung Vereinfachung, Modularität, Wiederverwendbarkeit und Zusammenarbeit getan. Als Roboterkontrollarchitektur mit Hardwareabstraktionsschicht und dem Konzept von möglichst atomaren Programmen, die zusammen kombiniert werden können, löst ROS viele Probleme bei der Roboter-Softwareentwicklung. Durch ei-

³ <http://www.willowgarage.com> [Stand: 07.02.2012]

ne Kommunikationsschicht erleichtert ROS die Verwendung von Komponenten und macht diese anpassbar. Gestartete Softwarekomponenten, deren Interprozesskommunikation⁴ und das daraus resultierenden Peer-to-Peer-Netzwerk aus ROS-Prozessen, wird als *Berechnungsgraph* (engl.: „computation graph“) bezeichnet (Cousins, 2010, Ceriani und Migliavacca, 2010). Hierbei entsprechen Knoten (Nodes) den Softwarekomponenten und Kanten implizieren die Kommunikationswege zwischen den Komponenten.

ROS soll als ein Betriebssystem verstanden werden und bringt viele Komponenten durch sein Softwarerepository mit (Quigley *et al.*, 2009). Auf der anderen Seite sind die vorhandenen Unterstützungsmöglichkeiten bei der Konfiguration und Erstellung von Software recht rudimentär und auch nur punktuell vorhanden. Viele der mitgelieferten Tools und deren Funktionalität sind nur per Kommandozeile (engl.: „command-line interface“ (CLI)) und ausschließlich textuell erreichbar.

Jang *et al.* (2010), die Entwickler der Middleware *OPRoS*, bemängeln bei existierenden Roboter Middlewares die Fokussierung auf das Framework und eine damit verbundene Vernachlässigung der Unterstützungsmöglichkeiten für Entwickler bei der Komponentenentwicklung sowie deren Komposition. Die Aufgaben, die ein Softwareentwickler einer komponentenbasierten Roboterkontrollarchitektur hat sind:

- Wiederverwendung von Komponenten
- Implementierung von Komponenten
- Komposition und Konfiguration von Komponenten

Da Softwareentwicklung ein notwendiger und immer vielseitiger werdender Prozess ist, wird es zunehmend wichtiger, den Entwickler hierbei zu unterstützen (Cote *et al.*, 2006).

Visuelle Programmierung

Der Mensch versucht seit jeher komplizierte Abläufe zu visualisieren. Bilder, Ikonen und Diagramme gegenüber ihrer textuellen Repräsentationen viele Vortei-

⁴ Datenaustausch zwischen Programmen und/oder Threads, die auf einem oder bei *verteilten Systemen* auf mehreren, über ein Netzwerk verbundenen, Computern laufen (engl.: „inter-process communication (IPC)“).

1. Einleitung

le. Bilder können viele Informationen tragen und helfen beim schnellen Verstehen und Erinnern. Zum Beispiel verwenden wir Straßenschilder anstelle von Text, um uns schnell die Situation im Straßenverkehr klar zu machen. Bildhafte Darstellungen vereinfachen das Erlernen von Konzepten und können Sprachbarrieren aufweichen. Durch Diagramme können zusammengehörige Informationen gruppiert und so das Wiederfinden von Informationen verbessert werden (Roy *et al.*, 1998).

Auch in der Softwareentwicklung wird vermehrt auf solche Darstellungsformen gesetzt. Nach Kim und Jeon (2008) zeigt die wachsende Anzahl an visuellen Programmiersprachen, dass graphische Programmierung state-of-the-art ist. Ein Programm, dass den Entwickler graphisch bei Erstellung, Wartung und der Fehler suche unterstützt und die Benutzbarkeit von ROS als Middleware vereinfacht, ist eine wünschenswertes Ziel. Die unterstützende Entwicklungsumgebung sollte einfach zu verwenden sein, um den Einstieg in die Roboterentwicklung zu ermöglichen. Die Implementierungszeit für Komponentenbasierte Softwareentwicklung in ROS kann durch ein solches Programm erheblich verkürzt und die Fehleranfälligkeit durch geeignete Zwangsbedingungen (engl.: „constraints“) auf ein Minimum reduziert werden. Andererseits muss beachtet werden, dass Roboterprogrammierer eine bestimmte Methodik bei der Entwicklung von Software haben und neue Programme keine zu große Umstellung erfordern sollten (Jang *et al.*, 2010).

1.2. Aufgabenstellung

Diese Arbeit soll, auf den ROS-Konzepten aufbauend, Unterstützungsmöglichkeiten zur Entwicklung von Robotersoftware bieten. Dabei soll eine graphische Oberfläche entstehen, die als Kompositionsumgebung für Softwaremodule in ROS dient und die zudem die Konfiguration und Parametrisierung der Komponenten erleichtert. Hierdurch soll eine kürzere Implementierungszeit, eine einfachere Verwendung und ein höheres Maß an Struktur bei der Erstellung von Softwaresystemen in ROS erzielt werden. Fehlerquellen in den Softwaresystemen können durch geeignete Zwangsbedingungen innerhalb des Programms auf ein Minimum reduziert werden.

In der Arbeitsgruppe „Autonome Intelligente Systeme“, in der diese Arbeit entsteht, wird in vielen Bereichen ROS verwendet. Weitere Gründe für die Entscheidung das Programm für ROS zu entwickeln wurden bereits in Kapitel 1.1 genannt. Die eigentliche Komposition und das automatische, sequentielle Starten mehrerer Softwarekomponenten erfolgt bei ROS über XML-formatierte Konfigu-

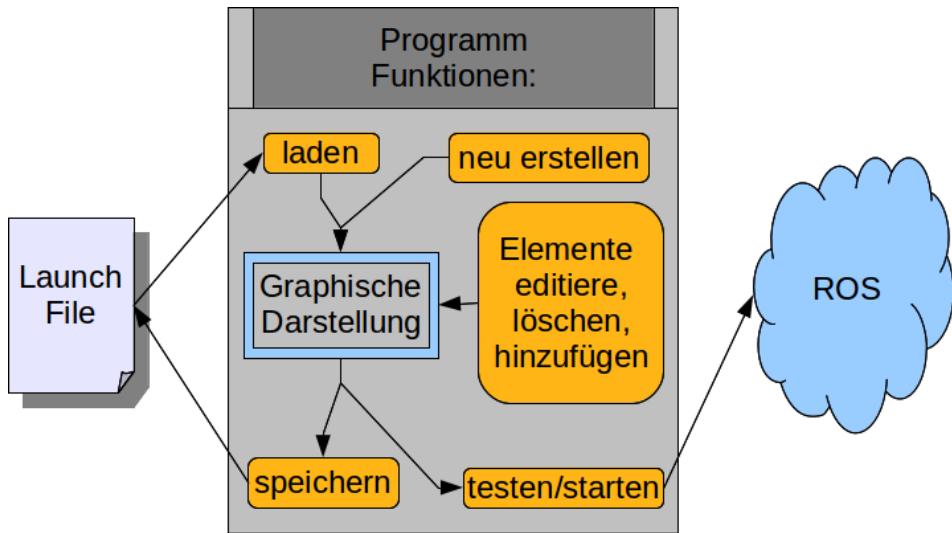


Abbildung 1.3.: Grundfunktionen des geplanten Programms. In der *graphische Darstellung* (blauer Rahmen) werden Elementen aus Launchfiles in Form eines Graphen mit Knoten und Kanten dargestellt, um den ROS-Berechnungsgraphen zu simulieren.

rationsdateien, die sogenannten *Launchfiles*. Launchfiles beschreiben die Struktur des ROS-Berechnungsgraphen und beinhalten neben den Softwarekomponenten unter anderem Parameter und Details über die Startumgebung. Die Erstellung von Launchfiles erfolgt bislang rein manuell und textuell, wobei die XML-Syntax streng eingehalten werden muss. Das hat zur Folge, dass die Beschreibung der Struktur der Robotersoftware durch manuell erstellte Launchfiles sehr fehleranfällig und zeitintensiv ist. Des Weiteren müssen die Eigenschaften der verwendeten Softwarekomponenten penibel und umständlich herausgefunden werden und deren gemeinsame Verwendung bekannt sein.

Das im Rahmen dieser Arbeit zu erstellende Programm, soll Launchfiles auf intuitive Weise graphisch erstellen und auch laden können. Elemente, z.B. neue Softwarekomponenten oder Parameter, können hinzugefügt, verändert oder gelöscht werden. Das Ergebnis kann aus dem Programm gestartet und gespeichert werden. Durch die graphische Entwicklung soll die Struktur des Berechnungsgraphen bereits während der Entwicklung übersichtlich zu erkennen sein. Abbildung 1.3 skizziert diese Prozesse.

Neben Softwaremodulen existieren eine Reihe verschiedener Elemente, die in einem Launchfile vorkommen können. Diese Elemente, die allesamt durch das Programm

1. Einleitung

manipulierbar sein müssen, werden später in Kapitel 2.4.1 näher erläutert. Die zentrale Eigenschaft des Programms ist jedoch die Behandlung der Softwaremodule (Nodes), deren Kommunikationsverwaltung untereinander und der zugehörigen Parametern. Diese Fähigkeiten sind essentiell für die Verwendung von ROS, da sie die Wiederverwendbarkeit und Anpassbarkeit ausmachen und bislang nur mit viel Wissen über die Komponenten und deren Verknüpfung bewältigt werden können.

Ein für die Informationen über die Module benötigter Teil der Arbeit stellt die Konzeption einer *Schnittstellendefinitionssprache* (engl.: „Interface Definition Language“ (IDL)) dar. Eine IDL ist eine deklarative formale Sprache, die eine Sprachsyntax zur Beschreibung von Schnittstellen einer Softwarekomponente bietet. Neben den Schnittstellen, einschließlich der zugehörigen Datentypen, können hiermit auch Parameter und vorhandene Methoden von Objekten definiert werden. *Spezifikationsdateien* sollen, basierend auf dieser IDL, die Eigenschaften der zu verwendenden Softwarekomponenten beschreiben. Die Informationen über die Kommunikationsmöglichkeiten und die Parameter der Komponenten sollen hierbei für Mensch und Maschine lesbar gestaltet werden, so dass eine intuitive Erstellung der Dateien ermöglicht wird. Im Programm sollen diese Informationen nicht nur dafür genutzt werden, dem Entwickler Verwendungsmöglichkeiten aufzuzeigen, sondern auch um fehlerhafte Benutzung zu erkennen. Zum Beispiel kann der Entwickler auf ungeeignete Parameterwerte hingewiesen werden, um diese anschließend zu korrigieren. Nach erfolgter Konzeption müssen für Testzwecke einige Spezifikationsdateien in der IDL erstellt werden.

Während des gesamten Entwicklungsprozesses des Programms sollen Entwickler und Benutzer des Roboterframeworks ROS mit einbezogen und befragt werden, damit das Produkt nicht an der Zielgruppe vorbei entwickelt wird. Die graphische Benutzerumgebung soll sich schlussendlich gut in ROS integrieren und keinen Fremdkörper darstellen, was eine Einhaltung der ROS-Konzepte und -Standards voraussetzt. Dadurch wird sichergestellt, dass ein Produkt entsteht, welches von den Entwicklern keine Umstellung des Entwicklungsprozesses fordert und auch leicht an interessierte Mitglieder der ROS-Gemeinschaft zur Weiterentwicklung übergeben werden kann. Des Weiteren soll die graphische Oberfläche (engl.: „Graphical User Interface“ (GUI)) betriebssystemunabhängig gestaltet werden.

Abschließend soll die Umgebung erprobt werden. Hierbei wird die Aufgabe sein möglichst alle Formen von Launchfiles laden und erstellen zu können. Dadurch soll sichergestellt werden, dass die Entwicklungsumgebung voll und ganz in die ROS-Welt passt, in der jedes Programm genau für ein Problem zuständig ist und dieses Problem sehr gut löst.

Abgrenzung

Das geplante Programm soll kein „Schweizer Taschenmesser“ der Softwareentwicklung in ROS werden. Es soll keine vollständige *integrierte Entwicklungsumgebung* („*integrated development environment*“ (IDE)) zur Entwicklung von Komponenten durch einen Texteditor mit Quelltextformatierungsfunktion, Compiler (bzw. Interpreter), Linker und Debugger sein. Außerdem soll hiermit keine graphische Quelltext-Programmierung erfolgen, wie zum Beispiel in LabVIEW (NI, 2012) oder Simulink (MathWorks, 2012), wo grafische Symbole zu Flussdiagrammen vernetzt und auf diese Weise Programme erstellt werden. Der Fokus liegt auf der Behandlung der Launchfiles und besonders auf einer vereinfachten Verwendung der Kommunikationsmöglichkeiten zwischen Nodes. Das Programm bezieht sich primär auf bereits existierende Komponenten, deren Komposition und Konfiguration innerhalb einer graphenähnlichen Darstellung zur Gestaltung von Roboter-Softwaresystemen.

1.3. Struktur der Arbeit

Der weitere Verlauf der Arbeit gliedert sich wie folgt:

In *Kapitel 2 „Grundlagen“* wird ein Überblick über die fundamentalen Konzepte gegeben. In diesem Kapitel soll vor allem das Roboterbetriebssystem ROS vorgestellt werden. Besonderes Augenmerk gilt hierbei der Syntax und Semantik der Launchfiles, den zugehörigen Launchfile-Tags und deren Verwendungsmöglichkeiten, da sie für die Implementierung des geplanten Programms von zentraler Bedeutung sind. Des Weiteren werden graphische Programmierkonzepte gezeigt und ein Einblick in komponentenbasierte Entwicklung sowie Roboterkontrollarchitekturen gegeben.

Anschließend wird in *Kapitel 3 „Verwandte Arbeiten“* zunächst ein Überblick über Techniken und Konzepte gegeben, die ähnliche Themen behandeln. Dabei werden einige graphische Programmierumgebungen für Roboter vorgestellt, die für verschiedene Kontrollarchitekturen existieren. Diese werden anschließend in den Kontext der Arbeit eingeordnet.

In *Kapitel 4 „Implementierung des Programms“* werden zunächst die Herangehensweise an die Umsetzung sowie die Entstehungsschritte von Programm und Spezifikationsdateien erörtert. Hier wird das Zusammenwirken mit der Communi-

1. Einleitung

ty deutlich gemacht, indem auf Befragungen von Entwicklern von ROS und der Zielgruppe, den Benutzern von ROS, eingegangen wird, die die Anforderungen an das Programm verfeinern. Schließlich wird das fertige Produkt aus Programm und Spezifikationsdateien umfassend beschrieben.

Nachfolgend wird in *Kapitel 5 „Evaluation und Bewertung“* die Leistung des Programms betrachtet. Hierbei wird in verschiedenen Tests das Laden, Speichern und Erstellen von Launchfiles überprüft, die Funktion von Spezifikationsdateien aufgezeigt und deren Verwendung im Programm analysiert. Des Weiteren wird die Verwendbarkeit des Programms auf verschiedenen Systemen getestet und die Implementierung anhand der Anforderungen bewertet.

In *Kapitel 6 „Zusammenfassung und Ausblick“* werden die Ergebnisse der vorliegenden Arbeit zusammengefasst und mögliche Einsatzfelder für das vorgestellte Programm aufgezeigt. Des Weiteren werden zusätzliche Möglichkeiten für die entwickelten Spezifikationsdateien sowie Ausbaumöglichkeiten des Programms diskutiert.

Der *Appendix* enthält die durchgeführte Umfrage und deren Ergebnisse. Außerdem enthält er den Quelltext des Programms, die Testfiles, die für die Evaluation in Kapitel 5 genutzt und erstellt wurden, sowie eine kurze Vorstellung der Webseite zum Projekt. Unter anderem ist dort ein Online-Tutorial verlinkt, welches die einfache Verwendung des Programms zeigt.

2 Kapitel 2

Grundlagen

In diesem Kapitel werden die Grundlagen für diese Diplomarbeit vorgestellt. Zunächst werden relevante Programmierkonzepte präsentiert, um eine Einordnung des zu erstellenden Programms zu ermöglichen. Dabei stehen vor allem die Konzepte im Vordergrund, die sich auf Roboterprogrammierung beziehen. Anschließend erfolgt eine kurze Vorstellung von Roboterframeworks und eine genauere Betrachtung des *Robot Operating System* (ROS). Die hier beschriebenen Konzepte, besonders in Bezug auf Launchfiles, sind essentiell für das finale Programm.

2.1. Relevante Programmierkonzepte

Roboterprogrammierung umfasst sowohl die Erstellung von Algorithmen, die einen Roboter veranlassen eine bestimmte Aufgabe (engl. „task“) zu erfüllen, als auch die Ebene der Anpassung und Konfiguration für die Aufgaben oder die Umwelt des Roboters (Gumbley und MacDonald, 2005). Die Herangehensweise, um ein Verhalten für Roboter zu erstellen, unterscheidet sich dabei von Fall zu Fall. So existieren spezielle Programmiersprachen, Bibliotheken oder auch Programmierschnittstellen (engl.: „*application programming interfaces*“ (APIs)), um das gewünschte Verhalten zu generieren (Macdonald *et al.*, 2003). Wird ein Roboter neu entwickelt und die Kommunikation mit der Hardware programmiert sowie die Systemressourcen durch den Entwickler verwaltet, so wird von *System-Level-Programmierung* gesprochen. Stehen Treiber und Betriebssystem bereits zur Verfügung und der Entwickler programmiert ausschließlich die Aufgabe des Roboters, so wird dies *Task-Level-Programmierung* genannt.

Für Haushaltsroboter existieren sehr einschränkende Assistenzsysteme (APIs), die die Verhaltensbeschreibungsmöglichkeiten auf ein Minimum reduzieren, um von Laien bedient werden zu können. Zum Beispiel wird der Roomba Staubsaugerroboter der Firma iRobot¹ durch ein einfaches Menü programmiert, bei dem je

¹ <http://www.irobot.com> [Stand: 07.02.2012]

2. Grundlagen

nach Modell nur Start, Stop sowie zeitgesteuertes Reinigen möglich ist. Der Vorteil ist, dass die Bedienung intuitiv ist und keine Programmiersprache erlernt werden muss.

Mehr Freiheiten bieten Roboter *Kontrollarchitekturen* bzw. *Roboterframeworks*. Diese erlauben meist das Schreiben eigener Algorithmen für das Verhalten und bieten trotz der Flexibilität großen Komfort, was die allgemeine Bedienbarkeit eines Roboters betrifft. Solche Frameworks werden in Kapitel 2.2 gesondert betrachtet.

Die meisten Freiheiten, aber auch oft den größeren Aufwand muss ein Roboterprogrammierer betreiben, wenn er auf dem *System-Level* ein Robotersystem von Grund auf neu entwickelt. In diesem Fall müssen zunächst Softwarekomponenten für die Kommunikation mit der Hardware erstellt werden und sich um die Verwaltung der Systemressourcen selbst gekümmert werden. Nachdem die Treiber für die Hardwareteile in Programmiersprachen wie C oder Assembler erzeugt wurden, kann das Verhalten für den Roboter auch in Hochsprachen wie C++ programmiert werden.

Mit steigender Abstraktion und somit wachsendem Komfort sinkt die Flexibilität, aber auch die Entwicklungszeit. Nicht jeder, der sich mit der Robotik auseinandersetzt, muss ein eigenes Robotersystem von Grund auf erstellen. Weniger versierte Benutzer, die nur Grundfunktionen eines Robotersystems benötigen, können diese durch Programmierassistenten bestimmen, da sie dafür keine Programmiersprache erlernen müssen. Für interessierte Entwickler kam in den letzten Jahren vermehrt günstige Hardware auf den Markt, wie etwa die Sensorleiste *Microsoft Kinect*² oder das umfassende *Robot Developer Kit* der Firma Willow Garage, welches das Robotersystem *Turtlebot*³ beinhaltet. Dieses Kit bietet mehr Flexibilität als anwendungsspezifische Roboter, ist günstiger als Roboter aus Forschungsprojekten und bietet mehr Möglichkeiten als Spielzeuge, wie etwa die LEGO Mindstorms Kits⁴ (Gerkey und Conley, 2011). Das Roboterframework ROS bietet hier die nötige Abstraktion, damit der Entwickler auf dem *Task-Level* agieren kann und sich auf die beabsichtigten Anwendung konzentrieren kann.

² <http://www.xbox.com/de-DE/kinect/> [Stand: 07.02.2012]

³ <http://www.willowgarage.com/turtlebot> [Stand: 07.02.2012]

⁴ <http://mindstorms.lego.com/en-us/Default.aspx> [Stand: 07.02.2012]

2.1.1. Manuelle und automatische Programmierung

Macdonald *et al.* (2003) unterscheiden zwei Hauptmethoden der Programmierung:

1. **Manuelle Programmierung** - Bezeichnet die meist textbasierte Programmierung, bei der das gewünschte Verhalten manuell eingegeben werden und der dabei entstandene Quelltext auf den Roboter übertragen, kompiliert und ausgeführt werden muss. Hiermit kann sowohl auf dem System-Level als auch auf dem Task-Level entwickelt werden. Die verwendeten Programmiersprachen unterscheiden sich sehr stark. Sowohl systemnahe Sprachen, wie Assembler oder C, als auch Hochlevelsprachen, wie Java oder C++, werden zu diesem Zweck verwendet. Einige Roboterframeworks bieten APIs, die diese Hochlevelsprachen um geeignete roboterspezifische Funktionen erweitern. Solche Erweiterungen können zum Beispiel ein Navigationsinterface sein, so dass Treiber für die Motorensteuerung nicht umständlich selbst angesprochen werden müssen, oder Eigenschaften einer Middleware, die eine Kommunikationsinfrastruktur bieten.
2. **Automatische Programmierung** - Umfasst die Techniken des *Programming by Example* (PbE), bzw. *Programming by Demonstration* (PbD) und wird für Verhaltensprogrammierung, vor allem in der Industrie, eingesetzt. Der Benutzer demonstriert, was zu tun ist und der Roboter zeichnet diese Bewegungen auf, um sie wiederholen zu können. Hierbei existieren auch verschiedene Verbesserungen, so dass ein Roboter eventuell sinnvollere Bewegungen als die demonstrierten berechnen kann oder, dass er verschiedene Abläufe autonom zeitlich anordnen kann, um ein Ergebnis zu erzielen.

2.1.2. Graphische Programmierung

Graphische Programmierung stellt einen hybriden Ansatz aus manueller und automatischer Programmierung dar. Die vom Benutzer angeordneten Symbole oder Ikone werden automatisch in Quelltext übersetzt (Macdonald *et al.*, 2003, Biggs und Macdonald, 2003). Es wird somit manueller Input benötigt, um Aktionen und den Programmfluss zu beschreiben. Für den Benutzer bedeutet dies ein höheres Maß an Kontrolle gegenüber rein automatischer Programmierung und eine einfachere Verwendung als bei textbasierten Systemen (Bischoff *et al.*, 2002). Neben dem Vorteil der einfacheren Verwendung, der häufig auf Kosten der Flexibilität

2. Grundlagen

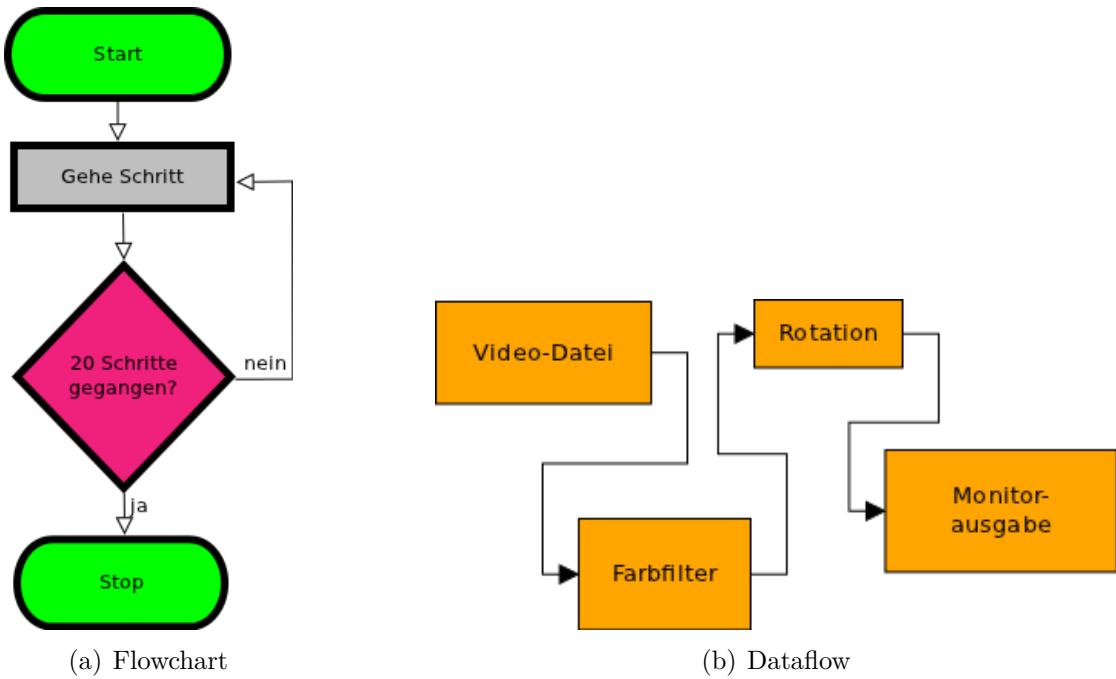


Abbildung 2.1.: (a) Flowchart-Darstellung eines einfachen Algorithmus: Gehe so lange vorwärts, bis du 20 Schritte getan hast!
 (b) Datenflussdiagramm eines einfachen Algorithmus: 1) Wende einen Farbfilter auf das aktuelle Bild der Videodaten an! 2) Drehe das Bild! 3) Gib es auf dem Bildschirm aus!

gegenüber textbasierter Programmierung geht, bietet graphische Programmierung eine Strukturierung, die eine bessere Übersicht über das zu erschaffende Programm mit sich bringt. Typischerweise wird graphische Programmierung eher für einzelne Programme als für ganze Robotersysteme eingesetzt (Biggs und Macdonald, 2003). Allerdings sind auch graphische Systembeschreibungen in solchen Umgebungen möglich, wenn man die Abstraktion erhöht und Komponenten statt Anweisungsblöcken als zentrale Einheit verwendet. Folgende sind die Hauptdarstellungsformen der graphischen Programmierung (auch *ikon-basierte Programmierung* genannt):

- *Programmablaufpläne (PAP)* oder *Flussdiagramme* (engl.: „flow charts“) wie in Abbildung 2.1 (a). Hiermit werden Abläufe in einem Algorithmus skizziert. Der sogenannte Kontrollfluss läuft dabei über Knoten, die den Instruktionen des Algorithmus entsprechen. Auch spezielle Knoten, wie Schleifen und andere Kontrollkonstrukte, sowie Start- und Endknoten, sind hierbei möglich.

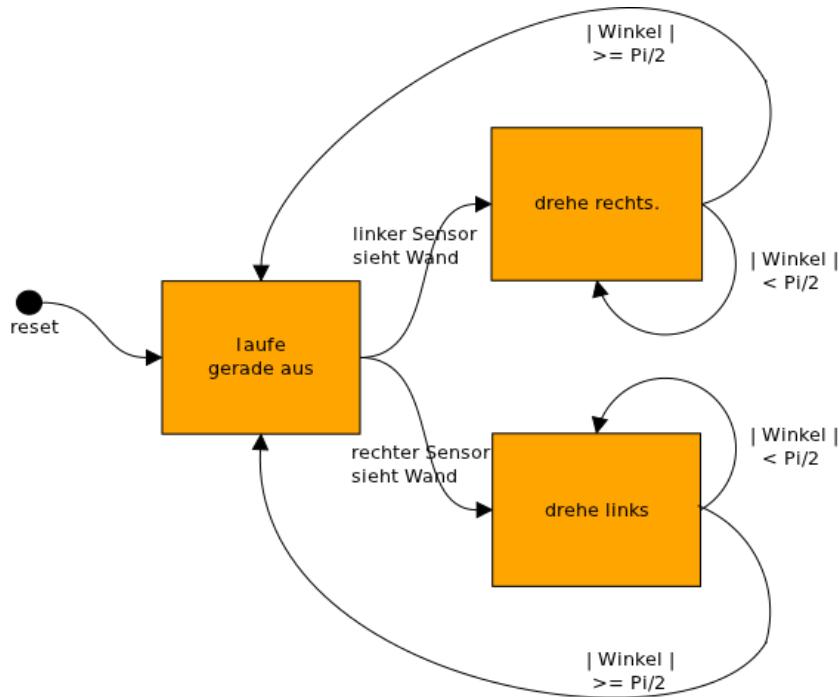


Abbildung 2.2.: State Machine-Darstellung eines einfachen Algorithmus: „1) Beuge dich vorwärts! 2.a) Wenn du links eine Wand siehst, drehe dich 90 Grad nach rechts oder 2.b) Wenn du rechts eine Wand siehst, drehe dich 90 Grad nach links! Fahre fort mit Punkt 1!“

- *Endliche Automaten* (engl.: „state machines“), dargestellt in Abbildung 2.2. Jedes Verhalten bildet einen Zustands-Knoten und gerichtete Pfeile zeigen die möglichen Zustandsübergänge für den Fall an, dass die mit ihnen verknüpften Bedingungen erfüllt sind.

Als eine besondere Form der Flussdiagramme sind die *Datenflussdiagramme* (engl.: „data flow graphs“) zu nennen, bei denen der Weg der Daten durch die verschiedenen Programmteile im Vordergrund steht und nicht der Kontrollfluss (siehe Abbildung 2.1 (b)). Die Programmteile werden hierbei als *Knoten* (engl.: „nodes“) mit genau definierten Eingängen (engl.: „Inputs“) und Ausgängen (engl.: „Outputs“) betrachtet. Über gerichtete Kanten zwischen Outputs und Inputs wird der Datenfluss vorgegeben. Quellknoten, zum Beispiel Sensoren, liefern die Daten, die von den Verarbeitungsknoten weiterverarbeitet werden, sobald Daten an deren Input verfügbar sind.

2.1.3. Komponentenbasierte Softwareentwicklung

Die Fortschritte in der Hardwaretechnologie, die sich schnell ändernden Anforderungen an Programme und der Trend hin zu verteilten Systemen und verteiltem Rechnen (engl.: „distributed computing“) benötigen anpassbare Software, die leicht auf verschiedene Hardware- und Softwareplattformen portierbar ist (Brooks *et al.*, 2005). Programme müssen häufig mit anderen Programmen interagieren können und um neue Funktionalitäten erweiterbar sowie gut wartbar sein (Brugali und Scandurra, 2009, Brugali und Shakhimardanov, 2010, Zwintzscher, 2004). Aus diesem Grund wurden einige aufeinander aufbauende Konzepte entwickelt, die zur komponentenbasierten Softwareentwicklung führten.

Die *Objektorientierte Programmierung* (engl.: „Object-oriented Programming (OOP)“) bietet einige Vorteile gegenüber prozeduraler Programmierung. Zu den wichtigsten Vertretern objektorientierter Sprachen zählen *C++*, *Java* und *Smalltalk*. Ihre Grundidee ist, Daten (Variablen) und Methoden (Funktionen) möglichst eng zu einer Klasse zusammenzufassen. Eine Instanz einer Klasse bildet ein Objekt. Die Kapselung durch Klassen sorgt dafür, dass Methoden von Objekten einer Klasse nicht die Daten von Objekten anderer Klassen ohne weiteres manipulieren können. Somit erhält man eine Technik, die die Flexibilität und Wiederverwendbarkeit von Quelltext erhöht. Die Flexibilität von *Objektorientierter Programmierung* wird jedoch dadurch eingeschränkt, dass sie auf die Verwendung von jeweils einer Programmiersprache und auf den Adressraum von einem zugrunde liegendem System beschränkt ist.

Die *Verteilte Objekt Technologie* (engl.: „Distributed Object Technology“) ist im Gegensatz dazu in der Regel sprachenunabhängig, da hierfür einzelne Objekte in unterschiedlichen Objektorientierten Sprachen erzeugt sein können. Sie erlaubt den Aufruf von nicht lokalen Prozeduren (engl.: „Remote Procedure Call (RPC)“) und somit die Verwendung mehrerer Adressräume (Plasil und Stal, 1998). Bei *Publish/Subscribe*-Protokollen veröffentlichen Komponenten Daten, die andere Komponenten erwarten. Typischer Weise übernimmt ein zentraler Prozess die Zuordnung von Publishern und Subscribers. Bekannte Publish/Subscribe-Middlewares sind *Real-time Innovations* (RTI) oder die *Inter Process Communication* (IPC) der Carnegie Mellon Universität, welche allerdings auch das *Client/Server*-Modell unterstützt. Viele solcher Middlewares verwenden XML-Beschreibungen als Datendefinitionen für die Übermittlung, da sie einfach zu verwenden sind und nur wenig Quelltext für die Kommunikation zwischen den Komponenten benötigen (Siciliano und Khatib, 2008). *XML-RPC* (Extensible Markup

Language Remote Procedure Call) ist geeignet, um auf verteilten Systemen Methodenaufrufe zu t tigen. Die systemunabh ngige Daten bertragung erfolgt  uber das Hypertext Transfer Protocol (HTTP), wobei die Daten in XML kodiert sind.

Die von der Object Management Group (OMG) entwickelte Middleware *CORBA* beinhaltet eine Interface Definition Language (IDL) und ein Application Programming Interface (API), welches dem Programmierer erlaubt Client/Server-Objektverbindungen  ber einen Object Request Broker (ORB) zu erstellen (Object Management Group, 2011). Der ORB ist ein Vermittler f r die Kommunikation von Objekten in einem verteilten System und erlaubt dadurch die  bermittlung von Parametern, den Aufruf von Methoden und das  bermitteln von Ergebnissen. Vorhandene Komponenten aus verschiedenen Systemen k nnen durch die Verwendung von Interface-Wrappern in die CORBA-Welt integriert werden. Das Interface einer Komponente wird hierf r in der IDL definiert. Zum Zusammenf gen der Komponenten wird anschlie end eine Programmiersprache wie C, C++ oder Java ben tigt, weshalb CORBA als objektorientierte Middleware eingestuft wird und nicht als Kompositionssystem oder Komponentenprogrammiersprache (Gill und Smart, 2002, Henning, 2008, Siciliano und Khatib, 2008).

Komponenten

Komponenten (engl.: „Components“) stellen einen weiteren Entwicklungsschritt gegen ber Objekten dar und bieten noch mehr Abstraktion. W hrend Objekte erst zur Laufzeit existieren, sind Komponenten Bin rdateien, die direkt verwendet werden k nnen (Brooks *et al.*, 2005). Sie bieten eine M glichkeit die Objektimplementierungen zu Paketen zu schn ren und so ihre Verwendung zu vereinfachen. Diese gepackten Objekte, zum Beispiel aus Sun *Enterprise JavaBeans* (EJB) oder Microsoft *Component Object Model* (COM) k nnen auf verteilten Systemen agieren und m ssen instantiiert werden, um verwendet zu werden. Eine Komponente fungiert dabei abstrahierend wie eine Black-Box, indem sie Implementierungsdetails versteckt. Komponenten bieten durch ihre hohe Abstraktion Portabilit t und die F higkeit mit anderen Komponenten auf einer programmiersprachenunabh ngigen Ebene zusammen zu wirken. Des Weiteren sind sie meist fein granuliert und somit gut wartbar. Sie besitzen die M glichkeit der hierarchischen Anordnung, wodurch sie in Verbindung mit anderen Komponenten neue Komponenten bilden k nnen. Dies wird durch ein wohldefiniertes, offenes Interface erm glicht, welches ihre Funktionalit t nach au en anbietet und Funktionalit t anderer Komponenten importieren kann. Ein System aus Komponenten wird durch ein Verbinden von

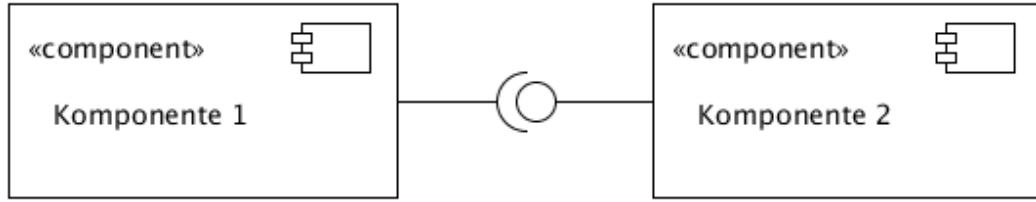


Abbildung 2.3.: UML-Diagramm einer Komponentenverknüpfung über Interfaces.
Komponente 1 bietet ein Interface an, welches *Komponente 2* anfordert.

anfordernden mit anbietenden Interfaces (siehe Abbildung 2.3) erreicht. Programme, die aus Komponenten zusammengesetzt sind, haben den Vorteil, dass sie so flexibel sind, dass einzelne Komponenten ausgetauscht oder rekonfiguriert werden können. Je nach System ist das selbst zur Laufzeit möglich.

Der *Komponentenentwickler* besitzt in der Regel Spezialwissen über die Domäne der jeweiligen Komponente, die er erstellt. Auf diese Weise entstehen qualitativ hochwertige Module, die der *Anwendungsentwickler* zusammensetzen und für seinen konkreten Anwendungskontext anpassen, konfigurieren und mit zusätzlichen Funktionen erweitern kann (Zwintzscher, 2004). Durch die Kapselung der Funktionalität einer Komponente, wird ihre Komplexität vom Anwendungsentwickler ferngehalten und dieser kann sich auf den beabsichtigten Anwendungsfall konzentrieren.

Middlewares

Die meisten neueren Roboter-Softwareframeworks verwenden einen komponentenbasierten Entwicklungsansatz (Shakhimardanov *et al.*, 2011). Eine einzelne Komponente ohne ein *Komponentenframework* (Middleware) ist im allgemeinen nicht lauffähig oder verfehlt zum mindesten den Sinn von komponentenbasierter Softwareentwicklung. Eine Middleware legt fest, wie Komponenten verbunden werden können, indem sie die Menge und Art der Interfaces sowie zugehörige Regeln definiert. Des Weiteren definiert sie den für die Komposition notwendigen Quelltext.

Schnittstellendefinitionssprachen

Schnittstellendefinitionssprachen, bzw. die Middleware-Techniken auf denen sie aufbauen, wie z.B. CORBA, bieten die Kommunikationsgrundlage für Komponenten aus verschiedenen Programmiersprachen (Plasil und Stal, 1998). Brugali und Scandurra (2009) beschreiben in ihrem Artikel, dass Spezifikationen einer Softwarekomponente der Schlüssel für die Nutzung der Komponente in größeren Softwarekonstrukten sind. In diesen Spezifikationen werden die *Schnittstellen* (Interfaces) einer Komponente definiert. Schnittstellen sind die von außen sichtbaren Teile einer Komponente und ermöglichen die Interaktion zwischen verschiedenen Komponenten. Bei der Spezifikation wird zwischen *anbietenden* (provided) und anfordernden (required) Interfaces unterschieden. Eine weitere Klassifizierung ist die Unterteilung in Daten- und Service-Interfaces (Brugali und Scandurra, 2009). Dateninterfaces (Publish/Subscribe) dienen dem Datenaustausch zwischen Komponenten und bei Serviceinterfaces (Client/Server) triggert eine Komponente eine Aktion, die eine andere Komponente ausführen soll und bekommt daraufhin eventuell ein Ergebnis zurückgeliefert.

2.2. Komponentenbasierte Roboterframeworks und -betriebssysteme

Obwohl modularer Aufbau durch Komponenten nach Jang *et al.* (2010) immer noch eine Ausnahme darstellt, ist der Trend in der Robotik hin zu heterogen verbundenen Hardwarekomponenten erkennbar (Mohamed *et al.*, 2008). Diese Komponenten werden im System durch zugehörige Softwaremodule gesteuert. Dabei können die Module von diversen Herstellern stammen, unterschiedliche Kommunikationsmechanismen nutzen und sogar in verschiedenen Programmiersprache geschrieben sein. Den Vorteilen des modularen Designs stehen somit die Integrationsprobleme bezüglich der Kommunikation, Interaktion und Konfiguration gegenüber. Solche Probleme versuchen Middlewares zu lösen, indem sie eine Abstraktionsschicht bilden und Werkzeuge anbieten, um zwischen Komponenten zu vermitteln (Mohamed *et al.*, 2008). Ziele sind oft eine Verbesserung des Entwicklungsprozesses und somit eine Reduzierung von Entwicklungszeit und -kosten. Dies wird durch die modularen Design-Mechanismen, eine High-Level-Abstraktion und komponentenbasierte Entwicklung, die die Wiederverwendbarkeit von existierenden Komponenten ermöglicht, erreicht.

2. Grundlagen

Roboterbetriebssysteme leisten darüber hinaus noch einiges mehr. Sie stellen wohlstrukturierte und gut getestete Services für oft benötigte Funktionen bereit, verwalten Ressourcen und bieten Hardwareabstraktion. Einige Roboterframework-Designer stellen darüber hinaus Softwaretools, wie grafische Editoren zum Entwickeln neuer Roboterprogramme, zur Verfügung, die die zugrunde liegende Komplexität der Kommunikationsschicht verbergen und durch geeignete Zwangsbedingungen es erschweren fehlerhafte und die Architektur verletzende Programme zu erstellen (Siciliano und Khatib, 2008).

Es existieren viele Roboterframeworks und -kontrollarchitekturen und alle haben ihre Vor- und Nachteile. Einige bekannte Vertreter, die auf unterschiedlichen Middlewares basieren, sollen hier kurz vorgestellt werden. Diese Frameworks verwenden alle ein komponentenorientiertes Konzept und sind frei verfügbar. Genauer wird anschließend auf *ROS* eingegangen, da es das Roboterbetriebssystem ist, auf dem diese Arbeit aufbaut.

Orocос

Orocос⁵ steht für „Open Robot Control Software“ und wurde als ein gemeinschaftsprojekt der KU Leuven, LAAS Toulouse und KTH Stockholm entwickelt. Es beinhaltet drei Hauptbibliotheken, Real-time toolkit (RTT), Kinematics and dynamics library (KDL) und Bayesian filtering library (BFL). Veröffentlicht wurde es unter der GPL- und LGPL-Lizenz, ist unter Linux lauffähig und verwendet als Kommunikationsmiddleware CORBA. In Orocос kann Robotersoftware in C++ erstellt werden. Das RTT bietet hierfür ein C++-Klassenframework, um Komponenten zu entwickeln, die für Echtzeitanwendungen geeignet sind (Brugali und Shakhimardanov, 2010). KDL und BFL bieten Hilfsmittel die häufig für die Steuerung verwendet werden und erleichtern so den Entwicklungsprozess.

Orca

Orca⁶ wurde an der Universität von Sydney als ein Integrationsframework für Robotersoftware entwickelt. Orca ist unter Linux und einige Teile auch unter Windows lauffähig und verwendet für die komponentenorientierte Softwareentwicklung die Middleware ICE (Makarenko *et al.*, 2006). Um die Komponenten in C++ oder Java zu entwickeln, werden einige Vorgaben, wie Kommunikationsschnittstellen, die alle Komponenten besitzen sollten, gemacht (Shakhimardanov und Prassler,

⁵ <http://orocos.org> [Stand: 07.02.2012]

⁶ <http://orca-robotics.sourceforge.net/> [Stand: 07.02.2012]

2007). Eine Konfigurationsdatei spezifiziert die zu instantiierenden Komponenten und ihre individuelle Konfiguration (Brugali und Shakhimardanov, 2010).

Player/Stage

Player/Stage⁷ ist eine Roboterplattform, die eine Infrastruktur, Treiber und einige Algorithmen für mobile Roboterprogramme bietet. Während Stage eine 2D-Simulationsumgebung darstellt, dient Player als Hardwarekomponenten-Repositoryserver und jede Komponente besteht aus Treiber und Interface (Mohamed *et al.*, 2008, Kramer und Scheutz, 2007). Die Interfaces können für eigene Programme in C, C++, Java, TCL, Lisp oder Python genutzt werden, um mit anderen Komponenten verbunden zu werden, Sensoren auszulesen oder Aktuatoren zu steuern (Gerkey *et al.*, 2003). Player läuft auf Linux, Solaris und BSD Unix.

Carmen

Carmen⁸ ist das „Carnegie Mellon Robot Navigation Toolkit“. Es bietet eine Open-Source-Sammlung von Roboter-Steuerungssoftware die in C++ geschrieben und auf Linux limitiert ist. Als Middleware wird IPC verwendet und die Architektur ist dafür ausgelegt einzelne Roboter zu steuern (Kramer und Scheutz, 2007, Namoshe *et al.*, 2008).

ROS

Das *Robot Operating System* (ROS) ist ein Open-Source-Projekt, dass von der Firma Willow Garage entwickelt und in regelmäßigen Abständen als Distribution veröffentlicht wird. Vielmehr als ein Roboterframework versteht sich ROS als ein Meta-Betriebssystem für Roboter, das unter Linux und Mac OS X betrieben werden kann. Es bietet hierfür betriebssystemähnliche Dienste, wie Hardwareabstraktion, eine Nachrichtensteuerung zwischen Prozessen, Treiber für Hardwarekomponenten, Paketmanagement und Tools für häufig benötigte Aufgaben, an (Ceriani und Migliavacca, 2010).

Der Schwerpunkt bei ROS liegt in der Wiederverwendbarkeit von Quelltext für die Roboterforschung und -entwicklung. Um dies zu ermöglichen, verfügt ROS

⁷ <http://playerstage.sf.net/> [Stand: 07.02.2012]

⁸ <http://carmen.sourceforge.net/home.html> [Stand: 07.02.2012]

2. Grundlagen

über eine Community-Website⁹, auf der die Zusammenarbeit an Software, das Teilen von Quelltext und das Veröffentlichen von Software-Paketen gefördert wird (Ceriani und Migliavacca, 2010).

Durch die Verwendung von *XML-RPC*, welches für viele Programmiersprachen implementiert wurde, können Softwarekomponenten zum Beispiel in C++, Python, Octave, Java, Lua oder LISP erstellt und trotzdem gemeinsam verwendet werden. Die Komponenten werden als *Nodes* bezeichnet und sind ohne spezielle Ausführungslogik einzeln verwendbar. Auch die Kommunikation wird sprachenneutral gehalten, indem eine einfach-gehaltene Schnittstellendefinitionssprache verwendet wird. Die Nachrichtendefinitionen, die in dieser IDL beschrieben sind, werden automatisch in die jeweiligen Programmiersprachen übersetzt (Quigley *et al.*, 2009). ROS bietet folgende Kommunikationsarten:

- asynchrones Streamen von Daten über *Topics* genannte Kanäle (siehe Kapitel 2.3.5)
- synchrone RPC-basierte Kommunikation über *Services* (siehe Kapitel 2.3.6)
- Speichern und Laden von Daten auf einem *Parameter Server* (siehe Kapitel 2.3.2)

Um die Komplexität von ROS zu reduzieren, Komponenten zu starten und zu konfigurieren oder sich die laufenden Komponenten und deren Kommunikation zu visualisieren, existieren viele Tools. Das spiegelt das, von der Unix-Philosophie stammende, Konzept in ROS wider, welches die Verwendung von einer Vielzahl von Tools anstatt einer Monolithischen Entwicklungs- und Laufzeitumgebung vorsieht (Cousins, 2010).

Ein Hauptdesignkonzept von ROS besteht laut Quigley *et al.* (2009) im Peer-to-Peer-Modell. So werden Aufgaben durch viele einzelne Prozesse gelöst, die auch auf verschiedenen Host-Systemen, das heißt auf verteilten Systemen, laufen können. Diese Prozesse können sich gegenseitig finden und miteinander kommunizieren. Durch die Verwendung von atomaren Prozessen, also kleinen Berechnungskomponenten, wird die allgemeine Entwicklungszeit verkürzt sowie die Wartung vereinfacht und die Wiederverwendbarkeit und Zusammenarbeit von Komponenten erhöht. Kleine Komponenten können besser getestet werden und sind leichter austauschbar.

⁹ <http://www.ros.org> [Stand: 14.02.2012]

2.2. Komponentenbasierte Roboterframeworks und -betriebssysteme

Auf der Entwicklungsebene kann ROS wie folgt eingeteilt werden:

1. „Main“ ist der Teil, der von Willow Garage und einigen externen Entwicklern betreut wird und generelle Tools bereit stellt
 - Packaging- und Build-Tools, die Prozesse und Daten in Pakete packen und ausführbar machen
 - Kommunikationsinfrastruktur
 - ROS-APIs für verschiedene Programmiersprachen, zum erzeugen von neuen Komponenten
 - Analyse-Tools, wie *rosnode* oder *rostopic*, die das Peer-To-Peer-Netzwerk untersuchen und auswerten
2. „Universe“ ist der Teil, der von der internationalen ROS-Community betreut wird.
 - *Bibliotheken*: Hardwaretreiber, Simulationen, ROS-gewrappte Third-Party-Software wie *OpenCV* oder *YAML-CPP*, die nicht überall über den System Paketmanager bezogen werden können, etc.
 - *Algorithmen und andere Software Module*: Robotersteuerung, Navigation, etc., die in Pakete als ROS-Ressourcen zur Verfügung stehen
 - *Anwendungsprogramme*: Zusammengesetzte Softwaremodule, die gemeinsam ein Ziel erreichen, wie z.B. Software für einen Roboter, um ein Frühstück vorzubereiten

ROS bietet bei der Programmierung von Robotern einige Vorteile:

- Wiederverwendbarkeit und Organisation von Prozessen (Nodes), Tools, Treibern, Daten, etc. in Paketen
- Quelltextunabhängige Kommunikation zwischen Komponenten sowohl über Topics (siehe Kapitel 2.3.5) genannte Kanäle nach dem Publish/Subscribe-Verfahren als auch über Service-Aufrufe nach dem Client/Server-Verfahren
- Launchfiles (siehe Kapitel 2.4), eine Konfigurationsdatei zum zentralen Starten und Parametrisieren von Komponenten

2. Grundlagen

- Verknüpfung von existierenden Komponenten zu neuen Komponenten über das Konzept der Launchfiles

Diesen Vorteilen stehen auch einige Nachteile gegenüber:

- ROS ist vor allem eine Sammlung von Tools ohne eigene Entwicklungsprogramme für neue Komponenten, oder deren Verknüpfung
- Erstellung von Paketen erfolgt ausschließlich über Kommandozeilen-Tools
- Erstellung von neuen Nodes wird durch häufig gleichen Quelltext erschwert
- Die textbasierte Erstellung von Launchfiles ist schwierig und fehleranfällig
 - Die verwendete XML-Struktur der Launchfiles ist nicht intuitiv lesbar und erfordert viel Einarbeitungszeit
 - Die Launchfiles beinhalten viele, bis auf Attribute, ähnliche Quelltextzeilen, was die Lesbarkeit und die Fehlersuche verschlechtert
 - Spezifikationen von Nodes müssen im Wiki gesucht werden

Um eigene Komponenten zu erstellen, kann der Entwickler auf allgemeine Entwicklungsumgebungen zurückgreifen und diese für ROS anpassen. Es existieren Konfigurationsanleitungen¹⁰ für Eclipse, CodeBlocks, Emacs, Vim und Netbeans. Was jedoch fehlt ist eine Entwicklungsumgebung für die Komponentenverknüpfung und Konfiguration. Es gibt keine graphische Benutzeroberfläche, mit der Launchfiles generiert oder vorhandene Konfigurationen verändert werden können, indem man Parameter oder Namen darin editiert. In der Launchfileverwendung liegt großes Verbesserungspotential, denn es existiert kein Schutz vor einer fehlerhaften Benutzung von ROS-Ressourcen. Nodespezifikationen sind nur ersichtlich, falls ein Node bereits ausgeführt wird oder falls diese Informationen vom Node-Programmierer auf der Wiki-Seite hinterlegt wurden. In jedem Fall ist die korrekte Verwendung von bereits existierenden Nodes in einem Launchfile kompliziert und mit einem hohem Aufwand für den Ersteller verbunden und erschwert dem Anfänger in der Roboterentwicklung den Einstieg. Eine Verbesserung diesbezüglich würde ROS als komponentenbasiertes Roboterframework mit einer großen Entwicklergemeinschaft leichter benutzbar machen.

¹⁰ <http://www.ros.org/wiki/IDEs> [Stand: 07.02.2012]

2.3. Konzepte und Techniken von ROS

ROS-Entwickler bieten viele fertige und getestete Ressourcen, wie Hardwaretreiber für diverse Hardwarekomponenten oder Algorithmen für verschiedene Aufgaben, an. Diese Ressourcen werden entweder über die ROS-Distribution dem Benutzer zugänglich gemacht, oder können über die Communityseite bezogen werden. Werden diese Ressourcen ausgeführt, entsteht der Berechnungsgraph, der ein verteiltes Netzwerk von Prozessen, den sogenannten Nodes, darstellt.

2.3.1. Nodes

Nodes sind Prozesse, die beliebige Berechnungen, wie etwa Motorsteuerung, Pfadplanung oder Lokalisierung, durchführen. In einem laufenden System arbeiten später viele Nodes zusammen und jeder von ihnen stellt für sich ein Software-Modul mit einem einzelnen Zweck dar. Jeder Node hat eine unabhängige Rolle und kann individuell erstellt sein. Zur Laufzeit werden diese Nodes aber lose im Berechnungsgraphen verknüpft. Für ROS spielt hierbei die Programmiersprache, die für die Erstellung eines Nodes verwendet wurde, keine Rolle. Nach Cousins (2010) gibt es ROS-Wrapper zum Schreiben von Nodes in C++, Python, C, LISP, Java, Lua und Octave. Andere Sprachen können verwendet werden, sobald dafür ein ROS-Wrapper geschrieben wurde.

Nodes können eigene Anwendungen sein oder Bibliotheken, Treiber oder Tools, die als ROS-Node in ein eigenes Softwaresystem eingebunden werden. Alle diese Nodes werden in ROS zusammen mit Message-Definitionen (siehe Kapitel 2.3.4), Parameterkonfigurationen, dem Quelltext und anderen Daten in Paketen (Packages) organisiert. Tabelle 2.1 zeigt die üblichen Daten in Paketen sowie deren Organisation innerhalb der Pakete. Die Pakete wiederum können zu Stacks zusammengefasst werden und diese letztlich zu Distributionen und werden so dem Benutzer zugänglich gemacht.

Nodelets

Nodelets sind eine spezielle Art von Nodes. Sie ermöglichen die Ausführung von mehreren Algorithmen in einem Prozess, ohne dass Daten zwischen den Algorithmen mit Hilfe der CPU kopiert werden müssen (engl.: „zero copy transport“).

<i>Package-Inhalt</i>	<i>vorgegebene Ordner</i>
Quelltext	/src
Header-Deklarationen	/src
Scripts	/src
Message Definitionen	/msg
Service Definitionen	/srv
Manifest-Datei	/
Konfigurationsdateien	beliebig
Launchfiles	beliebig
Metadaten	beliebig
Node-Binaries	/bin oder /node
...	

Tabelle 2.1.: Inhalt eines ROS-Package und zugehörige Standard-Ordner

Dynamic Reconfigure Nodes

Dynamic Reconfigure Nodes sind Nodes, die zur Laufzeit partiell konfiguriert werden können. Hierbei stellen diese Nodes eine Teilmenge ihrer Parameter so bereit, dass externe Programme diese inklusive Namen, Typen und dem Intervall von gültigen Werten (engl.: „range“ anfordern und ändern können.

2.3.2. Master Service

Der Master Service (*roscore*) ist der zentrale XML-RPC-Server und muss stets als erstes gestartet werden. Er ermöglicht die Namensregistrierung der Nodes sowie deren Auffindung im Berechnungsgraphen. Des Weiteren werden in ihm auch die Topic- (siehe Kapitel 2.3.5) und Service-Registrierungen (siehe Kapitel 2.3.6) gespeichert. Er macht die Kommunikationspartner miteinander bekannt, so dass die Kommunikation selbst über TCP/IP oder UDP nach dem Peer-to-Peer-Modell, also direkt zwischen den Knoten, erfolgen kann. Über den Master Service werden somit keine weiteren Daten direkt geleitet.

Parameter Server

Der Parameterserver ermöglicht das zentrale Speichern von persistenten Konfigurationsparametern und anderen Daten. Hierbei werden die Daten als Key-Value-Paare gespeichert. Der Parameter Server ist ein Teil des Master Service und wird automatisch gestartet.

2.3.3. Parameter

Parameter sind, solange der Master Service (siehe Kapitel 2.3.2) läuft, persistente Daten, wie Konfigurationen und Initialisierungseinstellungen, die auf dem Parameter Server gespeichert werden.

Parameterfiles

Parameter können, neben einer direkten Verwendung in ROS, auch in sogenannten *Parameterfiles*, die YAML-formatiert (Ben-Kiki *et al.*, 2009) vorliegen müssen, gespeichert sein. Diese können später in Launchfiles (siehe Kapitel 2.4) aufgerufen und somit auf den Parameter Server übertragen werden. Dies geschieht wie in Kapitel 2.4.2 beschrieben. Auf diese Weise können Parametrisierungen wiederholt und eine erneutes manuelles Konfigurieren vermieden werden.

2.3.4. Messages

Die Kommunikation zwischen Nodes erfolgt über Messages. Messages sind eine Datenstruktur aus typisierten Feldern. Die Standard-Grundtypen und ihre Äquivalente in C++ und Python sind in Tabelle 2.2 abgebildet. Außerdem sind Arrays aus diesen Grundtypen, wie in Tabelle 2.3 und Konstanten erlaubt.

Bags

Bags erlauben das Speichern und Wiedergeben von ROS-Messagedaten. Auf diese Weise kann zum Beispiel während der Entwicklung und dem Testen von Softwaredaten immer auf gleichen Sensordaten gearbeitet werden.

2. Grundlagen

<i>Grundtyp</i>	<i>Einheit</i>	<i>C++</i>	<i>Python</i>
bool	unsigned 8-bit int	uint8_t	bool
int8	signed 8-bit int	int8_t	int
uint8	unsigned 8-bit int	uint8_t	int
int16	signed 16-bit int	int16_t	int
uint16	unsigned 16-bit int	uint16_t	int
int32	signed 32-bit int	int32_t	int
uint32	unsigned 32-bit int	uint32_t	int
int64	signed 64-bit int	int64_t	long
uint64	unsigned 64-bit int	uint64_t	long
float32	32-bit IEEE float	float	float
float64	64-bit IEEE float	double	float
string	ascii string (4)	std::string	string
time	secs/nsecs signed 32-bit ints	ros::Time	rospy.Time
duration	secs/nsecs signed 32-bit ints	ros::Duration	rospy.Duration

Tabelle 2.2.: Grundtypen für Nachrichten in ROS und ihre Entsprechungen in C++ und Python (nach Quelle: <http://ros.org/wiki/msg> [Stand: 07.02.2012])

<i>Array-Typ</i>	<i>C++</i>	<i>Python</i>
feste Länge	boost::array, std::vector	tuple, list
variable Länge	std::vector	tuple, list
uint8[]	std::vector	string
bool[]	std::vector<uint8_t>	list(bool)

Tabelle 2.3.: Arrays aus Grundnachrichtentypen in ROS und ihre Entsprechungen in C++ und Python (nach Quelle: <http://ros.org/wiki/msg> [Stand: 07.02.2012])

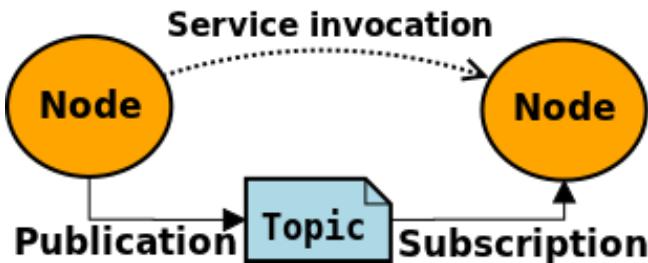


Abbildung 2.4.: Synchrone (oben) und asynchrone (unten) Kommunikation in ROS, Quelle: <http://www.ros.org/wiki/ROS/Concepts> [Stand: 07.02.2012]

2.3.5. Topics

Um den Inhalt der Messages zu identifizieren und die Kommunikation zu organisieren, werden die Messages durch ein Transportsystem, den Topics, geleitet (siehe Abbildung 2.4). Dabei kann ein Node Messages in ein Topic schreiben und ein anderer Node liest über das Topic genau diese Messages aus. Das Transportsystem der Topics verfolgt das asynchrone Publish/Subscribe-Interaktionsparadigma, welches das Entkoppeln von verteilten Komponenten ermöglicht. Messages werden zwischen Publishern und Subscribers ausgetauscht ohne, dass die interagierenden Nodes sich kennen (Brugali und Shakhimardanov, 2010). Durch dieses Modell ist auch One-To-Many- und Many-To-Many-Kommunikation möglich. Es kann also mehrere Publisher und/oder Subscriber für ein und das selbe Topic geben.

2.3.6. Services

Über Services ist auch eine synchrone Kommunikation nach dem Client/Server-Modell möglich. Hierfür bietet ein Node einen Service an, den ein anderer Node anfordert und anschließend auf die Antwort wartet (siehe Abbildung 2.4). Im Gegensatz zu Topics (siehe Kapitel 2.3.5) kann hier immer nur ein Node einen bestimmten Service anbieten, der von beliebig vielen Nodes angefordert wird. Hier ist also nur eine One-To-Many-Kommunikation möglich.

2.3.7. Names und Remapping

Namen (engl.: „Names“) und *Namensräume* (engl.: „Namespaces“) sind in ROS unerlässliche Konzepte, die ähnlich einem Dateisystem aufgebaut sind. Durch sie erhalten komplexe Systeme eine übersichtliche Struktur und ermöglichen so eine Kommunikation zwischen einzelnen Ressourcen im Berechnungsgraphen.

Es existieren vier Typen von Ressourcennamen im Berechnungsgraphen:

1. „base“ - alle Ressourcen-Namen, die ohne einen gesonderten Namespace gestartet werden, sind Basisnamen: `/node_name`
2. „relative/name“ - Namen, die so ausgewertet werden, sind relativ zum Namespace in dem sie gestartet werden: `/ns1/node1` hat den Namespace `/ns1`, innerhalb dieses Namespace würde der Name `node2` bei der Ausführung zu `/ns1/node2` aufgelöst
3. „/global/name“ - diese Namen werden mit absolutem Namenspfad gesetzt: `/ns1/node2`
4. „~private/name“ - hiermit wird der Node-Name in einen Namespace verwandelt: Wenn `node1` im Namespace `/ns1/` die Ressource `~foo` verwendet, hat er den privaten Namespace `/ns1/node1` und die Ressource wird zu `/ns1/node1/foo` aufgelöst

Listing 2.1: Remapping

```
1 rosrun rospack_tutorials talker chatter:=wg/chatter
```

Da jede Ressource im Berechnungsgraphen in einem Namespace definiert ist und diesen mit anderen Ressourcen teilen kann, können isolierte Bereiche in Systemen erzeugt werden, die verhindern, dass Topics (siehe Kapitel 2.3.5) fälschlicherweise mit anderen Ressourcen verbunden werden. Wenn ein Node in einen anderen Namespace gesetzt wird, so wirkt sich das auf alle Namen innerhalb des Nodes aus. Um nicht nur auf Ressourcen innerhalb oder oberhalb des eigenen Namespaces zugreifen zu können, existiert eine Möglichkeit Namen zur Laufzeit umzubenennen. Diese Möglichkeit wird als *Remapping* bezeichnet. Das Remapping kann entweder während des Startens der Ressource direkt, wie in Listing 2.1 gezeigt, wobei die Ressource 'chatter' in '/wg/chatter' umbenannt wird, oder innerhalb von Launch-

files (siehe Kapitel 2.4) durch einen `<remap>`-Tag erwirkt werden. Auf diese Weise können kompilierte Programme zur Laufzeit verschiedene Rollen im Berechnungsgraphen besitzen (Ceriani und Migliavacca, 2010).

2.4. Launchfiles

Jeder Node kann über die Kommandozeile einzeln gestartet und konfiguriert werden. Je nach Robotersystem und Aufgabe kann das schnell komplexe Züge annehmen, da für ein System beliebig viele Nodes gestartet und konfiguriert werden können. Das System besteht letztlich aus allen Komponenten, wie Hardwaretreibern, zugrunde liegenden Funktionen, wie Navigation und Hindernisvermeidung, Bilderkennung oder Lokalisierung und der Implementierung der eigentlichen Aufgabe. Ein solches System muss in der Regel mehr als nur einmal ausgeführt werden. Damit bei einem erneuten Aufruf des gesamten Softwaresystems nicht alles manuell erneut gestartet werden muss, existiert in ROS das Konzept der Launchfiles, die Start- und Konfigurationsinformationen beinhalten und über das Tool `roslaunch` gestartet und abgearbeitet werden. Sie stellen dabei eine XML-Beschreibung des Berechnungsgraphen dar und ermöglichen auch die Parametrisierung der Nodes sowie eine Beschreibung des Robotersystems auf dem das Softwaresystem ausgeführt werden soll. Beim Ausführen eines Launchfiles werden die in ihm beschriebenen Nodes auf einem optional spezifizierten Host-System instantiiert und können zentral beendet werden. Mehrere Instanzen eines Nodes können durch verschiedene Namen und Namespaces gestartet werden. Auch Parameter können mehrfach in einem Launchfile vorkommen, allerdings wird der Parameter dem letzten Eintrag nach auf dem Parameterserver gespeichert.

Launchfiles bieten dem Benutzer folgende Vorteile:

- Verbinden mehrerer Nodes durch eine einzelne Datei
- Parametrisierung der Nodes, was Nodes an die konkreten Anforderungen anpasst
- Hierarchische Verknüpfung von mehreren Launchfiles, was zu übersichtlicheren Dateien führt und die Wiederverwendbarkeit von Launchfiles erhöht
- Nodes können automatisch neu gestartet werden, wenn sie fehlerhaft beendet werden

- Umbenennen von Nodennamen, Namespaces, Topics und anderen Ressourcen ohne neu kompilieren zu müssen
- Wiederverwendung von Launchfiles auf mehreren Systemen oder für einen erneuten Aufruf des Berechnungsgraphen

2.4.1. Launchfile-Elemente

Die Elemente in XML-Dateien werden Tags genannt. Die Launchfile-Tags werden sowohl inhaltlich als auch strukturell auf verschiedene Weisen genutzt. Sie besitzen alle unterschiedliche Pflichtattribute und zusätzliche optionale Attribute, bei denen, falls man sie nicht setzt, Default-Werte angenommen werden. Listing 2.2 zeigt die syntaktische Form eines Launchfile-Tags. Hierbei ist wichtig, dass der „Elementname“ das erste Wort nach dem voran gestellten „<“ ist und nach den Attributen, die beliebig permutiert sein dürfen, ein abschließendes „/>“ steht.

Jeder Tag besitzt zusätzlich zu den im weiteren Verlauf aufgeführten Attributen in jedem Fall noch die folgenden beiden Attribute als optionale Attribute:

- *if='value'*, wertet den Tag beim Start aus, falls 'value' den Wahrheitswert Wahr annimmt
- *unless='value'*, falls 'value' nicht Wahr ist, wird der Tag beim Start beachtet

Listing 2.2: Launchfile-Tag Struktur

```
1  <Elementname Pflichtattribut_1='Wert1' ... Pflichtattribut_n='Wert2' Optional-Attribut_1  
   ='Wert3' ... Optional-Attribut_n='Wert4' />
```

<launch>

Jeder Launchfile beginnt mit einem *<launch>*-Tag und schließt mit einem *</launch>*-Tag. Weitere Funktionen hat der *<launch>*-Tag nicht. Das erlaubte optionale Attribut innerhalb dieses Tags ist:

- *deprecated='deprecation message'*, welches den Benutzer beim Starten des Launchfiles warnt, dass diese Datei veraltet ist

<group>

<*group*>-Tags, die direkt dem <*launch*>-Tag untergeordnet sind und Container für andere Tags darstellen, dienen der Strukturierung und können zwei optionale Attribute besitzen:

- *ns='namespace'*, welches den Ressourcen in der Gruppe einen gemeinsamen Namespace zuordnet
- *clear_params='true/false'*, das nur bei gesetztem ns-Attribut gesetzt werden darf und alle Parameter im Gruppen-Namespace löscht

<node>

Die <*node*>-Tags werden zum starten von Nodes verwendet und haben folgende Pflichtattribute:

- *pkg='mypackage'*, was das Package, zu dem der Node gehört, bestimmt
- *type='nodetype'*, welches die zum Node korrespondierende, ausführbare Datei selbst bezeichnet
- *name='nodename'*, womit der Name gesetzt wird

Optionale Attribute sind:

- *args='arg1 arg2 arg3'*, ermöglicht das Übergeben von Argumenten
- *machine='machine-name'*, was dafür sorgt, dass der Knoten auf der angegebenen Maschine gestartet wird
- *respawn='true/false'*, startet bei 'true' den Knoten selbstständig neu, falls dieser stoppt
- *required='true/false'*, beendet bei 'true' die Ausführung des gesamten Launchfiles, falls dieser Knoten beendet wird
- *ns='foo'*, startet den Knoten in einem Namespace
- *clear_params='true/false'*, löscht alle Parameter im Node-Namespace
- *output='log/screen'*, std{out,err} werden im Falle von 'screen' auf dem Bildschirm ausgeben und im Falle von 'log' in einen Logdatei in

\$ROS_HOME/log geschrieben

- *cwd='ROS_HOME/node'*, setzt im Falle von 'node' das Arbeitsverzeichnis des Nodes auf das Verzeichnis der ausführbaren Datei, im Falle von 'ROS_HOME' ist es das ROS-Verzeichnis
- *launch-prefix='prefix arguments'*, ermöglicht es Kommandos vor dem Start des Knotens auszuführen

<test>

Mit dem <test>-Tag wird prinzipiell auch ein Node gestartet. Dem entsprechend hat dieser die gleichen Pflicht-Attribute, wobei statt *name="nodename"* *test-name="testname"* verwendet werden muss. Unterschiede gibt es bei den optionalen Attributen, so ist es nicht möglich ein *respawn="true"* einzufügen, da Test-Nodes terminieren müssen. Des Weiteren existiert kein *output*-Attribut, da hier andere Log-Mechanismen verwendet werden. Das *machine*-Attribut wird einfach ignoriert. Hinzu kommen dafür:

- *time-limit='60.0'*, was die Sekunden angibt, bis der Test als fehlerhaft gewertet wird
- *retry='0'*, womit die Zahl der Startversuche angegeben wird, bevor ein Test als fehlerhaft bewertet wird

<remap>

Durch den <remap>-Tag kann auf strukturierte Weise eine Umbenennung von Node-Argumenten erfolgen. Bei Topic Umbenennungen muss darauf geachtet werden, dass Original- und Ziel-Topic vom gleichen Typ sind. Folgende Attribute werden für das Remapping verwendet:

- *from='original-name'*, der Name, der geändert werden soll
- *to='new-name'*, der Name, der neu gelten soll

<param>

Parameter, die durch diese Tags angegeben werden, werden auf dem Parameter-Server gespeichert bevor ein Knoten gestartet wird. Auch für <param>-Tags existieren verschiedene Benutzungsszenarien, mit denen Parameter auf dem Server

gespeichert werden können. In jedem Fall muss ein `name='namespace/name'`-Attribut gesetzt werden, anschließend muss eine der folgenden Optionen gewählt werden:

1. Angabe von einem `value='value'`-Attribut, welches den Wert des Parameters angibt mit optionaler Festlegung des Typs mit Hilfe eines `type='str/int/double/bool'`-Attributes.
2. Angabe eines `textfile='$(find pkg-name)/path/file.txt'`-Attributs, womit ein String des Inhalts einer Datei gespeichert wird.
3. Durch Angabe eines `binfile='$(find pkg-name)/path/file'`-Attributs, wird der Inhalt einer Datei als base64-kodiertes XML-RPC-Binärobject gespeichert.
4. Angabe eines `command='$(find pkg-name)/exe'`-Attributs, wodurch ein String mit der Ausgabe des Kommandos geschrieben wird.

<rosparam>

Durch die Angabe von `<rosparam>`-Tags können YAML-formatierte Dateien oder Text mit Parametern in den Parameter-Server geladen, von diesem geschrieben oder einzelne Parameter gelöscht werden. Für die Attribute gibt es zwei verschiedene Usecases:

1. Ist ein `file='$(find pkg-name)/path/foo.yaml'`-Attribut, mit dem Pfad zu einer YAML-Datei angegeben, so kann optional ein `ns='namespace'`-Attribut angegeben werden, das den Namespace der Parameter festlegt, sowie ein `command='load/dump'`-Attribut, wobei 'load' als Standard-Wert angenommen wird.
2. Ist ein `param='param-name'`-Attribut gesetzt, so kann auch das Attribut `command='delete'` vorhanden sein, welches eine Löschung des Parameters 'param-name' bewirkt. Alternativ kann zwischen einem öffnenden und einem Schließenden `<rosparam>`-Tag YAML-formatierter Text stehen.

<machine>

Durch einen `<machine>`-Tag wird das System beschrieben, auf dem die Nodes gestartet werden sollen. Er beinhaltet folgende Pflichtattribute:

- `name='machine-name'`, womit das Systems benannt wird Dieser Name kor-

respondiert auch mit dem *machine*-Attribute von *<node>*-Tags

- *address='server.uni-bonn.de'*, welches die Netzwerkadresse des Systems definiert

Als optionale Attribute existieren:

- *ros-root='/path/to/ros/root/'*, zur Definition des ROS_ROOT-Verzeichnisses
- *ros-package-path='/path1:/path2'*, zum Setzen des ROS Paket-Pfades des Systems
- *default='true/false/never'*, um das System als 'default' zu setzen oder nicht
- *user='username'*, für das Setzen eines SSH-Benutzernamens, um sich an einem System anzumelden
- *password='passwhat'*, für das Setzen eines SSH-Benutzerpassworts, um sich an einem System anzumelden
- *timeout='10.0'*, zum Festlegen der Zeit in Sekunden, bis ein Start des Launchfiles auf dem System als fehlerhaft bewertet wird

<include>

Mit dem *<include>*-Tag können weitere Launchfiles in einen Launchfile eingebunden werden. Die Pflichtangabe stellt somit das Attribut *file='\$(find pkg-name)/path/filename.xml'* dar, womit der Pfad zu der zu inkludierenden Datei gesetzt wird. Als optionale Attribute existieren noch:

- *ns='foo'*, welches den Inhalt der Datei in einen Namespace importiert
- *clear_params='true/false'*, womit alle Parameter im angegebenen Namespace des *<include>*-Tags gelöscht werden und welches das Setzen eines *ns*-Attributs voraussetzt

<arg>

Durch *<arg>*-Tags kann die Wiederverwendbarkeit und Konfigurierbarkeit von Launchfiles erhöht werden, indem Werte über die Kommandozeile, über *<include>*-Tags oder durch andere Dateien übergeben werden. Diese Argumente sind nicht global und können in folgenden Usecases verwendet werden:

1. Angabe von `<arg name="foo" />`, deklariert das Vorhandensein eines Arguments und kann durch ein Kommandozeilen-Argument oder einen `<include>` spezifiziert werden.
2. Angabe von `<arg name="foo" default="1" />`, was zusätzlich einen Default-Wert angibt, der wie bei 1. überschrieben werden kann.
3. Angabe von `<arg name="foo" value="bar" />`, wodurch ein konstanter, überschreiben Wert gesetzt wird.

`<env>`

Mit `<env>`-Tags werden Umgebungsvariablen gesetzt. Die Verwendung setzt folgende Pflichtattribute voraus:

- `name='environment-variable-name'`, zur Benennung der Umgebungsvariablen
- `value='environment-variable-value'`, zum setzen des Wertes der Variablen

2.4.2. Launchfile-Syntax

Tags können ineinander verschachtelt vorkommen, wobei kein Tag einen Tag vom gleichen Typ beinhalten darf. Tabelle 2.4 stellt die Beziehungen der Tags in Launchfiles dar und zeigt, dass vier hierarchische Klassen von Tags existieren. Die oberste Klasse stellt der `<launch>`-Tag dar, der alles Weitere direkt oder indirekt beinhaltet. Die zweite Klasse sind `<group>`-Tags, die als direkte Kind-Tags innerhalb eines `<launch>`-Tags vorkommen können und außer `<launch>`-Tags ebenfalls alle anderen Tags beinhalten können. Die anderen Tags, namentlich `<node>`, `<test>`, `<param>`, `<rosparam>`, `<machine>`, `<include>`, `<remap>`, `<env>` und `<arg>`, bilden die dritte Klasse. Einige Tags der dritten Klasse können auch auf der untersten hierarchischen Ebene Vorkommen und somit ihrerseits innerhalb von Tags der dritten Klasse erscheinen. So sind `<env>`, `<remap>`, `<param>` und `<rosparam>` innerhalb von `<node>`- oder `<test>`-Tags möglich, `<env>` innerhalb von `<machine>`-Tags und `<env>` und `<arg>` innerhalb von `<include>`-Tags. In solchen Fällen führt die Verwendung eines `<rosparam>`-Tags innerhalb eines `<node>`-Tags Beispielsweise dazu, dass dort angegebene Parameter als private Parameter behandelt und nur diesem Node zugeordnet werden.

2. Grundlagen

	<i>launch</i>	<i>group</i>	<i>node</i>	<i>test</i>	<i>machine</i>	<i>include</i>
launch						
group	X					
node	X	X				
test	X	X				
machine	X	X				
include	X	X				
param	X	X	X	X		
rosparam	X	X	X	X		
remap	X	X	X	X		
env	X	X	X	X	X	X
arg	X	X				X

Tabelle 2.4.: Mögliche Positionen von Tags in Launchfiles (Spalten entsprechen möglichen übergeordneten, Zeilen eventuell untergeordneten Tags)

Im Falle einer Verschachtelung von Tags werden die übergeordneten Tags erst durch ein abschließenden Tag beendet, wenn alle untergeordneten Tags eingefügt wurden. In Listing 2.3 wird der übergeordnete Tag '`<launch>`' in *Zeile 1* nicht direkt mit '`/>`' beendet, sondern erst nach den untergeordneten Tags in den *Zeilen 2-5*. Dies geschieht durch einen erneuten Aufruf des Tags mit einem vorangestellten Tag-Schließungs-Symbol '`</launch>`' in *Zeile 6*. Diese Verschachtelung kann sich in ROS-Launchfiles, wie in XML üblich, über mehrere Ebenen erstrecken. Die Zahl der Ebenen entspricht der Anzahl der hierarchischen Klassen. So zeigt das Listing 2.3 die maximale Verschachtelungstiefe eines Launchfiles, wobei die *Zeilen 1-4* auch entsprechend Elemente aus den Klassen 1 bis 4 repräsentieren.

Listing 2.3: Verschachtelung von Tags

```

1 <launch>
2   <group ns='namespace'>
3     <node name='mimic' pkg='turtlesim' type='mimic'>
4       <remap from='output/command_velocity' to='/turtlesim2/turtle1/command_velocity' />
5     </node>
6   </group>
7 </launch>

```

2.4.3. Beispiele

Im folgenden sollen einige Beispiele für Launchfiles gezeigt werden. Diese sollen einige Grundprinzipien von Launchfiles vorstellen, decken aber nicht alle Möglichkeiten aus semantischer und syntaktischer Sicht ab:

Beispiel 1

Listing 2.4 zeigt ein minimales Beispiel für das Starten eines Nodes und enthält somit auch nur die Pflichtattribute des Nodes.

Listing 2.4: Node starten

```
1 <launch>
2   <node name='listener' pkg='test_ros' type='listener.py' />
3 </launch>
```

Beispiel 2

In Listing 2.5 werden ein Include und ein Remap gezeigt, die durch eine Group in einem speziellen Namespace gesetzt werden.

Listing 2.5: Namespaces Includes und Remaps

```
1 <launch>
2   <group ns='included'>
3     <include file='${find roslaunch}/example-include.launch' />
4     <remap from='chatter' to='hello' />
5   </group>
6 </launch>
```

Beispiel 3

Unterschiedliche Arten der Übergabe von Parametern sollen in Listing 2.6 beschrieben werden. Während in *Zeile 3* ein einfacher Parameter als ein String abgespeichert wird, wird in *Zeile 5* eine Datei von Parametern innerhalb eines Nodes geladen und ein anderer Namespace erzwungen. Der Parameter in *Zeile 8* wird automatisch und privat als Integer-Wert für den Node gespeichert.

Listing 2.6: Parameter

```
1 <launch>
2   <!-- force to string instead of integer -->
```

2. Grundlagen

```
3   <param name='somestring2' value='10' type='str' />
4   <node pkg='move_base' type='move_base' respawn='false' name='move_base' output='screen'>
5     <rosparam file="$(find wlan_mapper)/costmap_common_params.yaml" command='load' ns='
6       global_costmap' />
7   </node>
8   <node pkg='amcl' type='amcl' name='amcl' respawn='true'>
9     <param name='min_particles' value='50' />
10    </node>
11  </launch>
```

Beispiel 4

Das in Listing 2.7 angegebene lauffähige Beispiel startet zwei Simulationsumgebungen „turtlesim“ in unterschiedlichen Namespaces, wovon nur eine mit der Keyboard-Steuerung des „turtle_teleop_key“ Zeile 12-14 verbunden wird. Durch Topic-Remapping in einem Übergangsknoten „mimic“ Zeile 8-11 werden die Positionen und Steuerbefehle jedoch auf die zweite Umgebung übertragen und somit angeglichen.

Listing 2.7: Lauffähiges Group- und Remapping-Beispiel

```
1 <launch>
2   <group ns='turtlesim2'>
3     <node name='sim' pkg='turtlesim' type='turtlesim_node' />
4   </group>
5   <group ns='turtlesim1'>
6     <node name='sim' pkg='turtlesim' type='turtlesim_node' />
7   </group>
8   <node name='mimic' pkg='turtlesim' type='mimic'>
9     <remap from='input/pose' to='/turtlesim1/turtle1/pose' />
10    <remap from='output/command_velocity' to='/turtlesim2/turtle1/command_velocity' />
11  </node>
12  <node name='teleop' pkg='turtlesim' type='turtle_teleop_key'>
13    <remap from='turtle1/command_velocity' to='/turtlesim1/turtle1/command_velocity' />
14  </node>
15 </launch>
```

3 Kapitel 3

Verwandte Arbeiten

In diesem Kapitel werden einige Programmierumgebungen für diverse Roboterframeworks und Debugging-Tools für ROS vorgestellt, die auf die ein oder andere Weise im Kontext dieser Arbeit stehen. Hierfür wurden Beispiele für verschiedene grafische Programmiermethoden repräsentativ ausgesucht und ihre zugrunde liegenden Konzepte beschrieben. Anschließend erfolgt ein kurzes Fazit, welches die vorgestellten Programme und deren Konzepte in den Kontext der Arbeit einordnet.

3.1. Roboter-Entwicklungsumgebungen

Neben *ROS* existieren eine Vielzahl an Roboterkontrollarchitekturen, wie *Orocos*, *Orca* und *OPRoS*, sowie einige IDEs, um Komponenten für diese in High-Level-Sprachen textuell zu erzeugen, wie zum Beispiel *Eclipse*¹ oder *NetBeans*². Die Komponentenerstellung durch universell verwendbaren IDEs bietet vielen Programmieren die Möglichkeit in gewohnter Umgebung zu entwickeln, benötigt aber auch eine Konfiguration und Anpassung der IDEs für diesen Zweck. Eigenständige Programme können meist ohne aufwendige Konfigurationen verwendet werden und sind für Anfänger besser geeignet, da sie spezialisierter sind. Es existieren auch eine Reihe von Ikon-basierte Programmierkonzepte, wie *MORPHA*³ und Entwicklungsumgebungen, wie *LabVIEW* (NI, 2012). Diese Konzept werden schon seit einigen Jahren in wissenschaftlicher und technischer Software eingesetzt (Cote *et al.*, 2004). Häufig werden hierfür die Umgebungen so erweitert, dass sie auf die Robotik beziehungsweise die Komponentenerstellung in einer bestimmten Kontrollarchitektur spezialisiert sind. Beispiele hierfür sind die in Kim und Jeon (2008) verglichenen *Lego Mindstorms*-Programmierumgebungen und das *Microsoft Robotics Studio*, die eine graphische Erzeugung von Komponenten ermöglichen.

¹ <http://www.eclipse.org/> [Stand: 07.02.2012]

² <http://www.netbeans.org/> [Stand: 07.02.2012]

³ <http://www.morpha.de/> [Stand: 07.02.2012]

3. Verwandte Arbeiten

Programmierumgebungen, die wiederverwendbare Bausteine und ihre Verbindungen zueinander visualisieren, helfen bei der Strukturierung von Systemen und erleichtern das Verständnis für Mechanismen. Durch sie werden zusammengesetzte Programme strukturiert und sind somit leichter zu verstehen (Cote *et al.*, 2004). Jang *et al.* (2010) kritisieren, dass komponentenbasierten Programmiersysteme für die Roboterprogrammierung sich noch nicht allgemein durchgesetzt haben. Sie führen dies auf eine Fokusierung der Framework-Ersteller auf die Middlewares und eine damit verbundene Vernachlässigung der Unterstützungsmöglichkeiten für Entwickler bei der Komponentenentwicklung sowie deren Komposition, zurück.

Die nachfolgend betrachteten Entwicklungsumgebungen wurden für Roboterframeworks erstellt, die stark auf Wiederverwendbarkeit setzen und Repositories für Komponenten zur Verfügung stellen, da sie das Prinzip der komponentenbasierten Softwareentwicklung verfolgen. Manche der vorgestellten Programme basieren auf einer existierenden Entwicklungsumgebung, andere sind eigenständige Programme, die graphische Programmierung unterstützen. Der Nutzen von graphischen Entwicklungsumgebungen ist groß, da sie die Roboterprogrammierung vereinfachen und den Programmierprozess verbessern können. Die im Folgenden präsentierten Entwicklungsumgebungen sind, bis auf das Programm RobotFlow und das noch in der Entwicklung befindliche BRIDE, immer auf ein bestimmtes Roboterframework zugeschnitten, manche sogar auf einen speziellen Robotertyp. Die zu verwendende Programmiersprache ist bei diesen Programmen stets begrenzt. Manche der gezeigten Beispiele setzen auf die Komposition von Teilprogrammen, andere ermöglichen die Programmierung der Komponenten selbst. Durch einige wird bei der Komposition ein Datenflussmodell beschrieben, bei anderen findet das Automatenmodell Anwendung.

BRICS/BRIDE

Entwickler des *Best Practice in Robotics*-Forschungsprojekt (BRICS⁴, welches sich zum obersten Ziel setzt, den Roboter-Entwicklungsprozess zu Strukturieren und zu Formalisieren und dafür Programme, Berechnungsmodelle und Bibliotheken für eine kürzere Entwicklungszeit anbieten, arbeiten an einer auf Eclipse basierenden Entwicklungsumgebung BRIDE⁵. Diese soll nach dem modellgetriebenen Softwareentwicklungsansatz für verschiedene Roboterarchitekturen Softwaremodelle in C++ Programme übersetzen können.

⁴ <http://www.best-of-robotics.org/> [Stand: 07.02.2012]

⁵ <http://www.best-of-robotics.org/bride/> [Stand: 07.02.2012]

3.1. Roboter-Entwicklungsumgebungen

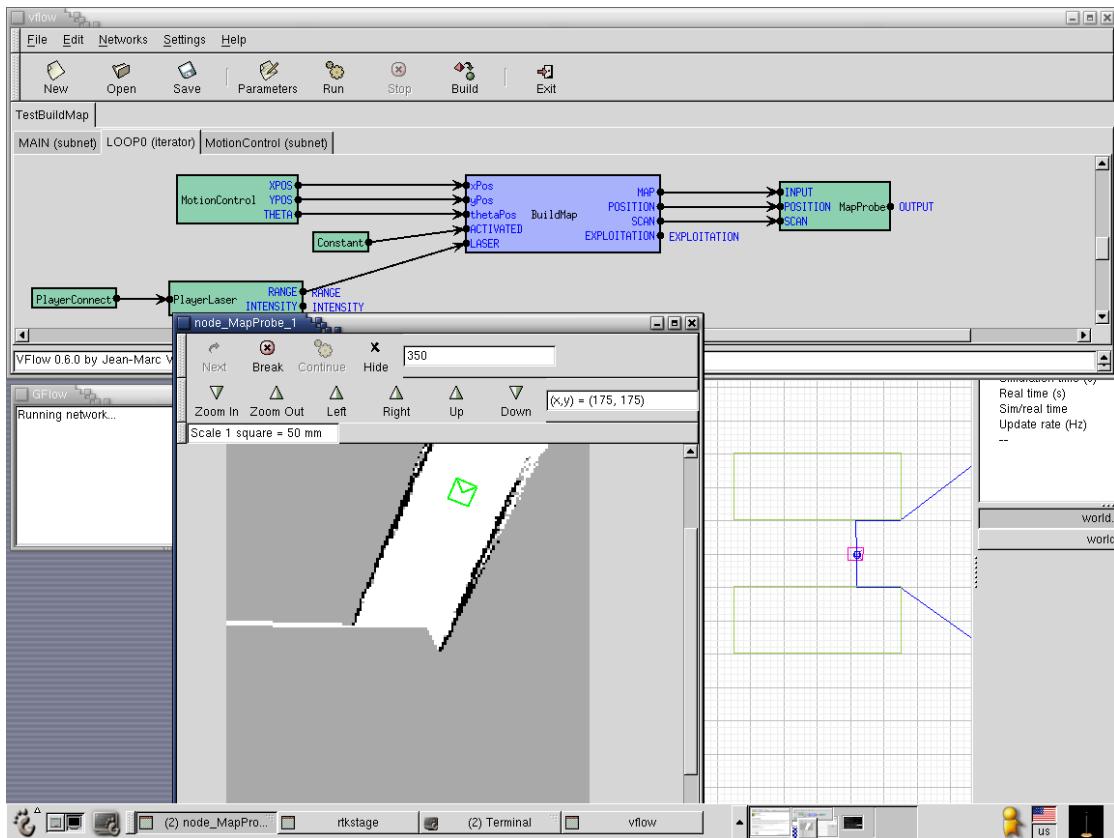


Abbildung 3.1.: RobotFlow - "22/04/2002 : Test network with mapping algorithm", Quelle: <http://robotflow.sourceforge.net/> [Stand: 07.02.2012]

FlowDesigner/RobotFlow

RobotFlow⁶ ist eine Erweiterung für die Entwicklungsumgebung FlowDesigner⁷, die nach dem Datenfluss-Modell arbeitet. Die Komponenten für FlowDesigner sind in C++ geschriebene Softwaremodule, die über die Entwicklungsumgebung verbunden werden können (siehe Abbildung 3.1) und synchronen Datenaustausch beherrschen. RobotFlow erweitert FlowDesigner um Eigenschaften, die für die Programmierung von mobilen Robotern benötigt werden. Darunter fallen sogenannte Processing-Blöcke, die Module, zum Beispiel für verschiedene Roboter, Computer Vision, Treiber oder ein vorgegebenes Verhalten, beinhalten. Diese Processing-Blöcke, die zum Beispiel Interface Blocks des Roboterframeworks MARIE oder

⁶ <http://robotflow.sourceforge.net/> [Stand: 07.02.2012]

⁷ <http://sourceforge.net/projects/flowdesigner/> [Stand: 07.02.2012]

3. Verwandte Arbeiten

Player/Stage-Simulatortreiber sein können, werden bei der Verwendung instantiiert. Zusammengesetzte Blöcke bilden Superblöcke, die ihrerseits wieder verwendet und verändert werden können. Ein daraus resultierendes Programm kann über die Umgebung gestartet oder die textuelle Repräsentation für eine spätere Ausführung über die Kommandozeile gespeichert werden (Cote *et al.*, 2004).

Lego Mindstorms/Entwicklungsumgebungen

Lego Mindstorms ist ein Roboterbaukasten bestehend aus einem programmierbaren Mikrocontroller sowie verschiedenen Elektromotoren und Sensoren. Mittlerweile existieren zwei Systeme unter dem Namen Lego Mindstorms, basierend auf unterschiedlichen Mikrocontrollern. Diese Systeme sind im allgemeinen nicht miteinander kompatibel und stellen unterschiedliche Produktgenerationen dar. Für den ersten Mikrokontroller, RCX (Robotic Command Explorer), existieren zwei Programmierumgebungen, RCX-Code und Robolab. Für das Nachfolgesystem, NXT, wurde eine neue Programmierumgebung namens NXT-G erstellt. Das Lego Robolab ist jedoch auch mit NXT kompatibel, so dass auch für dieses Nachfolgesystem eine simplere Programmierumgebung existiert.

RCX-Code

Abbildung 3.2 zeigt RCX-Code, welches eine Icon-basierte Programmierumgebung ist, die nach dem Flussdiagramm-Modell funktioniert. RCX-Code ist einfach gehalten, da es für die Benutzung durch Kinder konzipiert wurde. Einzelne Anweisungen der zugehörigen Scriptsprache können als Blöcke zur sequenziellen Abarbeitung wie Puzzle-Teile zusammengesteckt werden (Biggs und Macdonald, 2003). Diese Anweisungsblöcke umfassen hierbei Motorsteuerung, Sensorenbehandlung oder Variablenberechnungen. Ein Eventhandler der durch Sensoren getriggert wird ermöglicht sogar eine parallele Abarbeitung von Anweisungen.

LabVIEW-basiert

LabVIEW ist eine graphische Programmierumgebung von National Instruments (NI, 2012). Sie bildet die Basis für das Lego Robolab und die NXT-G Entwicklungsumgebung. LabVIEW ermöglicht die graphische Programmierung von textuellen

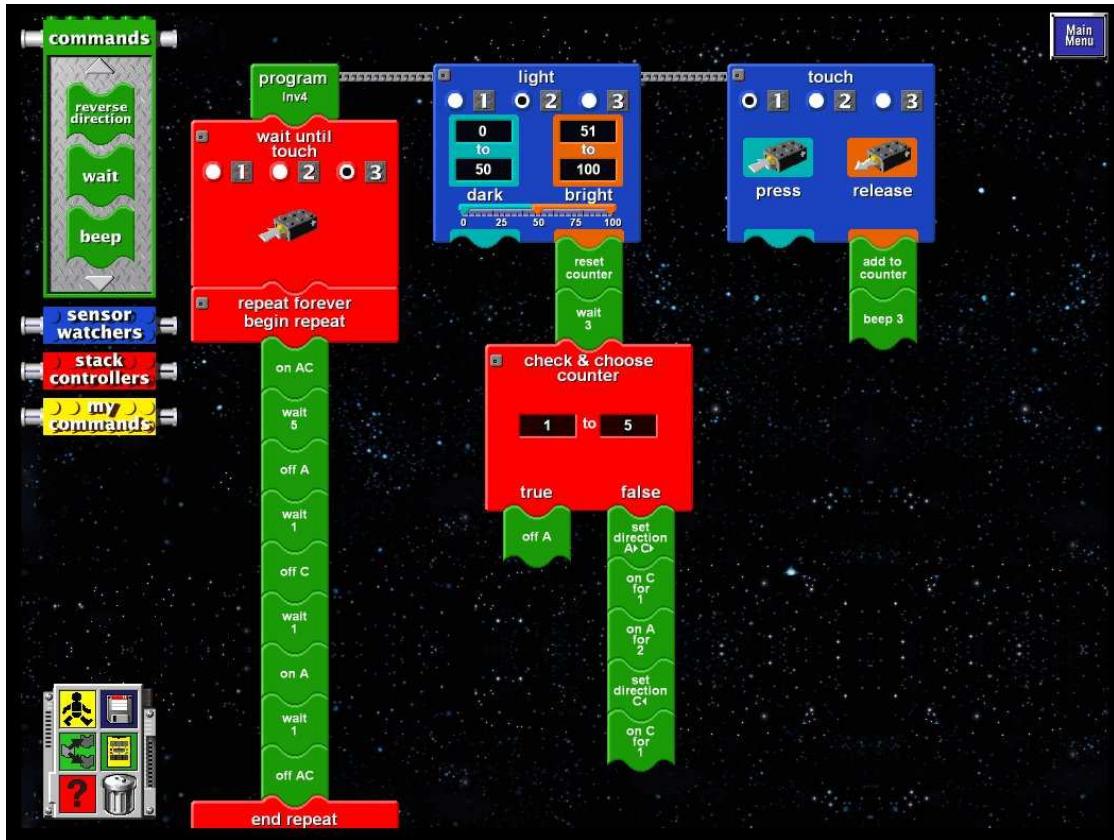


Abbildung 3.2.: RCX-Code, Quelle: Biggs und Macdonald (2003)

Komponenten, wie Schleifen, Bedingungsabfragen, etc. und deren Speicherung in der eigenen Sprache „G“. G-Programme, auch Virtuelle Instrumente (VIs) genannt, bestehen aus Nodes mit Inputs und Outputs. An diesen Inputs und Outputs können Verbindungen angelegt werden, um einen Datenfluss zu definieren. Nodes können hierbei zum Beispiel Datenquellen, Kontrollstrukturen, Benutzeroberflächen oder Funktionen sein.

Robolab

Robolab (siehe Abbildung 3.3) bietet drei Stufen der Entwicklung: 1. *Pilot*, 2. *Inventor* und 3. *Investigator*. Im Pilot können stark begrenzt Programmsequenzen, wie nach einer Vorlage konstruiert werden. Beim Inventor fallen alle Restriktionen weg und verschiedene Programmabläufe aus Virtuellen Instrumenten werden ermöglicht. Die Investigatorstufe erweitert das Programm um Logging-Funktionalität und Berechnungstools für bereits aufgezeichnete Daten.

3. Verwandte Arbeiten

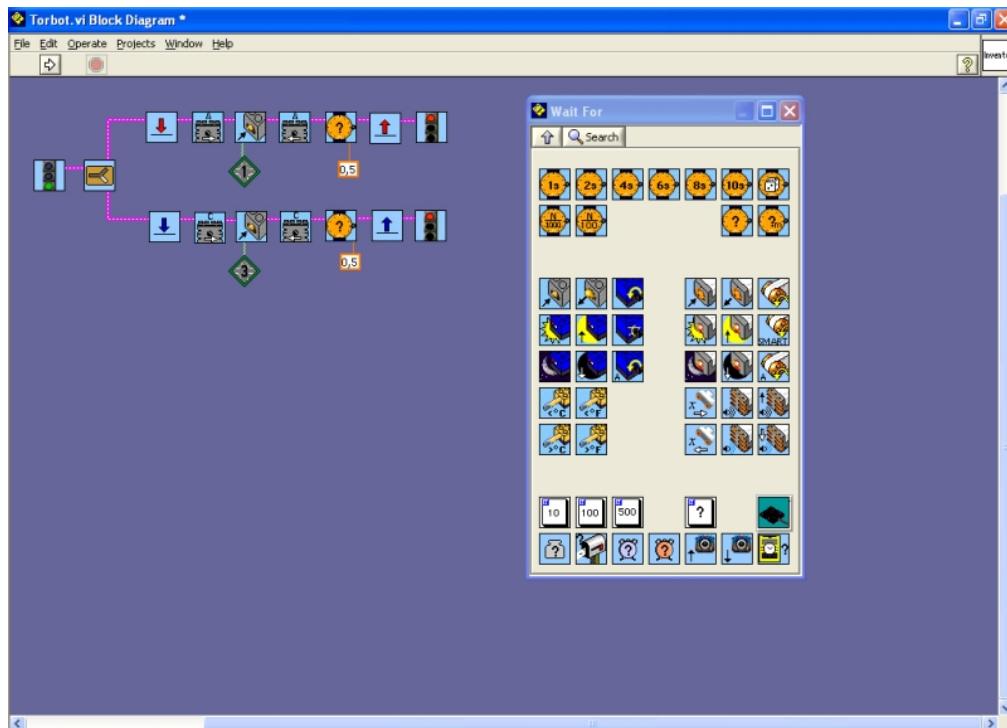


Abbildung 3.3.: Robolab, Quelle: <http://www.nxt-in-der-schule.de/lego-mindstorms-education-nxt-system/nxt-software/robolab-2.9> [Stand: 07.02.2012]

NXT-G

NXT-G (siehe Abbildung 3.4) ist eine eigene Sprache die an die Konzepte von Robolab angelehnt ist und somit auch Programme als Datenflussgraphen erstellt. NXT-G bietet hierfür eine andere Umgebung und auf das NXT-System spezialisierte Blöcke.

Nao Choregraphe

Der Nao Choregraphe ist ein auf den humanoiden Roboter Aldebaran Nao zugeschnittenes Programm, mit dem Flussdiagramme aus dem Kombinieren von Verhalten erstellt werden können (siehe Abbildung 3.5). Die Bausteine hierfür sind Blöcke mit vordefinierten Inputs und Outputs. Diese Blöcke dienen der Steuerung eines Roboters, der Verwendung von Sensoren oder stellen Programmschleifen dar. Eigene Blöcke können selbst geschrieben oder aber aus anderen Blöcken zusammengesetzt werden.

3.1. Roboter-Entwicklungsumgebungen

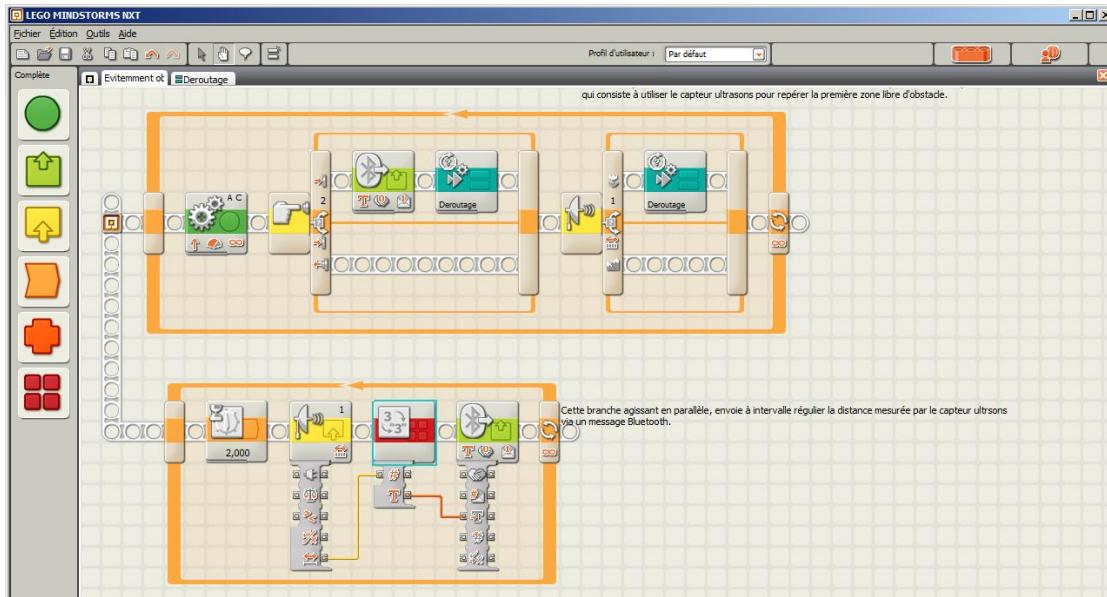


Abbildung 3.4.: NXT-G, Quelle: <http://www.generationrobots.de/roboterprogrammierung-mit-nxt-g-und-microsoft-vpl,fr,8,42.cfm> [Stand: 07.02.2012]

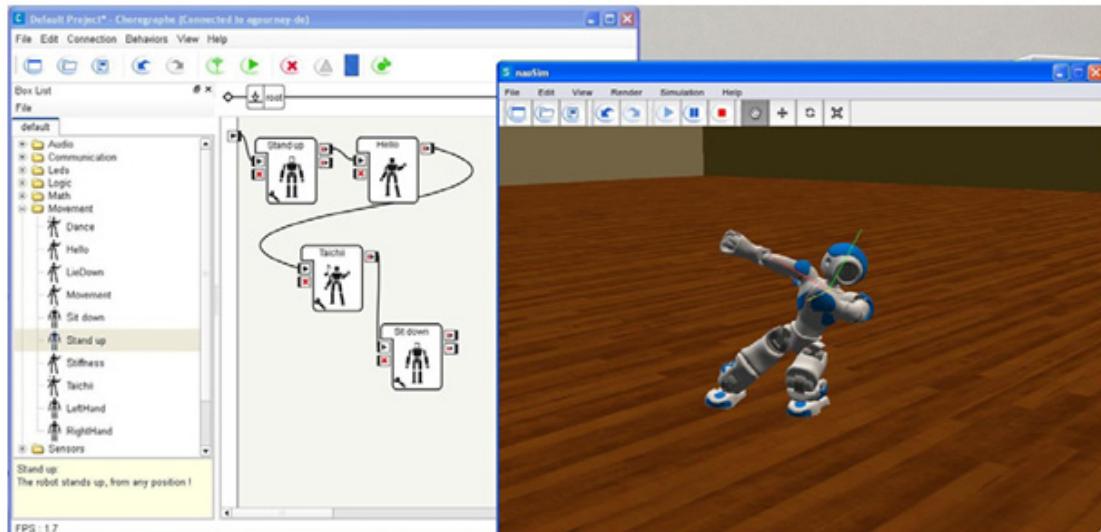


Abbildung 3.5.: Nao Choregraphe, Quelle: <http://www.aldebaran-robotics.com/en/Discover-NAO/Software/choregraphe.html> [Stand: 07.02.2012]

3. Verwandte Arbeiten

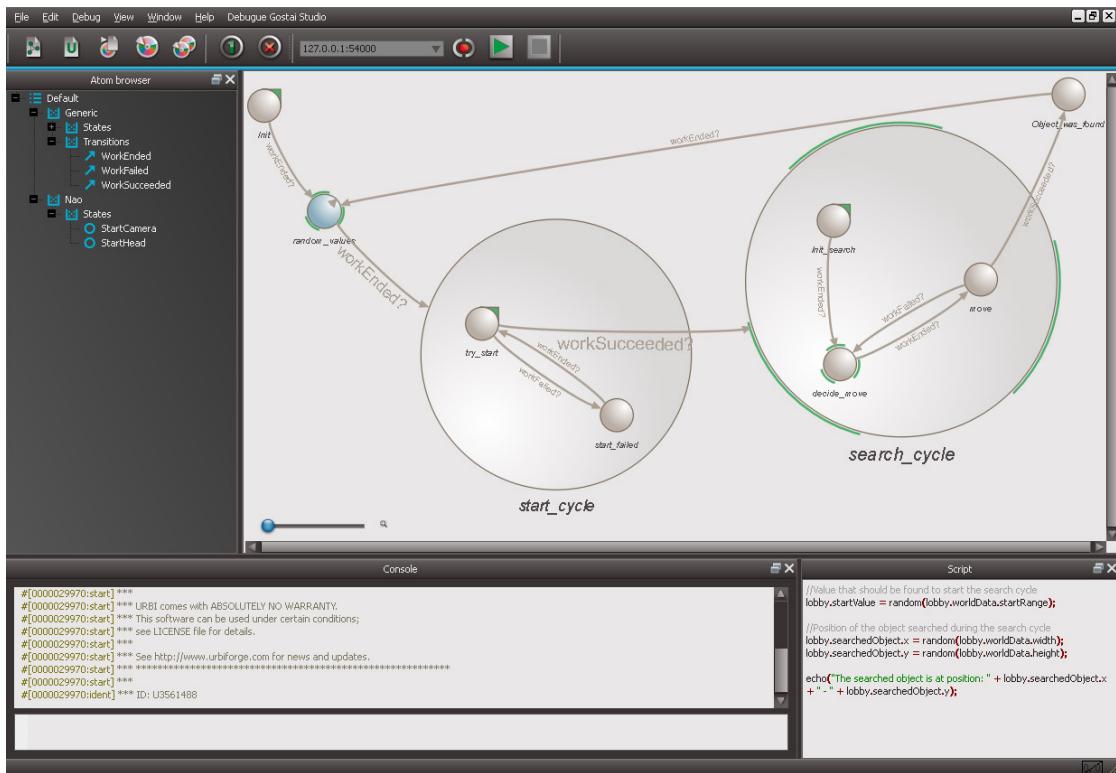


Abbildung 3.6.: Urbi - Gostai Studio, Quelle: http://www.gostai.com/products/jazz/gostai_studio/ [Stand: 07.02.2012]

Urbi Gostai Studio

Urbi⁸ ist eine Open-Source-Softwareplattform der Gostai Company, um Roboter oder allgemein komplexe Systeme auf einer Client/Server-Architektur zu steuern. Für eine Reihe festgelegter Robotermodelle der Ghostai Company existieren Komponenten, die über Urbi-Script kombiniert werden können, um ein Roboterverhalten zu erzeugen. Das Gostai Studio (Gostai, 2012) stellt eine Sammlung von Tools für diese, als Glue-Code fungierende, Scriptssprache Urbi-Script bereit. So werden Scripts auf einer Urbi Engine ausgeführt und mit dem visuellen State-Machine-Editor Urbi Live können diese Scripts erschaffen, gestartet und gestoppt werden. Bei Urbi Live entspricht jeder Zustand der State-Machine einem Urbi-Script und die Übergänge werden durch Events getriggert (siehe Abbildung 3.6). Das Urbi Lab ermöglicht schließlich noch die Entwicklung von graphischen Benutzeroberflächen und graphischen Fernsteuerungen für einen Roboter.

⁸ <http://www.urbiforge.org/> [Stand: 07.02.2012]

3.1. Roboter-Entwicklungsumgebungen

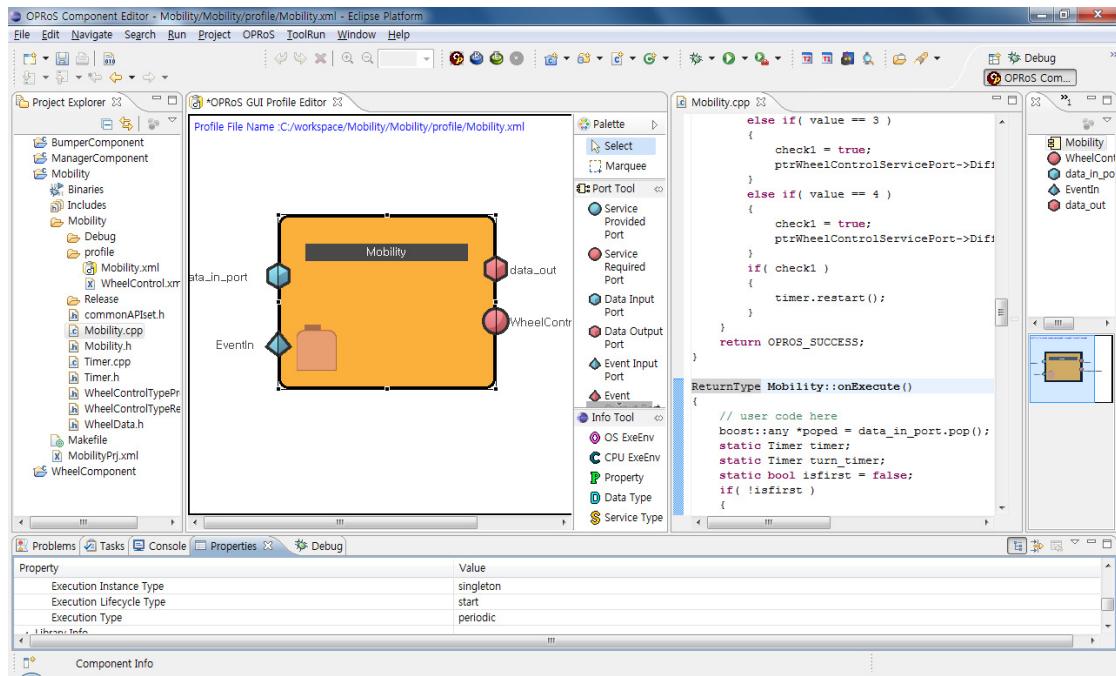


Abbildung 3.7.: OPRoS - Component Authoring Tool, Quelle: Jang *et al.* (2010, S.651)

OPRoS / IDE

Die *Open software Platform for Robotic Services* (OPRoS) ist eine Roboterkontrollplattform für Linux und Windows, die als Kommunikationsmiddleware CORBA verwendet und eine vollständige integrierte Entwicklungsumgebung, die als Plugin für Eclipse verfügbar ist, bietet. Ein Teil dieser IDE ist das *Component Authoring Tool* (siehe Abbildung 3.7) mit dessen Hilfe neue Komponenten in C++ erstellt werden können. Zusammen mit einer XML-formatierten Datei, dem sogenannten *Component Profile*, welche Informationen über die Schnittstellen (Ports) und Eigenschaften einer Komponente gemäß einer IDL enthält, ergibt das C++ Programm dann ein Paket (Jang *et al.*, 2010). Abbildung 3.8 zeigt einen anderer Teil der IDE, den *Component Composer*, der eine Liste aller lokal verfügbaren Komponenten enthält. Der Component Composer ermöglicht es, existierende Komponenten graphisch per Drag and Drop zu kombinieren, indem sie über die definierten Ports verbunden werden (Jang *et al.*, 2010, Park und Han, 2009). All diese Informationen über die Verknüpfungen, die relevanten Eigenschaften und Parameter können in einer XML-Datei gespeichert und anschließend manuell ausgeführt werden (Han *et al.*, 2010).

3. Verwandte Arbeiten

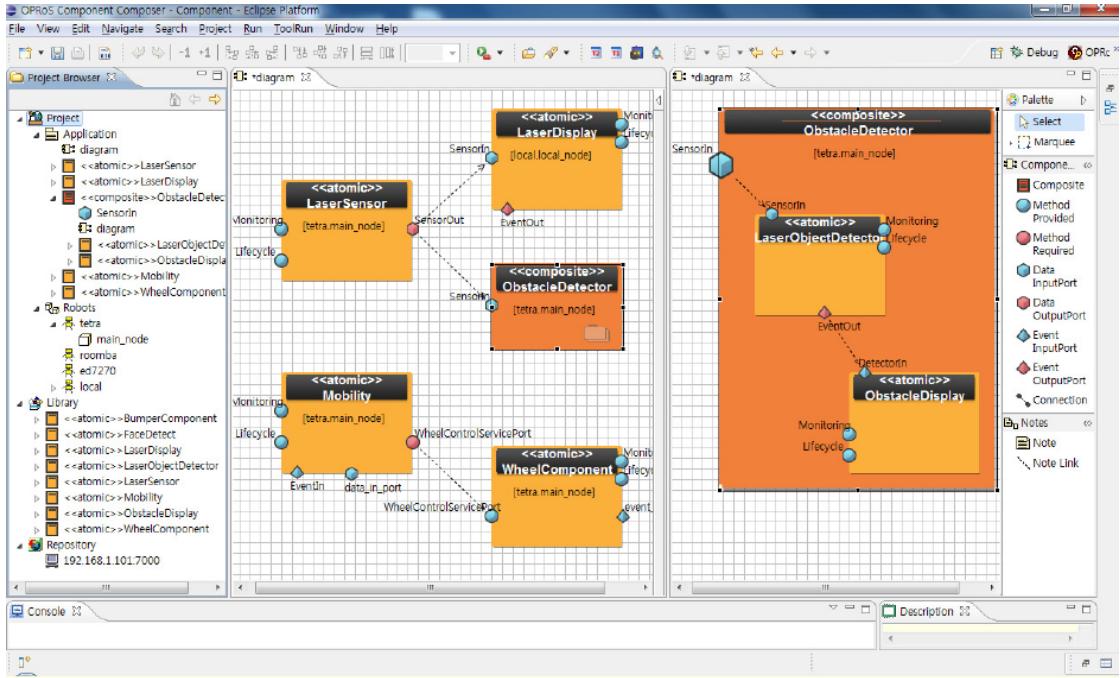


Abbildung 3.8.: OPROS - Component Composer, Quelle: Jang *et al.* (2010, S.651)

3.2. Debugging in ROS

Für ROS existieren einige graphische Programme, die das Debugging von Berechnungsgraphen ermöglichen und so dem Benutzer dienlich sind. Bezuglich der Visualisierung von Komponenten besitzt der *rxgraph* eine Sonderstellung. Mit Hilfe dieses Programms, wird der ROS-Berechnungsgraph dargestellt (siehe Abbildung 3.9). Der Datenfluss wird durch Nodes, den einzelnen Komponenten in ROS und den Verbindungen zwischen diesen, den Topics angezeigt. Dabei werden nur tatsächlich vorliegende Verbindungen und ROS Nodes verwendet, es werden also ausschließlich die Softwarekomponenten und ihre Verbindungen dargestellt, die aktuell auf dem System laufen. Dieses Tool dient einzig der Visualisierung des Berechnungsgraphen des aktuellen Systems und es ist nicht möglich auf andere Systeme zuzugreifen. Auch ist es nicht möglich Änderungen an dem vorliegenden Softwaresystem vorzunehmen oder dieses für eine erneute Ausführung abzuspeichern.

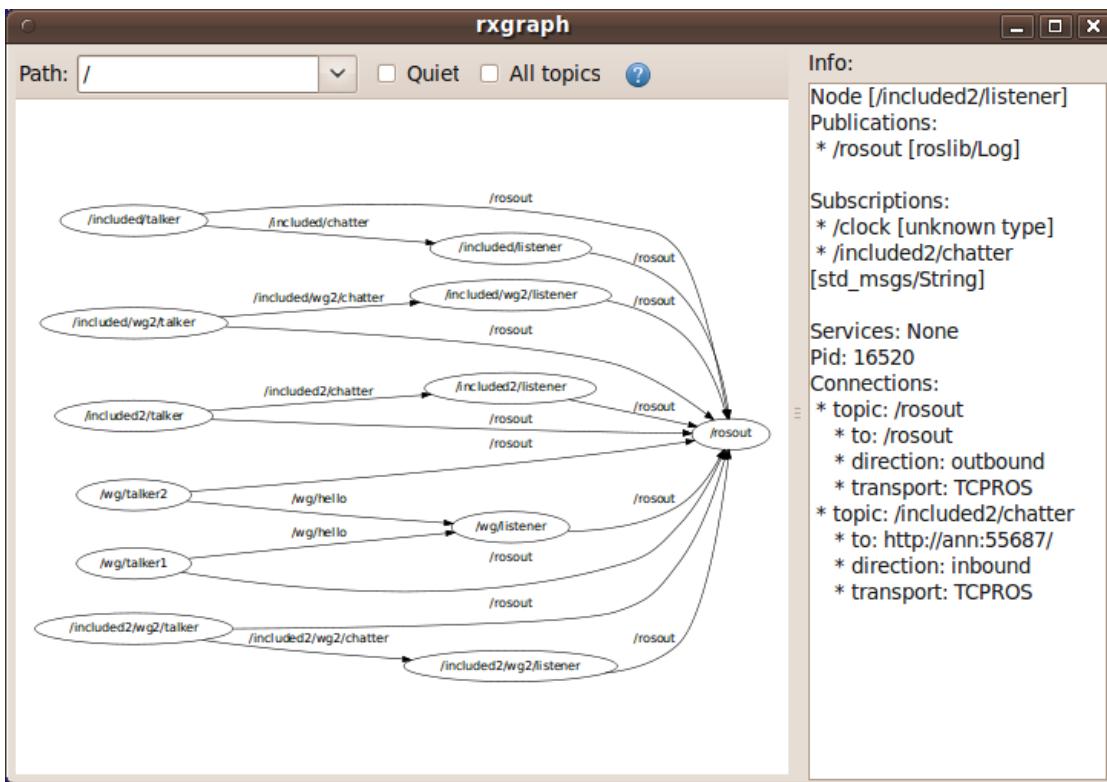


Abbildung 3.9.: rxgraph, Quelle: <http://www.ros.org/wiki/rxgraph> [Stand: 07.02.2012]

3.3. Fazit

Die gezeigten Programme und deren Konzepte sind für ihren jeweiligen Anwendungszweck gut geeignet. Einige ermöglichen eine Programmierung von Komponenten in einer bestimmten Sprache, andere bieten die Möglichkeit der grafischen Konfiguration von Robotern für die jeweilige Aufgabe. Wieder andere erlauben das Zusammensetzen von Komponenten, um größere wiederverwendbare Komponenten zu erstellen oder ein Programm zu bilden. Zu dieser Kategorie gehört unter anderem die *OPRoS IDE*, welche auf Interfacefiles zur Identifikation von Komponenten und deren Eigenschaften setzt. Manche der präsentierten Umgebungen, wie zum Beispiel der *FlowDesigner* erlauben eine Ausführung des erstellten grafischen Konstrukts aus dem Programm heraus und eine Abspeicherung in textueller Form für eine spätere Ausführung. Die Präsentation des Berechnungsgraphen eines Softwaresystems, wie es im *rxgraph* erfolgt, zeigt anschaulich über welche Interfaces Komponenten zur Laufzeit verbunden sind.

3. Verwandte Arbeiten

Einige dieser Ideen sollen im folgenden Kapitel (siehe Kapitel 4) für die Umsetzung des zu erstellenden Programms genutzt werden. Dafür müssen die Ideen für das ROS-Framework angepasst werden, wodurch eine Benutzerumgebung entsteht, die Komponenten aus diversen Programmiersprachen verwenden kann.

4 Kapitel 4

4 Implementierung des Programms

Dieser Abschnitt beschäftigt sich mit der Planung und der Umsetzung des Programms, welches, in Anlehnung an graphische ROS-Tools und an seine Funktion, *rxDeveloper* genannt wurde und im weiteren Verlauf auch so bezeichnet wird. Die Planung und Umsetzung umfasst nicht nur die Festlegung der Funktionen, sondern auch die verwendeten Bibliotheken und relevante Techniken. Außerdem beinhaltet sie die Konzeption der, für die Benutzerunterstützung notwendigen, Spezifikationsdateien. Zunächst wurde die Anforderungsanalyse durch eine Anfangsbefragungen (siehe Kapitel 4.1) der ROS-Entwicklern der Firma *Willow Garage* sowie einer Umfrage bei Softwareentwicklern und -benutzern aus der Community erhoben. Im Anschluß daran wurden die daraus gewonnenen Erkenntnisse genutzt, um die Anforderungsanalyse abzuschließen und den *rxDeveloper* und die zugehörigen Spezifikationsdateien zu erstellen. In unregelmäßigen Abständen von circa zwei Wochen wurden über den Zeitraum von drei Monaten die verschiedenen Entwicklungsstufen sowohl Community- als auch ROS-Entwicklern präsentiert und die Rückmeldungen genutzt, um das Produkt weiter zu verbessern. Das resultierende finale Gesamtpaket aus Spezifikationsdateien und *rxDeveloper* wird in den Kapiteln 4.2 und 4.3 vorgestellt und detailliert erläutert.

4.1. Anforderungsanalyse

Die bislang erfolgten Ausführungen stellen nur einen Rahmen dar. Die konkreten Eigenschaften und Fähigkeiten der geplanten IDE wurden in vorausgehenden Befragungen der Entwickler- und Benutzergemeinschaft evaluiert. Hierfür wurde zunächst bei *Willow Garage*, den Entwicklern von *ROS*, versucht eine Einschätzung der Wichtigkeit von verschiedenen Features zu erhalten, um anschließend einen Fragenkatalog zu generieren. Dieser wurde über Mailinglisten an die Benutzer und Entwickler in der Community herangetragen und die Ergebnisse ausgewertet.

4.1.1. Anfangsbefragung der ROS-Entwickler

In der Anfangsbefragung wurde den ROS-Entwicklern der Firma Willow Garage der Stand der Planung erläutert. So wurde die Grundidee des grafischen Launchfile-Editors und die visuelle Komposition von ROS-Nodes mittels Remappings vorgestellt und bereits erste Konzepte zu den Interfacedateien präsentiert. Anschließend wurde Auskunft über die Relevanz des Projekts, wichtige und sinnvolle Features und die zu verwendenden Bibliotheken eingeholt. Eine Kernfrage für die Implementierung betraf die grafische Umsetzung des *rxDevelopers*. Um ein möglichst abhängigkeitsarmes Programm zu schaffen, welches sich gut in die ROS Strukturen integriert, wurde hier das *Qt Development Framework* (Nokia (2012)) in Verbindung mit *C++* und der *C++-API* von ROS bestimmt. Qt wurde sowohl aus Gründen der Plattformunabhängigkeit als auch der zukünftigen Ausrichtungen bei Willow Garage auf dieses Development Framework, ausgewählt. Existierende ROS-Tools, die häufig zu diesem Zweck noch *wxWidgets* nutzen, sollen nach und nach auf Qt portiert werden. Auch die Verwendung des, für Computer und Menschen gleichermaßen lesbaren, *YAML*-Formats (Ben-Kiki *et al.*, 2009) für die Beschreibungen von Nodes mittels Spezifikationsdateien wurde hier beschlossen und ihre Nützlichkeit für das geplante Programm sowie für andere Projekte und Ideen, erkannt. So wurden solche Interfacedateien bereits bei Willow Garage angedacht, aber noch nicht umgesetzt. Die weiteren Ergebnisse der Befragung wurden in der Community-Umfrage verarbeitet und zur Diskussion gestellt.

4.1.2. Vorausgehende Befragung der ROS-Community

Die vorausgehende Befragung in der ROS-Community baute bereits auf der Anfangsbefragung der Entwickler bei *Willow Garage* auf und war an Community-Mitglieder gerichtet. Zu der Zielgruppe zählten sowohl Benutzer, die das Programm und seine Konzepte nach der Fertigstellung in ihren Arbeitsablauf integrieren sollten, als auch Entwickler, die an einer zukünftigen Weiterentwicklung mitwirken könnten.

Auf Grundlage der Rahmenbedingungen, sollten hier noch Design- und Implementierungsfragen geklärt werden. Eine Prämisse war es verschiedene Teilprobleme, wie das Lesen und Schreiben von Launchfiles, zu lösen aber externe Abhängigkeiten gering zu halten. Die Befragung selbst und ihre genaue Auswertung kann im Appendix (A.1) eingesehen werden. Dort befindet sich auch eine Auflistung der Kommentare der Umfrageteilnehmer, die ein zusätzliches Feedback darstellen.

Listing 4.1: Nodespezifikationsdatei - Konzept zum Zeitpunkt der Umfrage

```

1 type: node _type
2 subscriptions:
3   node/topic_name{topic_type}
4 publications:
5   node/topic_name1{topic_type}
6   node/topic_name2{topic_type}
7   node/topic_name3{topic_type}
8 services:
9 ...
10 parameters:
11 ...

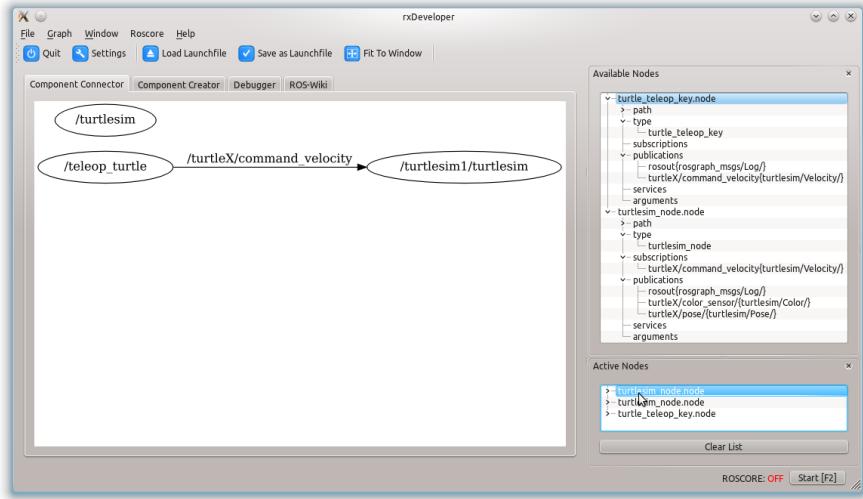
```

Zunächst wurden Details zu den geplanten Nodespezifikationsdateien evaluiert. Listing 4.1 zeigt das zu diesem Zeitpunkt erarbeitete Konzept. Bei dieser ersten Fragengruppe stellte sich heraus, dass das Konzept der Spezifikationsdateien angenommen wird und über zweidrittel der Befragten gewillt sind, solche Dateien zu erzeugen. Etwa gleich viele sprachen sich dafür aus, die Dateien auf einem übersichtlichen Niveau zu halten und nicht mit weiteren Informationen, wie etwa die im Vorfeld als lohnenswert erachteten Dependencies, zu belasten. Dependencies hätten Auskunft über Nodes geben können, die immer mit einem Node zusammen gestartet werden müssen.

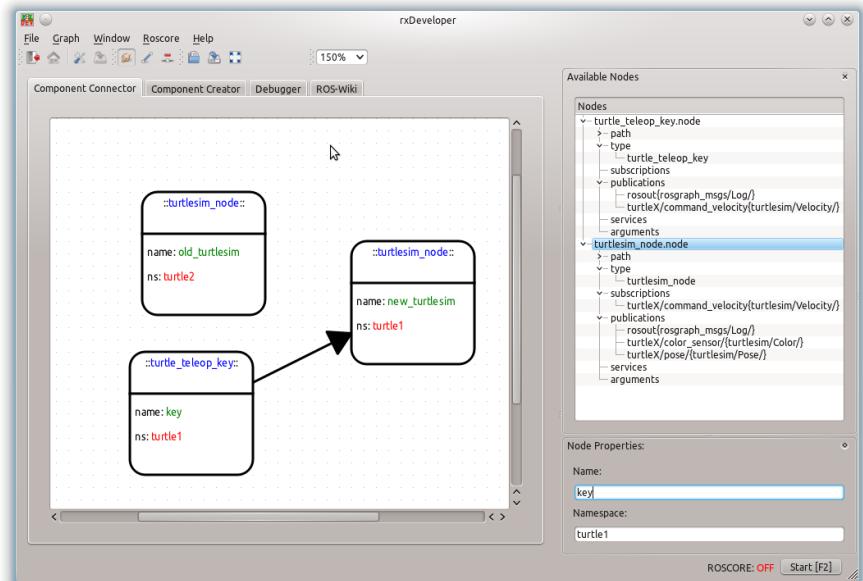
Die nächste Fragengruppe zielte auf die Wichtigkeit bestimmter Features ab. Als sehr wichtig wurde hier das Verändern von Nodeparametern eingestuft. Für sehr unwichtig wurden Möglichkeiten zum Editieren von Quelltext innerhalb des Programms erachtet und auch nur ein Viertel der Befragten konnten sich das Starten eines externen Editors aus dem Programm heraus vorstellen. Eine große Mehrheit sah auch keinen Nutzen im graphischen Anlegen von neuen *ROS*-Packages über das Programm. Im Gegensatz dazu fanden graphische Editier- und Erstellungsmöglichkeiten für Spezifikationsdateien großen Zuspruch. Fast als genauso lohnenswert erschienen den Befragten Möglichkeiten des Informationsimports für Spezifikationsdateien, etwa aus vorhandenen Launchfiles.

Der letzte Umfrageteil nahm Bezug auf die Verwendung geeigneter Hilfsbibliotheken für die Lösung einzelner Aufgaben im Programm. Hierbei wurden Bibliotheken zur Auswahl gestellt, die ohnehin schon durch *ROS* oder durch *Qt* vorhanden sind. Solche Bibliotheken lassen sich reibungslos und betriebssystemunabhängig integrieren, ohne das Projekt mit externen Abhängigkeiten zu belasten. Die Community bevorzugte meist die bereits in *ROS* verwendeten Bibliotheken, so bekam *TINYXML* (Thomason, 2012) als XML-Parser und -Emitter für Launchfiles den klaren Vorzug gegenüber den *Qt*-Implementierungen von *SAX* und *DOM*. Ebenso wurde *YAML-CPP* (Beder, 2012) für die Bearbeitung und Aufbereitung von Spezifikationsdateien bestätigt. Für die graphischen Darstellung der Launchfiles wird

4. Implementierung des Programms



(a) Graphviz-Prototyp: Rendering einer internen Repräsentation der Launchfiles in der DOT-Language



(b) QGraphicsScene-Prototyp: Darstellung einiger Launchfile-Elemente als QGraphicsItems

Abbildung 4.1.: Prototypen der graphischen Darstellung im Programm zum Zeitpunkt der Umfrage

die Verwendung der *QGraphicsScene*¹ mit *QGraphicsItems* bevorzugt. Diese Kombination erhielt eine vier fünfel Mehrheit gegenüber verschiedenen Ansätzen mit der *Graphviz DOT-Language*², welche immerhin im bekannten *ROS*-Tool *rxgraph* (siehe Abbildung 3.9) verwendet wird. Abbildung 4.1 zeigt die, für die Befragung erstellten, graphischen Darstellungen früher Prototypen des *rxDevelopers*, die auch eine Entwicklungsstufe des späteren Programms darstellen.

Weitere Ideen und Anmerkungen konnten während der Umfrage als Kommentare angefügt werden. Hierbei wurde einerseits eine nur optionale Verwendung der Spezifikationsdateien vorgeschlagen, andererseits wurde der Wunsch nach beliebigen Erweiterungen der Möglichkeiten innerhalb der Spezifikationsdateien geäußert. Dies hätte den Vorteil, dass der Benutzer diese Dateien auch als Grundlage für eigene Programme nutzen kann. In diesem Fall sollte das Programm zumindest keine Parsingfehler generieren. Die Unterstützung für Dynamic Reconfigure Nodes, war zwar bereits in Planung, aber während der Umfrage noch nicht ausgearbeitet. Dem entsprechend war es positiv zu werten, dass dieser Punkt seitens der Community nachgefragt wurde. Gleiches gilt für die Behandlung von Nodelets, welche sich auch schon in Planung befand, aber in den, mit Absicht einfach gehaltenen Beispielen der Umfrage, nicht aufgeführt wurde. Auf Seiten des Programms wurden Wünsche nach Editiermöglichkeiten für alle Launchfile-Elemente und eine korrekte Behandlung von Include-Tags geäußert. Außerdem wurde sich mehrfach gegen die Verwendung von Graphviz und für eine LabVIEW-ähnliche Gestaltung ausgesprochen, so dass Inputs und Outputs von Nodes leicht zu erkennen sind. Auch bereits im Vorhinein, in den Gesprächen mit Entwicklern von *Willow Garage* und in Vorüberlegungen, erarbeitete Zukunftsideen zu einer Weiteren Verwendung der Spezifikationsdateien wurden von der Community erneuert. So kamen auch hier Vorschläge zur Entwicklung eines Parsers, der automatisch Wiki-Einträge vornimmt, sowie die Idee zur automatischen Nodequelltext-Generierung aus den Spezifikationsdateien als zukünftige Entwicklungsoptionen.

Interpretation

Die Entwickler in der Community können sich den Nutzen des Programms gut vorstellen und erkennen gerade bei der angestrebten Kernfunktionalität, dem schreiben und weiterentwickeln von Launchfiles, die momentan unzureichend vorhandenen Unterstützungsmöglichkeiten für Entwickler. Als Teil der ROS-Gemeinschaft

¹ <http://developer.qt.nokia.com/doc/qt-4.8/qgraphicsscene.html> [Stand: 07.02.2012]

² <http://www.graphviz.org/doc/info/lang.html> [Stand: 07.02.2012]

und Kenner der *ROS*-Philosophie bevorzugen sie leichtgewichtige Programme, die sich mit einem Problem beschäftigen und dieses sehr gut lösen. Ein Programm für diese Zielgruppe muss so gestaltet sein, dass es optional verwendbar ist, es also keine Abhängigkeiten schafft. Jeder Entwickler arbeitet bereits in einer gewohnten Arbeitsumgebung und hat bestimmte Arbeitsabläufe. Aus diesem Grund verwundert es auch nicht, dass sich die Befragten gegen die Erschaffung einer größeren Entwicklungsumgebung aussprechen und keine Komplettlösung zum Erstellen von Nodes oder ähnlichem wünschen.

4.1.3. Anforderungen an das Programm

Die ursprünglichen Anforderungen an das geplante Programm, wie sie in der Aufgabenstellung dieser Arbeit (siehe Kapitel 1.2) beschrieben wurden, konnten durch die durchgeführten Befragungen ergänzt und genauer definiert werden. Über Rückmeldungen zu Programmpräsentationen gab es weitere Ideen, die mit in die Anforderungsanalyse einflossen. Das Ergebnis ist eine Anforderungskatalog in Bezug auf die Funktionen, Portabilität und den ROS-Rahmen.

Um als ein ROS-Tool akzeptiert zu werden, muss der *rxDeveloper* als ein eigenständiges und nicht funktional überlastetes Programm erstellt werden. Eine Gestaltung als Plug-In für Eclipse, wie beispielsweise die OPRoS IDE (siehe Kapitel 3.1) bietet zwar den Vorteil einer umfassenden ROS-IDE mit bereits vorhandenem Quelltexteditor für neue Nodes, widerspricht jedoch der Philosophie von ROS und ist auch nicht von der Entwicklergemeinschaft erwünscht. Des Weiteren darf die Benutzung des *rxDevelopers* nicht verpflichtend sein. Das bedeutet, dass es keinen Unterschied machen darf, ob man einen Launchfile, der im *rxDeveloper* entstanden ist, wieder im *rxDeveloper* oder auf herkömmliche Weise weiterverwendet. Da auch ROS auf verschiedenen Betriebssystemen lauffähig ist, ist es notwendig, dass das Programm betriebssystemunabhängig gestaltet wird. Weitere Anforderungen sind eine offene und modulare Gestaltung der Benutzerumgebung, die leicht erweiterbar und anpassbar ist. Auch das Spezifikationsdateiformat muss offen gestaltet sein, so dass niemandem die Verwendung verwehrt bleibt. Sollte sich das Spezifikationsdateiformat oder die Verwendete Technik dahinter ändern, so stellt die Modularität des Programms sicher, dass kein zusätzlicher Aufwand außer der Neuimplementierung der betreffenden Module entsteht. Die Spezifikationsdateien sind ein wichtiger Teil der Arbeit und ermöglichen die eigentliche Erleichterung bei der Entwicklung. Dennoch sollte die Benutzung solcher Dateien nicht verpflichtend und das Programm auch ohne die in den Dateien vorhandenen Informationen von Nutzen sein.

Funktionalität

Die Kernaufgabe des Programms ist das Öffnen oder Neuerstellen sowie das Abspeichern von korrekten Launchfiles. Dazu gehört vor allem die grafische Darstellung und das Ermöglichen von Änderungen durch den Benutzer sowie Funktionen zum Testen der Dateien. Folgende Punkte müssen hierbei abgedeckt werden:

(i) Unterstützung beim Design von Launchfiles

Öffnen und Editieren von existierenden Launchfiles

Bereits existierende Launchfiles sollen im Programm syntaktisch und semantisch korrekt geöffnet und fehlerfrei weiterverarbeitet werden können.

Abdeckung aller möglichen Launchfile-Elemente

Das Programm muss alle Elemente eines Launchfiles (siehe Kapitel 2.4.1) darstellen und erzeugen können. Dabei müssen alle Eigenschaften die ein Element haben kann ermöglicht und durch geeignete Zwangsbedingungen für ausschließlich gültige Eigenschaftenkombinationen gesorgt werden. Hierdurch soll sichergestellt werden, dass auch neu erzeugte Dateien syntaktisch und semantisch korrekte sind.

Vollständigkeit in Bezug auf Details und Positionen der Launchfile-Elemente

Neben dem unterschiedlichen Informationsgehalt zwischen zwei verschiedenen Elementen des gleichen Typs (siehe Kapitel 2.4.1), können diese auch syntaktisch (siehe Kapitel 2.4.2) verschiedene Positionen einnehmen. Um die Vollständigkeit des Programms sicherzustellen, müssen alle möglichen Kombinationen beachtet werden.

Freie Positionen der Launchfile-Elemente in der Darstellung

Die Elemente sollen zur persönlichen Strukturierung der Launchfiles frei positionierbar sein und beim erneuten Öffnen eines Launchfiles die vom Benutzer bestimmte Struktur wiederherstellen. Auf diese Weise kommen die ausschlaggebenden Vorteile der graphischen Programmierung zum Tragen.

Anzeigen einer Liste von existierenden Nodes/Nodelets

Zur Übersicht sollte eine Liste mit existierenden Komponenten dargestellt werden. Diese soll Informationen bieten, die den Benutzer über die Schnittstellen der Komponenten aufklären und so die Komponenten-Verwendung vereinfachen. Die Liste soll durch Spezifikationsdateien gefüllt werden, die für das Programm zugänglich sein müssen.

4. Implementierung des Programms

Komposition der Elemente und Konzeption des ROS Berechnungsgraphen

Eine Hauptfunktion des Programms soll die Vereinfachung bei der Erstellung und Verbesserung bei der Darstellung von geplantem Datenfluss bei der Interprozesskommunikation sein. Vor allem die Neukonnektierung von Komponenten (Nodes/Nodelets) über Remaps soll graphisch so gestaltet werden, dass die interagierenden Komponenten verbunden dargestellt werden. Verbindlungsmöglichkeiten zwischen Komponenten sollen, sofern die Informationen durch Spezifikationsdateien vorhanden sind, angezeigt und so die Zeit für die Entwicklung reduziert werden.

Parametrisierung von Nodes

Spezifikationsdateien sollen auch dafür genutzt werden, um mögliche Parameter einer Berechnungskomponente zur Verfügung zu stellen, damit der Benutzer einzelne Parameter bei Bedarf komponentenkonform anpassen kann.

Launchfilegenerierung aus dem dargestellten Graphen

Nach Beendigung des Editierungs- bzw. Neuerstellungsprozesses sollen aus den graphischen Darstellungen im Programm Launchfiles generiert werden können.

(ii) Erstellung und Editieren von Daten

Erstellung von ROS-Paketen und Dateien

Pakete stellen in ROS ein wichtiges Konzept dar und enthalten neben ausführbaren Programmen viele wichtige, oft textbasierte Daten (siehe Tabelle 2.1). Auch Launch- und Parameterfiles werden in Paketen organisiert. Die Erstellung von Paketen und das Erstellen von Daten in der Paketstruktur erfolgt bislang über die Kommandozeile. Der *rxDeveloper* sollte hier Möglichkeiten bieten Pakete grafisch zu erstellen und zu verwalten. Der Ablehnung für dieses Feature durch die Community steht der Nutzen für das Programm gegenüber. Wenn Launchfiles gespeichert werden, sollen diese unter Umständen in ein bestimmtes Paket gespeichert werden. Von daher ist eine einfache Verwaltung für Pakete sinnvoll.

Erstellung und Veränderung von Komponentenspezifikationen

Der *rxDeveloper* kann seine Stärken nur entfalten, wenn ihm über die Spezifikationsdateien Informationen zu den Komponenten zur Verfügung stehen. Vor allem für selbst erstellte Komponenten sollte ein Spezifikationsdatei-Generator und die Möglichkeit der einfachen Modifikation von existierenden Spezifikationsdateien zur Verfügung stehen.

(iii) *Unterstützung beim Testing und Debugging*

Testing

Für den Fall, dass der erstellte Launchfile auf dem gleichen System ausgeführt werden soll, auf dem der *rxDeveloper* läuft, soll eine lokale Ausführung des Launchfiles aus dem *rxDeveloper* ermöglicht werden. Das Starten und Stoppen dient nicht nur als Vereinfachung gegenüber der konventionellen Startmethode über die Kommandozeile, es kann auch verwendet werden, um Informationen über die Lauffähigkeit des Launchfiles zu gewinnen und Fehler aufzuspüren.

Debugging

Für einzelne Analyse- und Debuggingaufgaben existieren in ROS schon spezialisierte graphische Programme. Diese alle einzeln per Kommandozeile zu öffnen ist unübersichtlich und umständlich. Der *rxDeveloper* sollte diese Werkzeuge zur Verfügung stellen, so dass der für den Nutzer gewohnte Arbeitsablauf nicht gestört und trotzdem vereinfacht wird.

Öffnen vorhandener Launchfiles

Beim Öffnen existierender Dateien sollen mögliche Fehlerquellen angezeigt werden, so dass diese nicht mühsam manuell gesucht werden müssen.

4.2. Nodespezifikationsdateien

Die Nodespezifikationsdateien sollen verwendet werden, um dem Benutzer des Tools die Eigenschaften einer ROS-Komponente bekannt zu machen. Auf diese Weise können die Komponenten in die Benutzeroberfläche integriert werden und zeigen dort ihre Eigenschaften bereits zur Entwicklungszeit. Der große Vorteil ist, dass der Benutzer schon offline, also noch vor der Ausführung des Launchfiles und der Betrachtung des daraus entstehenden Berechnungsgraphen eine Vorstellung erhält, wie die Komponenten aussehen und arbeiten werden. Obgleich sie viele Vorteile mit sich bringen, ist die Verwendung der Spezifikationsdateien nicht zwingend notwendig für den *rxDeveloper*.

Die hier beschriebenen Nodespezifikationsdateien verwenden das YAML-Format und sind somit intuitiv lesbar und erstellbar. Die Syntax und Semantik des Formats sind zusammen mit der *rxDeveloper*-Benutzeroberfläche entstanden und gewachsen. Nodespezifikationsdateien gelten für Nodes, Nodelets und für Dynamic

Reconfigure Nodes gleichermaßen, somit müssen sie alle Eigenheiten der verschiedenen Nodetypen ermöglichen. Eine Spezifikationsdatei ist dabei stets für genau eine Komponente zuständig und deklariert für diese folgende Daten:

Type In ROS bezeichnet das den ursprünglichen Komponenten-Bezeichner, welcher nicht mit dem Namen, den eine Komponente bei der Instantiierung erhält, verwechseln werden darf.

Package Der Package-Name legt das Paket fest, in welchem sich die Komponente befindet. Gemeinsam mit dem Type wird so eine Komponente eindeutig festgelegt.

Interfaces Eine Liste von unterschiedlichen Schnittstellen der Komponente. Für ROS existieren Client/Server, Publish/Subscribe und Parameter als Schnittstellen einer Komponente (siehe Kapitel 2.3).

4.2.1. Syntax und Semantik der Spezifikationsdateien

Der allgemeine Aufbau einer Spezifikationsdatei wird durch das Schema in Listing 4.2 veranschaulicht. Spezifikationsdateien bestehen aus Pflichtelementen und optionalen Elementen. Erstere bilden den Kopf, letztere den Rumpf einer Spezifikationsdatei. Die Reihenfolge in der diese Elemente geschrieben werden ist prinzipiell beliebig. Es bietet sich jedoch aus Übersichtsgründen an die Pflichtelemente stets zu oberst aufzuführen. Die zwei Pflichtelemente sind, entsprechend der YAML-Terminologie (Ben-Kiki *et al.* (2009)), jeweils ein *Mapping* von einem *Skalar* zu einem anderen *Skalar*. Ein Mapping ist definiert als ein *Schlüssel/Wert-Paar* (Key/Value), ein *Skalar* ist eine Folge von Null oder mehr Unicode-Zeichen. Das erste Mapping bezieht sich auf den Node-Typ und wird mit „*type*“ eingeleitet. Das Zweite dient der Zuordnung des Nodes zu einem ROS-Paket und wird mit „*package*“ gesetzt. Die optionalen Elemente bestehen aus *Mappings von Sequenzen*. Diese Sequenzen verfügen über weitere *Mappings von Skalaren zu Skalaren*. Diese Einträge sind deshalb optional, da die repräsentierten Eigenschaften nicht in jedem Node vorhanden sind. Für die jeweiligen *Mappings* werden je nach Element unterschiedliche *Schlüsselbegriffe* (Keys) verwendet. Hier können folgende Schlüsselbegriffe verwendet werden:

- „*subscriptions*“, für Topics, die der Node empfängt
- „*publications*“, für Topics, auf die der Node Informationen sendet

- „services“, für die synchronen Kommunikationsmöglichkeiten des Nodes
- „parameters“, für die Einstellungsmöglichkeiten des Nodes

Eine Sequenz der Subscriptions oder Publications benötigt jeweils zwei Mappings, eingeleitet durch „topic“ und „type“, die den Namen eines Topics sowie dessen Typ, definieren. Für eine Sequenz der Services gilt das Gleiche, wie für die Subscriptions und Publications, jedoch ersetzt „name“ den „topic“-Key, da Services mit Namen aufgerufen werden und nicht über das asynchrone Topic System funktionieren. Parameter sind wiederum in Pflicht- und optionale Mappings unterteilt. Ein Parameter benötigt einen Namen, „name“ und einen Typ, „type“. Des Weiteren kann optional ein voreingestellter Wert angegeben werden, der mit „default“ eingeleitet wird. Für Dynamic Reconfigure Nodes, mit Parametern, die vom Typ her einem Zahlenwert entsprechen (int, double, float), kann auch noch optional ein Gültigkeitsbereich mit „range“ eingetragen werden.

Listing 4.2: Nodespezifikationsdatei - Schema

```

1  type: binary_node
2  package: node_package
3
4  subscriptions:
5  - topic: first/topic_name
6    type: topic_type
7  - topic: second/topic_name
8    type: topic_type
9  -
10 publications:
11 - topic: first/topic_name
12   type: topic_type
13 - topic: second/topic_name
14   type: topic_type
15 -
16 services:
17 - name: first/service
18   type: service_type
19 - name: second/service
20   type: service_type
21 -
22 parameters:
23 - name: firstParameter
24   type: string
25   default:
26 - name: ~anotherParameter
27   type: double
28   default: 0.0
29   range: [-1.0,1.0]
30 -

```

Eine Auflistung aller verwendbaren Keys bietet Tabelle 4.1. Sämtliche Values werden im *rxDeveloper* automatisch als Strings verarbeitet und sollten nicht explizit mit Anführungsstrichen versehen werden. Einzige Ausnahme hierfür bilden Parameter vom Typ *string*. Bei diesen sind Anführungszeichen für den Wert des Keys „default“ gefordert, da es sich hier explizit um Strings handelt. Die optionale Angabe eines Wertes für *range* bezieht sich primär auf *Dynamic Reconfigure Nodes*, bei

default	package	publications	services	topic
name	parameters	range	subscriptions	type

Tabelle 4.1.: Mögliche Schlüsselwörter einer Spezifikationsdatei

denen eine solche Angabe eine Rolle spielt. Ein Programm, wie der *rxDeveloper*, kann diese Information nutzen, um eine korrekte Parametrisierung sicherzustellen. Die Syntax für die Werte des „*range*“-Keys entspricht der üblichen Schreibweise für abgeschlossene Intervalle, also der Minimalwert und der Maximalwert stehen komma-getrennt in eckigen Klammern (siehe Listing 4.2, Zeile 29). Der Zahlentyp sollte dem angegebenen Parametertyp entsprechen.

Beispiel

Das Beispiel Listing 4.3 zeigt die unterschiedlichen Keys des Nodes *turtlesim_node* aus dem Paket *turtlesim*. Die Keys der Pflichtmappingfelder, „*type*“ (Zeile 1) und „*package*“ (Zeile 2) werden somit auf „*turtlesim_node*“ und „*turtlesim*“ gemappt. Da der Node nur ein Subscription-Topic besitzt, besteht die Sequenz, des optionalen Elements *subscriptions* (Zeile 3-5) auch nur aus einem Eintrag mit *topic*- und *type*-Mappings. Darauf folgen die Sequenzen für *publications* (ab Zeile 6) *services* (ab Zeile 13) und *parameters* (ab Zeile 28). Die Sequenz der Publications besitzt auch nur *topic*- und *type*-Mappings pro Sequenz-Eintrag (Zeilen 7-12) und bei Services das *name*-Mapping an Stelle des *topic*-Mappings tritt (Zeilen 14-27). Die Sequenz der Parameter beinhaltet jeweils *name*-, *type*- und *default*-Mappings (Zeilen 29-37). Ein Optionales *range*-Mapping ist in diesem Beispiel nicht vorhanden, da der Node zu diesem Beispiel keine dynamische Rekonfigurierung von Parametern unterstützt.

Listing 4.3: Nodespezifikationsdatei - Beispiel

```

1 type: turtlesim_node
2 package: turtlesim
3 subscriptions:
4   - topic: turtle1/command_velocity
5     type: turtlesim/Velocity/

```

```

6 publications:
7   - topic: rosout
8     type: rosgraph_msgs/Log/
9   - topic: turtle1/color_sensor
10    type: turtlesim/Color/
11   - topic: turtle1/pose
12     type: turtlesim/Pose/
13 services:
14   - name: /turtle1/teleport_absolute
15     type: turtlesim/TeleportAbsolute
16   - name: /reset
17     type: std_srvs/Empty
18   - name: /spawn
19     type: turtlesim/Spawn
20   - name: /clear
21     type: std_srvs/Empty
22   - name: /turtle1/set_pen
23     type: turtlesim/SetPen
24   - name: /turtle1/teleport_relative
25     type: turtlesim/TeleportRelative
26   - name: /kill
27     type: turtlesim/Kill
28 parameters:
29   - name: ~background_b
30     type: int
31     default: 255
32   - name: ~background_g
33     type: int
34     default: 86
35   - name: ~background_r
36     type: int
37     default: 69

```

4.2.2. Name und Pfad der Spezifikationsdateien

Die Spezifikationsdateien sollen vorzugsweise Teil der Pakete werden, in denen auch die zugehörigen Nodes liegen. Das erstellte Programm durchsucht in der aktuellen Version alle durch ROS-Tools erreichbaren Packages nach in einem Unterordner „/node“ befindlichen Spezifikationsdateien. Durch die Pflichtangabe des „*package*“ steht es dem Benutzer jedoch auch frei, Spezifikationsdateien in nicht zu der Datei gehörigen Paketen ohne Funktionseinbussen zu speichern. Auf diese Weise wird sichergestellt, dass auch Spezifikationsdateien zu Nodes, die in Paketen liegen, in denen der Benutzer keine Schreibrechte hat, erstellt und genutzt werden können. Das Ziel ist jedoch, dass Node-Ersteller die Spezifikationsdateien direkt im Paket mit dem erstellten Node mitliefern.

Für die Dateinamen existiert außer der wichtigen Endung *.node* keine besondere Benennungskonvention. Dennoch wird eine sinnvolle Benennung mit *nodetype.node* empfohlen, da auf diese Weise eine bessere Übersicht und eine leichtere Zuordnung auf Dateisystemebene gewährleistet wird. Ohne die Endung *.node* ist es dem Programm später nicht möglich die Dateien zu identifizieren, auch dann nicht, wenn sie im korrekten Ordner hinterlegt wurden. Listing 4.4 zeigt den gültigen Dateipfad für Spezifikationsdateien, wobei *<package path>* der Pfad zu einem beliebigen Paket

ist und `<nodetype>` für das in der Datei angegebene *type*-Mapping stehen sollte, aber nicht muss. Wie an anderer Stelle bereits erläutert ist jeder Spezifikationsfile für genau einen Node oder ein Nodelet zuständig.

Listing 4.4: Nodespezifikationsdatei - Dateipfadkonvention

```
<package path>/node/<nodetype>.node
```

4.2.3. Erstellung der Spezifikationsdateien

Eigenschaften die ein Node besitzt, sollten stets in den Dateien aufgeführt werden. Je mehr Informationen über einen Node enthalten sind, desto mehr kann das Programm den Benutzer bei der Roboter-Softwareentwicklung unterstützen. Ein Entwickler von Nodes sollte künftig die zugehörigen Spezifikationsdateien gleich mit erstellen und in den Paketen hinterlegen. Der Mehraufwand ist verglichen mit dem Nutzen minimal. Von der Verwendung der Nodespezifikationsdateien innerhalb des *rxDevelopers* abgesehen können diese auch schon bei der Konzeption von Nodes dienlich sein. Der Ersteller eines Nodes muss ohnehin planen, welche Eigenschaften ein Node haben soll. Eine Spezifikationsdatei kann somit als Strukturierungshilfe und Checkliste während der Node-Erstellung genutzt werden. Hinzukommt, dass die aufgeführten Informationen früher oder später zusammen getragen werden müssen, falls das erstellte Programm auf der ROS-Wikiseite veröffentlicht werden soll.

Der *rxDeveloper* besitzt einen *Specfile-Editor* der in Kapitel 4.3.8 beschrieben wird. Mit diesem können syntaktisch korrekte Spezifikationsdateien effizient und einfach erstellt und verändert werden. Die Verwendung von eigenen YAML-Zeilen ist nur über einen Texteditor möglich, jedoch nicht verboten. Das Format ist offen und kann von jedem Benutzer beliebig erweitert werden, solange nicht die in Tabelle 4.1 angegebenen Keys für eigene Definitionen verwendet werden. Dies ermöglicht, dass die Dateien in weitere Programme benutzt werden können, die auf die Spezifikationen von Nodes zugreifen sollen. Der *rxDeveloper* verwendet jedoch nur die in Kapitel 4.2.1 angegebenen Informationen aus den Dateien.

4.3. rxDeveloper

4.3.1. Implementierungsdetails

Bereits in einem frühen Stadium der Planung wurde *C++* als Programmiersprache, unter anderem aus Geschwindigkeitsbetrachtungen, festgelegt. Der *rxDeveloper* verarbeitet im Betrieb und besonders beim Starten viele Daten. Einer schnellen Compilersprache, die zudem noch hochgradig objekt-orientiert benutzt werden kann, wie *C++* ist in Hinblick auf die Kompatibilität zu den weiteren Bibliotheksentscheidungen, die geeignete Wahl für das Programm. Auch die Verwendung einer ROS API ist ein Kriterium. Der *rxDeveloper* ist in der Lage sich so in ROS zu integrieren, dass er mit dem *Master Service* (2.3.2) kommunizieren kann und die Ausführung von Launchfiles ermöglicht. Die Verwendung der *C++-API* für ROS ist folglich unumgänglich.

Mittels des *Qt Development Frameworks* (Nokia (2012)) wurde die graphische Oberfläche gestaltet. Diese Bibliothek ist für alle Betriebssysteme, auf denen ROS lauffähig ist, vorhanden, wird aktiv weiterentwickelt und bietet ein flexibles Lizenzmodell an. Neben der Tatsache, dass es die von den ROS-Entwicklern favorisierte Graphik-Bibliothek ist, lassen sich Qt-Oberflächen leicht erweitern und in C++-Programme integrieren. Qt-Oberflächen und die zugehörige C++-Programme können außerdem gut in ROS-Pakete eingefügt werden.

Der *rxDeveloper* verwendet neben Qt und ROS noch zwei wichtige Bibliotheken, welche automatisch bezogen werden können und somit keinen Mehraufwand für den Benutzer darstellen. Für das Lesen (Parsing) und Schreiben (Emitting) von Node-Spezifikationsdateien wurde die *YAML-CPP*-Bibliothek verwendet. Diese C++-Bibliothek wird in *ROS Diamondback*³ bereits mitgeliefert oder lässt sich seit *ROS Electric Emys*⁴ bequem als Systemabhängigkeit über ROS-Tools installieren. Das Lesen und Schreiben der Launchfiles erfolgt mittels *TINYXML*. Auch diese C++-Bibliothek ist wie auch *YAML-CPP* sehr leichtgewichtig und, je nach verwendeter ROS Distribution, verfügbar. *TINYXML* und *YAML-CPP* werden beide in anderen ROS-Tools verwendet und sind somit eine gute Wahl in Hinblick auf eine mögliche Weiterentwicklung seitens der Community. Die Entscheidungen zugunsten dieser und gegen alternative Bibliotheken werden von der Community gestützt und wurden in den Befragung (siehe Kapitel 4.1) zuvor evaluiert.

³ ROS Diamondback Distribution, veröffentlicht am 02. März 2011

⁴ ROS Electric Emys Distribution, veröffentlicht am 30. August 2011

Durch diese Entscheidungen und die damit verbundenen geringen Abhängigkeiten, ist das Programm schnell, ressourcensparend und kann als eigenständiges Programm ausgeführt werden. Eine Implementierung als Plugin für eine IDE, zum Beispiel Eclipse, hätte diese Anforderungen nicht erfüllt. Der *rxDeveloper* ermöglicht durch die Verwendung von ROS eine grafische komponentenbasierte Softwareentwicklung mit Komponenten aus verschiedenen Programmiersprachen. Das Programm und seine Konzepte sind als Open-Source-Projekt veröffentlicht, so dass eine einfache Verbreitung und unproblematische Verwendung ermöglicht wird.

4.3.2. Ausführung des Programms

Die Ausführung des Programms erfolgt nach der Kompilierung über die Kommandozeile. Die Binary-Datei *rxdev* kann entweder aus dem *bin*-Verzeichnis direkt aufgerufen werden oder, da das Programm als ROS-Package veröffentlicht wird, durch einen *rosrun*-Befehl wie in Listing 4.5 dargestellt. Dadurch ist es unabhängig vom momentanen Arbeitsverzeichnis. Wird zusätzlich noch der Pfad zu einem gültigen Launchfile angegeben, zum Beispiel durch „*rxdev pfad/zum/file.launch*“, so wird dieser Launchfile umgehend mit dem *rxDeveloper* geöffnet.

Listing 4.5: Programmstart per rosrun

```
rosrun rxDev rxdev
```

4.3.3. Funktionalität

Die Hauptaufgabe des *rxDeveloper* ist die Erstellung und Weiterverarbeitung von Launchfiles. Er ermöglicht das Lesen, Schreiben und Starten von syntaktisch korrekten Launchfiles und stellt diese dabei zur Weiterverarbeitet intuitiv, als Graph dar. Alle der in Kapitel 2.4.1 aufgeführten Elemente des Launchfiles können verwendet werden. Der Berechnungsgraph von ROS wird durch die Nodes und eventuelle Remaps beschrieben. Abbildung 4.2 zeigt schematisch wie zwei Komponenten (Nodes), die über einen Connector (Topic-Remap, siehe Kapitel 2.4.1) verbunden sind, präsentiert werden. Der Connector bietet dabei zwei Informationen:

1. die Beschriftung: Name, den das Topic erhalten soll
2. die Pfeilrichtung: zeigt immer vom Publisher auf den Subscriber

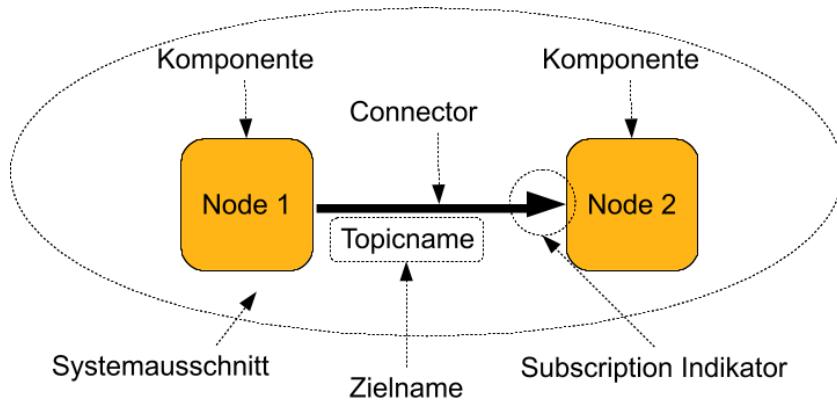


Abbildung 4.2.: Schematische Darstellung der Elemente bei einem Remap. *Node 1* publiziert ein Topic, welches von *Node 2* abgerufen wird.

Welches Topic die Umbenennung erfährt wird aus Gründen der Übersichtlichkeit an dieser Stelle nicht aufgeführt. Diese Information erhält man im zugehörigen Remap-Topic-Editor (siehe Kapitel 4.3.5). Durch die Graphdarstellung wird dem Benutzer ein weit höherer Grad an Übersichtlichkeit gegenüber der XML-Struktur der textuellen Launchfiles gegeben. Die implementierten Assistenzfunktionen, wie die Element-Editoren, der Parameter- und Remapassistent (siehe Kapitel 4.3.5), ermöglichen eine produktivere Erstellung von neuen Funktionen in den Launchfiles und sorgen für stets syntaktisch und semantisch korrekte Dateien.

Der *rxDeveloper* beinhaltet noch weitere Unterstützungsmöglichkeiten. So bietet er einen eingebauten Editor für die Erstellung und Anpassung von Spezifikationsdateien (siehe Kapitel 4.3.8) und grafische Hilfsmittel zur Verwaltung von ROS-Paketen. Durch diese Hilfsmittel können übergeordneten ROS-Pakete erstellt, betrachtet und verändert sowie neue Node-Quelltextdateien für C++ oder Python angelegt werden.

Das Konzept der Includefiles (siehe Kapitel 2.4.1) wurde konsequent weiterentwickelt und um mehr Struktur erweitert. Auf diese Weise können im *rxDeveloper* häufig verwendete Node-Kombinationen, beziehungsweise alle in Launchfiles möglichen Tag-Zusammenstellungen, als Komponenten abgespeichert werden und sind anschließend für den Benutzer verfügbar. Damit ist es möglich neue Komponenten aus mehreren atomaren Komponenten zusammenzusetzen und diese als Ganzes weiterzuverwenden. Listing 4.6 zeigt den gültigen Dateipfad für diese hierarchisch höheren Komponentendateien, so dass diese unter dem angegebenen Namen im *rxDeveloper* erscheinen und verwendet werden können. Wie schon bei der Speicherung der Spezifikationsdateien (siehe Kapitel 4.2.2) ist das Paket frei wählbar

und muss nicht mit dem Komponenteninhalt in Verbindung stehen, es wäre aber aus Gründen der Zuordnung zu empfehlen. Die Komponenten werden als Launchfiles gespeichert und zunächst über einen <include>-Tag eingefügt. Durch eine Option im Includefile-Editor (siehe Kapitel 4.3.5) des *rxDeveloper* können vorhandene Includes expandiert werden. Das bedeutet, dass ihr Inhalt in die graphische Darstellung überführt werden kann, um anschließend individuelle Anpassungen an den atomaren Komponenten vorzunehmen, sie neu zu speichern oder in andere Kompositionen zu integrieren.

Listing 4.6: Komponentendateien - Dateipfadkonvention

```
<package path>/component/name.launch
```

Das Starten der Launchfiles aus der graphischen Oberfläche heraus wurde für Testing-Zwecke implementiert. Des Weiteren können durch den *rxDeveloper* auch zusätzliche ROS-Tools, wie *rxgraph*, *rosutf*, *rxloggererlevel*, *rxconsole* und die *dynamic reconfigure gui* gestartet werden. Auf diese Weise werden Testing und Debugging in gewohnter Umgebung möglich und erleichtert.

4.3.4. Der Component Connector

Der *Component Connector* bezeichnet die Hauptansicht im *rxDeveloper* und wird in Abbildung 4.3 gezeigt. Er besteht aus der Arbeitsfläche (siehe 1.) und der Seitenleiste (siehe 2.). In der Arbeitsfläche werden die Launchfiles graphisch dargestellt. Die Seitenleiste beinhaltet Templates für sämtliche Elemente (siehe 3.), die in Launchfiles möglich sind (siehe Kapitel 2.4.1), sowie alle Nodes, für die Spezifikationsdateien hinterlegt wurden (4.). Mit den Tag-Buttons aus (3.) können auch beliebige Nodes und Nodelets erstellt werden. In einem Tab (5.) der Seitenleiste werden die bereits gespeicherte zusammengesetzte Komponenten, die in Kapitel 4.3.3 erläutert wurden, angezeigt.

Abbildung 4.4(a) zeigt wie im Node-Browser (2.) die Nodes, für die Spezifikationsdateien hinterlegt wurden, durchsucht und ihre Eigenschaften betrachtet werden können. Per Drag&Drop können diese Nodes der Arbeitsfläche hinzugefügt werden oder per Rechtsklick-Menü der Specfile-Editor (siehe Kapitel 4.3.8) geöffnet werden. Um die Spezifikationsdateien zu finden, greift der *rxDeveloper* auf ROS-Tools zurück, um alle Pakete des Systems zu durchsuchen. Im Komponenten-Tab (siehe Abbildung 4.4(b)) können zusammengesetzte Komponenten, die für das Pro-

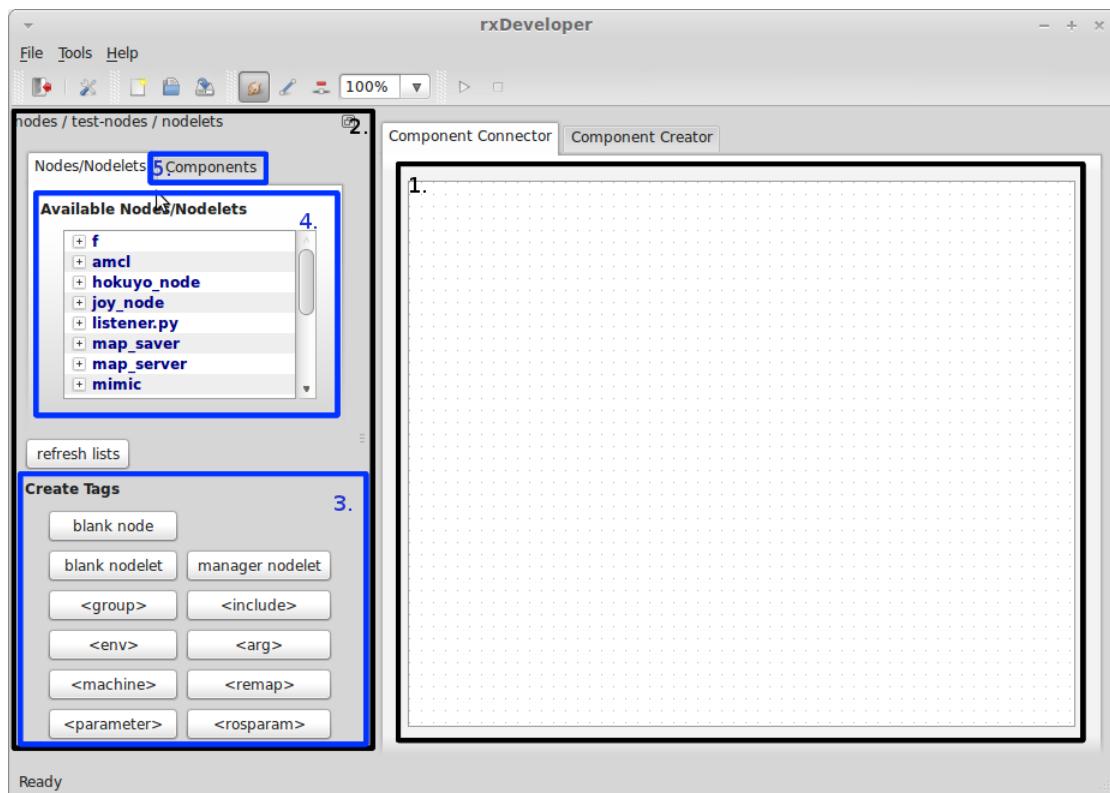


Abbildung 4.3.: rxDeveloper - Component Connector: 1. Arbeitsfläche; 2. Seitenleiste; 3. mögliche Launchfile-Elemente; 4. verfügbare Komponenten; 5. Tab für abgespeicherte zusammengesetzte Komponenten

gramm hinterlegt wurden, ausgewählt werden. Diese können per Doppelklick oder per Rechtsklick-Menü der Arbeitsfläche hinzugefügt oder auch geöffnet und so betrachtet werden. Auch die Komponentendateien werden durch absuchen aller Pakete mit Hilfe von ROS-Tools zusammengetragen.

Abbildung 4.5 zeigt die Toolbar, in der häufig verwendete Aktionen aufgeführt werden. Um Elemente auf der Arbeitsfläche zu verändern, werden die verschiedenen Arbeitsmodi genutzt. Die drei Arbeitsmodi der Arbeitsfläche, *Drag & Drop* (1.), *Remap* (2.) und *Delete* (3.) können in der Toolbar umgeschaltet werden. Im *Drag & Drop*-Modus können die Elemente vom Benutzer auf der Arbeitsfläche beliebig verschoben und teilweise ineinander gezogen werden. Elemente, die mit Doppelklick angewählt werden, öffnen den zugehörigen Elementeditor (siehe Kapitel 4.3.5) und lassen sich darin ändern. Im *Remap*-Modus werden Remap-Tags für Topic-Remappings durch ziehen einer Linie von einem Node zu einem anderen erstellt und im *Delete*-Modus können einzelne Elemente per Mausklick entfernt

4. Implementierung des Programms

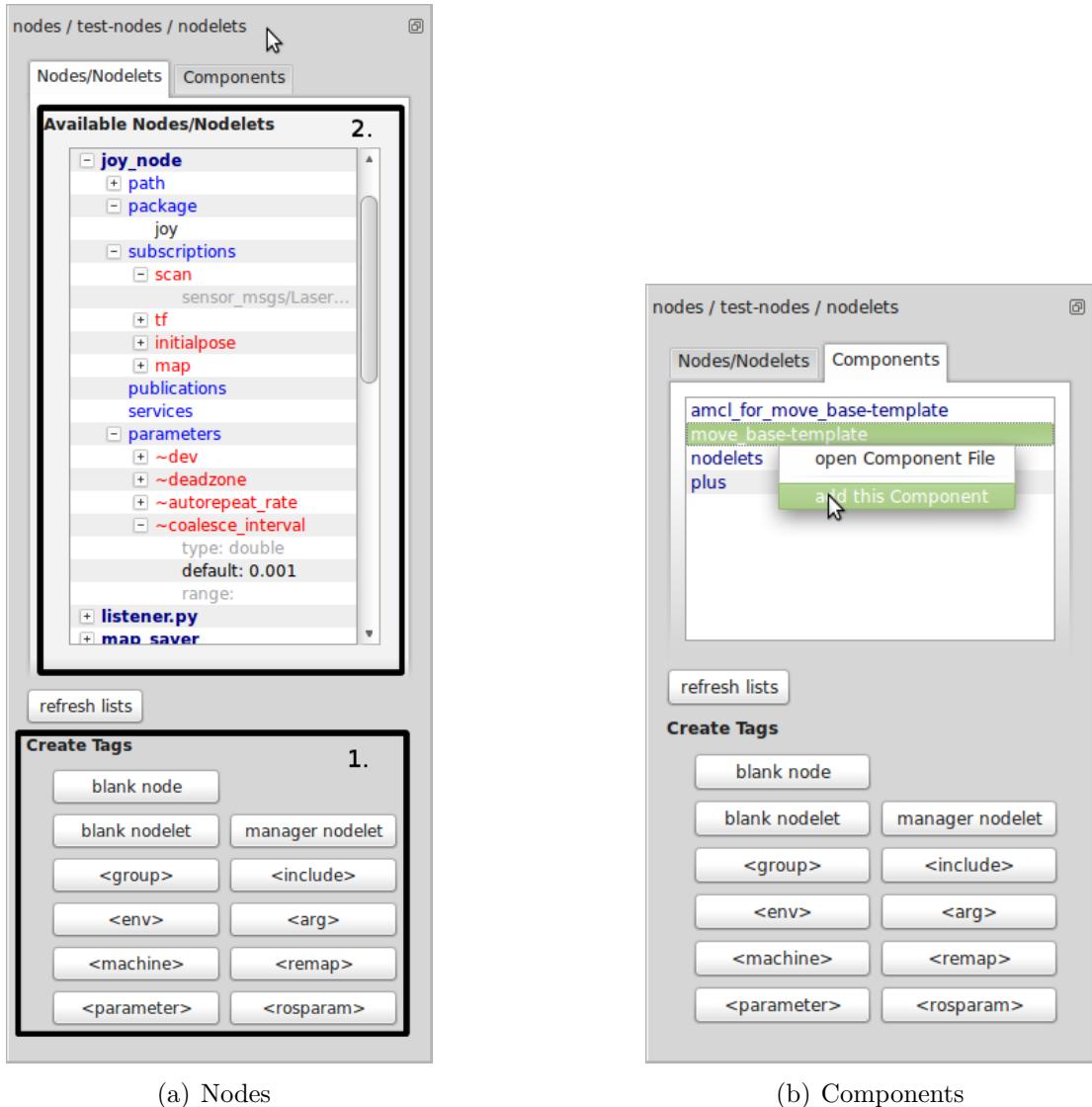


Abbildung 4.4.: rxDeveloper - Seitenleiste: (a) Node-Tab - 1. Tag-Buttons; 2. Browser für vorhandene Nodes; (b) Component-Tab

werden. Das Anlegen einer neuen Arbeitsfläche und das Speichern der Arbeitsfläche des *Component Connectors* als Launchfile sowie das Laden von existierenden Launchfiles kann durch die Symbole in 4. oder das Programm-Menü erfolgen. Will der Benutzer einen Launchfile testen, so existieren hierfür die Aktionssymbole Start und Stop unter 5. Außerdem kann die Zoomstufe der Arbeitsfläche (6.) bestimmt werden, um sich bei größeren Launchfiles mehr Überblick zu verschaffen.



Abbildung 4.5.: rxDeveloper - Toolbar und Menü: 1. Drag&Drop-Modus; 2. Remap-Modus; 3. Delete-Modus; 4. Gruppe (neu, öffnen, speichern) für Launchfiles; 5. Start/Stop; 6. Zoom Arbeitsfläche

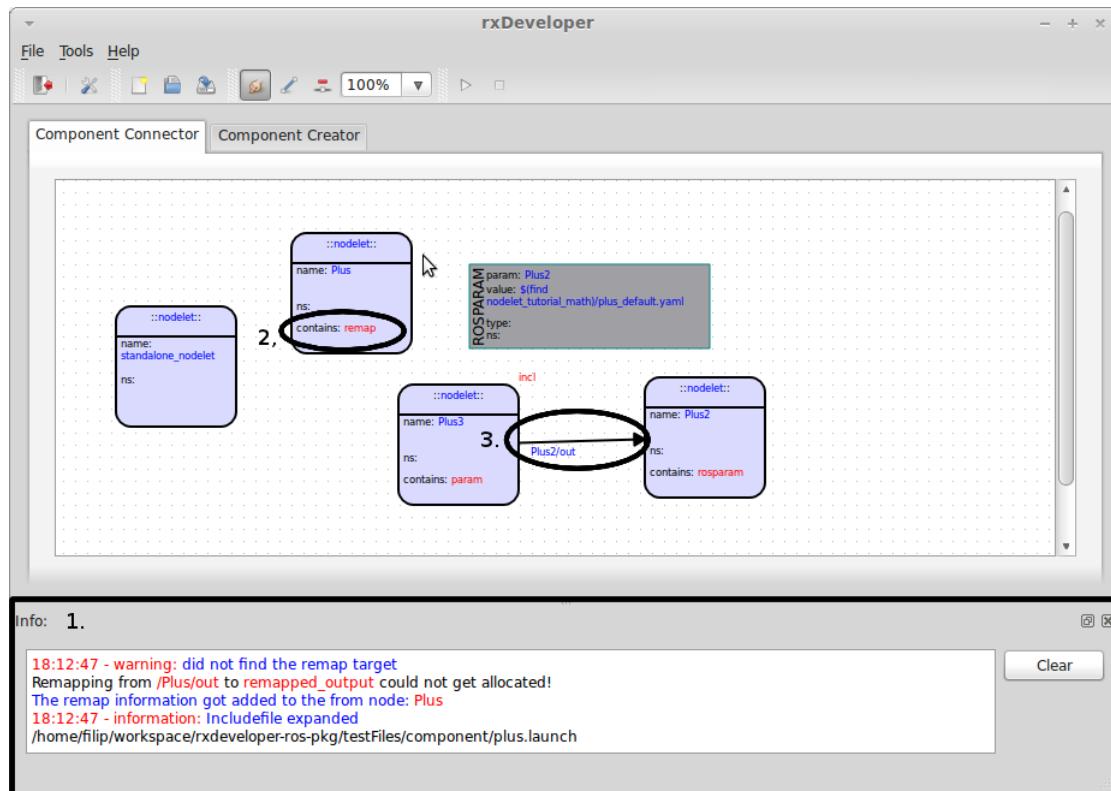


Abbildung 4.6.: rxDeveloper - Informationsleiste: 1. Informationsleiste; 2. missglückter Topic-Remap; 3. geglückter Topic-Remap

Beim Laden von vorhandenen Launchfiles versucht das Tool Remaps von Nodetopics per Remap-Pfeil darzustellen. Ist eine Zuordnung des Remaptopics zu seinem Ziel nicht möglich, zum Beispiel wenn im Remap-Tag keine globalen Namen (siehe Kapitel 2.3.7) verwendet wurden oder der Zielknoten nicht bekannt ist, so existiert ein Fallbackmodus, in dem der Remap in dem Startnode aufgeführt wird und dort weiterverarbeitet werden kann (siehe Abbildung 4.6, 2.). Falls dies geschieht, wird der Benutzer außerdem per *Warning* in einer Informationsleiste (siehe Abbildung 4.6, 1.) darauf hingewiesen. Ein missglücktes Remapping kann auch eine potentielle Fehlerquelle im Launchfile darstellen.

4. Implementierung des Programms

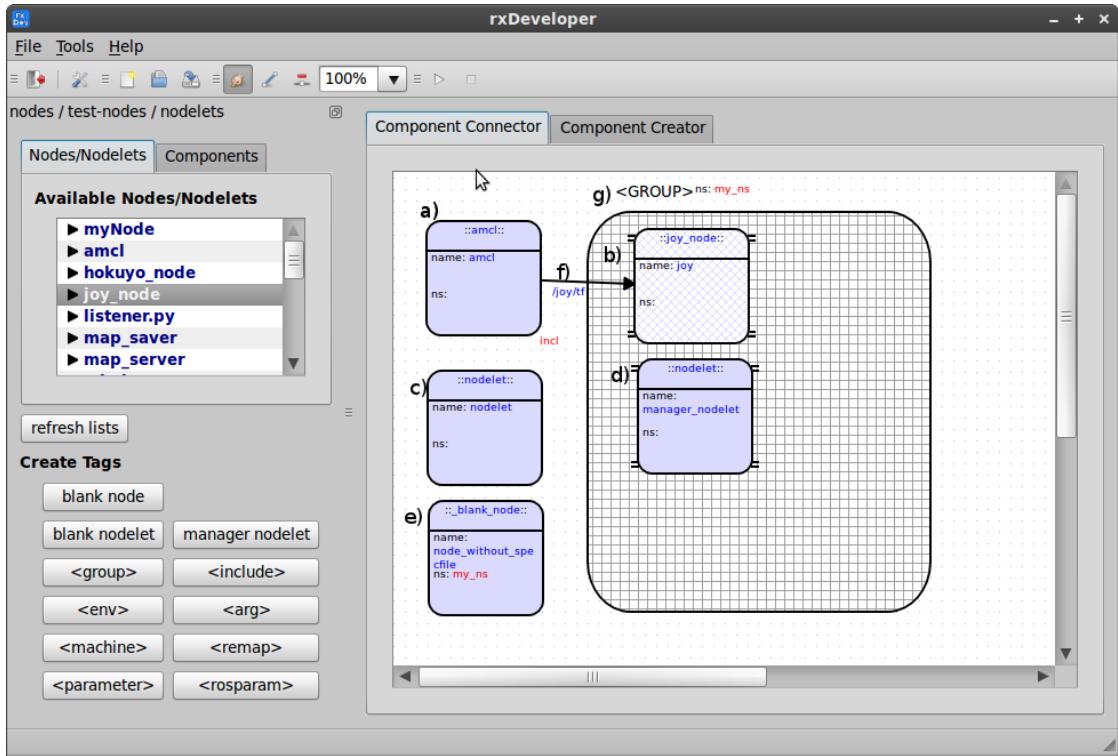


Abbildung 4.7.: rxDeveloper - ROS-Prozesse: a) Standard-Node erstellt aus einer Spezifikationsdatei; b) Test-Node erstellt aus einer Spezifikationsdatei; c) Nodelet erstellt aus einem Template; d) Nodelet-Manager erstellt aus einem Template; e) Standard-Node erstellt aus einem Template mit Angabe eines Namespaces; g) Group-Element mit Namespace welches b) und d) beinhaltet; f) Darstellung eines Topic-Remaps von einem Publish-Topic in a) auf ein Subscribe-Topic in b) und zugeordnet in a) (impliziert durch rotes "incl")

Falls ein Launchfile bereits einmal mit dem *rxDeveloper* abgespeichert wurde, so wird die graphische Darstellung der Tags automatisch wiederhergestellt. Dies wird durch in Kommentaren versteckte Metadaten in den Launchfiles erreicht, die für jedes Tag-Element die Koordinaten beziehungsweise, im Falle eines Group-Elements, auch die Dimensionen enthalten. Die Metadaten haben den Vorteil, dass kein eigenes Dateiformat für den *rxDeveloper* benötigt wird und es dem Benutzer somit ohne Umstände möglich ist, Launchfiles, die im *rxDeveloper* entstanden sind, auch ohne diesen weiter zu verarbeiten.

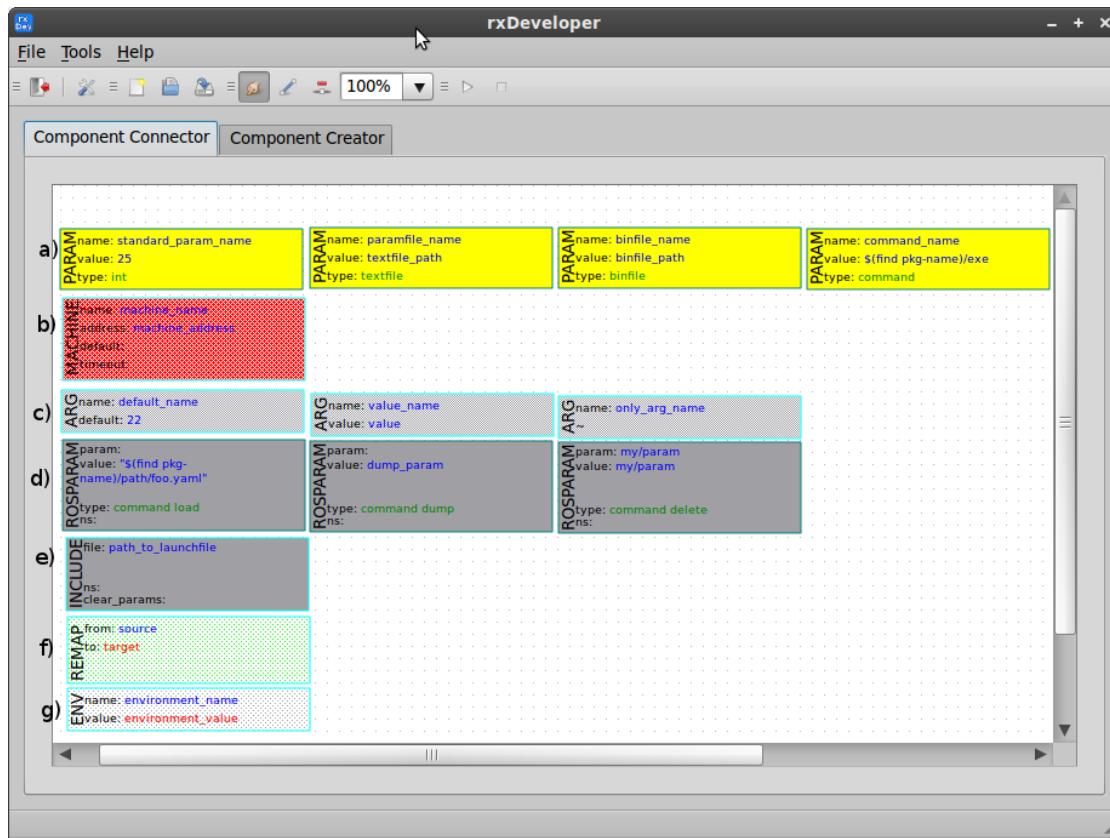


Abbildung 4.8.: rxDeveloper - weitere Launchfile-Elemente: a) verschiedene Parameter-Tags, b) Machine-Tag, c) verschiedene Arg-Tags, d) unterschiedliche Rosparam-Tags, e) Include-Tag, f) allgemeiner Remap-Tag (im Gegensatz zum expliziten Topic-Remap zwischen Nodes, siehe Abb. 4.7), g) Env-Tag

Darstellung der Launchfile-Elemente

Launchfiles können eine Vielzahl unterschiedlicher Elemente (siehe Kapitel 2.4.1) beinhalten, welche wiederum unterschiedlich konfiguriert sein können. Die Abbildungen 4.7 und 4.8 zeigen die unterschiedlichen Darstellungen der Launchfile-Elemente. Die wichtigsten Informationen werden im *rxDeveloper* direkt grafisch dargestellt. Alle weiteren Informationen können in den jeweiligen Elementeditoren (siehe Kapitel 4.3.5) betrachtet und verändert werden. Remaps, die durch einen Remap-Pfeil dargestellt werden erhalten eine zusätzliche Kennzeichnung des Nodes, für den der Remap ausgeführt wird (siehe Abbildung 4.7, a) u. f)). Die Pfeilrichtung zeigt stets vom Node des Publication-Tops auf auf den Node des

Subscription-Topics, wobei beide Nodes mögliche Positionen für ein Remapping-Tag sind. Die explizite Kennzeichnung des Nodes durch den roten Textzug „incl“, gibt dem Benutzer die Information darüber, wo der Remap-Tag letztlich in der textuellen Form des Launchfiles steht. Der Launch-Tag enthält bei ROS keine besondere Funktion in einem Launchfile, außer den Beginn und das Ende zu markieren. Er wird als implizit durch die Arbeitsfläche vorhanden angenommen und kann nicht manuell erzeugt werden. Da er ein optionales Attribut besitzt, existiert hierfür, wie für andere Elemente auch, ein Editor. Dieser Editor ist durch einen Doppelklick auf einen freien Teil der Arbeitsfläche aufrufbar.

Die Elemente können in der grafischen Darstellung frei bewegt und in manchen Fällen, wenn die Launchfilesyntax (siehe Kapitel 2.4.2) es zulässt, auch ineinander geschoben werden. Nodes können zum Beispiel Parameter-, Rosparam-, Remap- und Env-Tags enthalten. Beinhaltet ein Element mindestens ein Element eines anderen Typs, so wird dies durch einen weiteren Schriftzug „contains: [element-type]“ innerhalb des Elements dem Benutzer mitgeteilt. Ausnahme ist hier das Gruppen-Element `<group>`, welches die eingefügten Elemente in einem abgetrennten Arbeitsbereich, bis auf leichte graphische Modifikationen unverändert, darstellt. Abbildung 4.9 zeigt diese beiden Darstellungsformen. Hier beinhaltet das Gruppen-Element einen Node und einen Parameter. Die Zugehörigkeit zur Gruppe wird durch kleine Klammern an den Elementen angezeigt. Der Node beinhaltet seinerseits weitere Elemente vom Typ *remap*, *param* und *rosparam*.

4.3.5. Element-Editoren und Assistenten

Beim Hinzufügen eines Elements öffnet sich stets der zugehörige Editor, damit das Element mit Detailinformationen beschrieben werden kann. Existierende Elemente können weiter angepasst werden, indem ein Doppelklick auf ihnen ausgeführt wird. Dies führt dazu, dass der jeweilige Element-Editor geöffnet wird. Für jeden Elementtyp existieren ein exakt zugeschnittener Editor, der die zugehörigen Pflicht- und optionalen Attribute anbietet sowie es ermöglicht, andere untergeordnete Elemente zu verwalten. Für Attribute, die nur bestimmte andere Attribute erlauben, werden in den Editoren ungültige weitere Eingaben deaktiviert, so dass nur Launchfile-konforme Kombinationen, entsprechend der Angaben aus Kapitel 2.4.1, kreiert werden können. Einige besondere Editoren werden im folgenden exemplarisch näher erläutert:

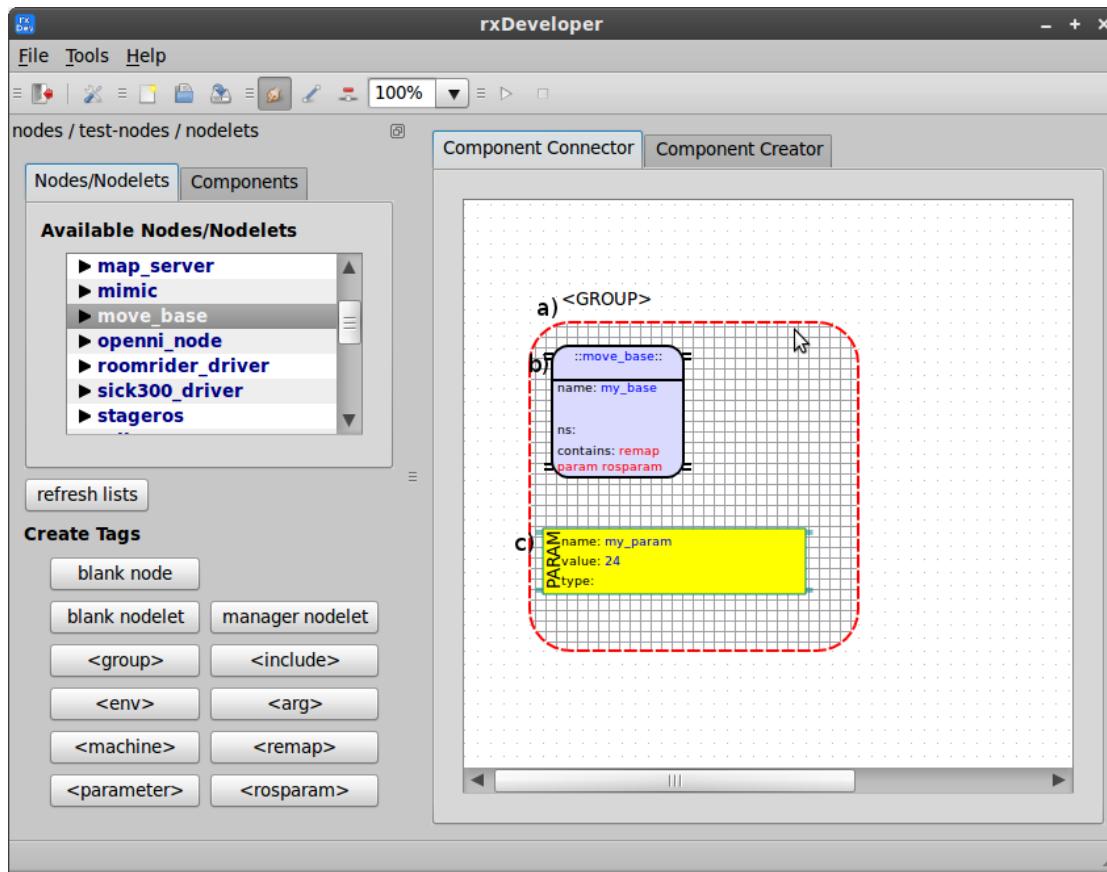


Abbildung 4.9.: rxDeveloper - verschachtelte Elementen: a) Group-Element, welches b) und c) beinhaltet. Elemente in einem Group-Bereich, werden mit kleinen Klammern dargestellt; b) Node-Element im Group-Element, der jeweils mindestens einen Remap-, einen Param- und einen Rosparam-Tag enthält (impliziert durch "contains: [element-type]"); c) Param-Tag im Group-Element.

Node-Editor

Der Node-Editor ist ein sehr wichtiger Editor unter den Element-Editoren. Er ermöglicht das Anpassen von neuen oder existierenden Node-, Nodelet- oder Testnode-Elementen. Mit diesem Editor (siehe Abbildung 4.10) ist es neben dem Anpassen aller Node-Eigenschaften (1.) auch möglich die, nach der Syntax von Launchfiles, legalen untergeordneten Elemente (2.) hinzuzufügen (a.), zu verändern, oder zu entfernen (b.). Die Verwaltung der untergeordneten Elemente kann hierbei für die jeweiligen Elemente über die Tabs durchgewechselt werden. Das

4. Implementierung des Programms

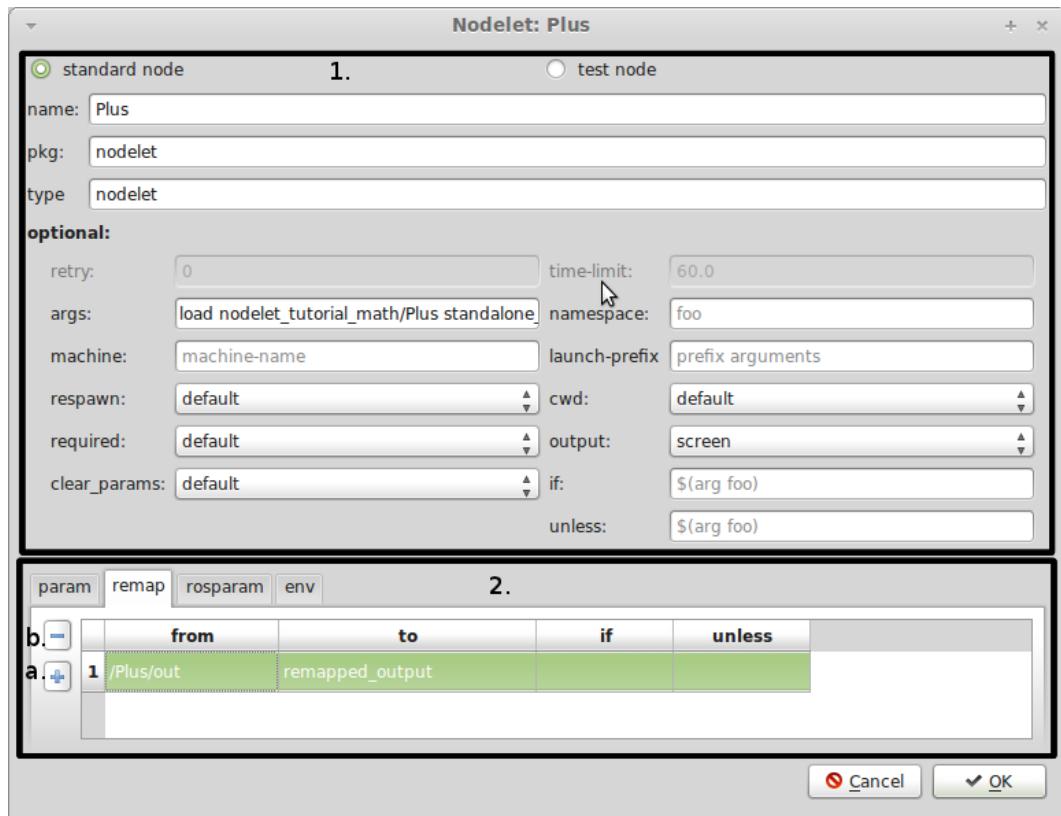


Abbildung 4.10.: rxDeveloper - Node-Editor: 1. Elementattribute; 2. Verwaltung der untergeordneten Elemente; a. neues untergeordnetes Element hinzufügen; b. untergeordnetes Element löschen

Verändern funktioniert durch einen Doppelklick auf das gewünschte untergeordnete Element, woraufhin sich dafür der entsprechende Element-Editor öffnet. Auch beim Hinzufügen neuer Elemente öffnet sich der entsprechende Editor. Abbildung 4.11 zeigt eine Assistenzfunktion, die beim Parametrisieren von Nodes hilft, für die Spezifikationsdateien angelegt wurden. Wird für einen solchen Node ein Parameter hinzugefügt, so kann man den gewünschten Parameter aus einer Liste aussuchen und anschließend im Parameter-Editor ändern. Dabei werden im Parameter-Editor die zuvor ausgewählten Informationen automatisch zunächst mit Default-Wert eingegeben. Handelt es sich bei dem zu editierenden Node um einen Dynamic Reconfigure Node und ist der ausgewählte Parameter ein dynamisch rekonfigurierbarer Parameter, so findet nach dem Hinzufügen auch eine Überprüfung des eingegebenen Wertes statt. Liegt der Wert außerhalb des gültigen Bereichs, der durch *range* in der Spezifikationsdatei angegeben wurde, so wird der Benutzer darüber mittels eines Dialogfensters informiert und kann den Wert entsprechend korrigieren.

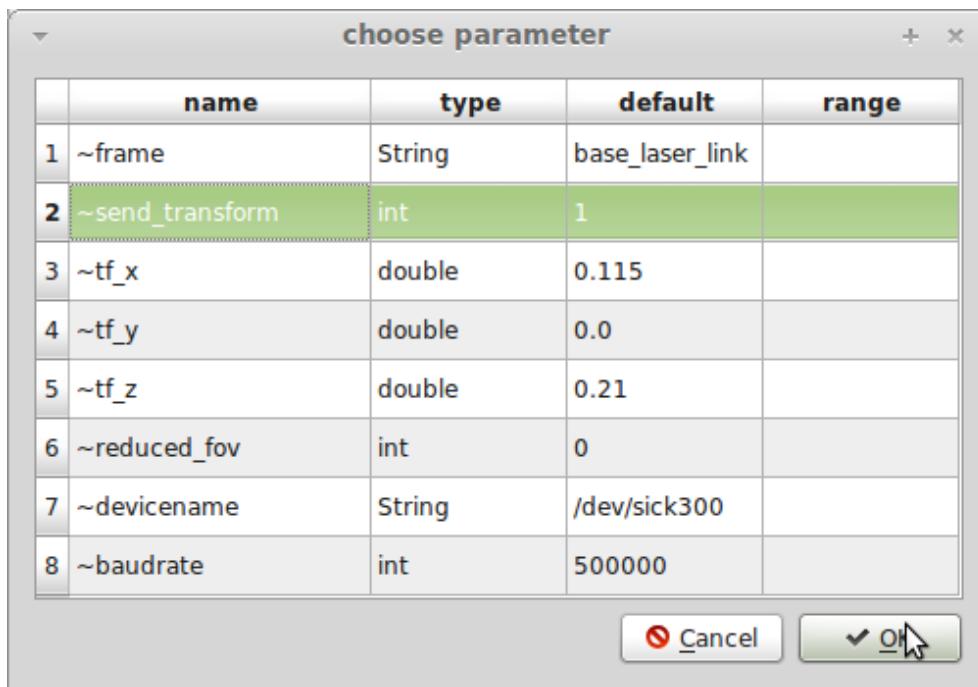


Abbildung 4.11.: rxDeveloper - Parameter-Assistent

Remap-Topic-Editor

Der Remap-Topic-Editor, der bei einem Doppelklick auf Remap-Pfeile oder beim Anlegen von Remaps im Remap-Arbeitsmodus (siehe Abbildung 4.5, 2.) erscheint, wird in Abbildung 4.12 dargestellt. In diesem kann durch Auswahl das gewünschte Remapping-Verhalten bestimmt werden, wobei der Assistent dem Benutzer nur die möglichen Topics zur Auswahl stellt, wenn diese in der Spezifikationsdatei hinterlegt wurden. Im Editor sind Remappings von Subscription-Tops auf Publication-Tops genauso möglich wie der umgekehrte Fall oder eine manuelle Eingabe. Die manuelle Eingabe ähnelt dann der des Editors für allgemeine Ressourcen-Remaps, welcher auch nur die Eingabe für die Attribute *from* und *to* benötigt.

Include-Editor

Der Include-Editor bietet wie auch andere Editoren die Möglichkeit einige Attribute zu editieren (Siehe Abbildung 4.13). Darüber hinaus können die Dateien, die hierüber eingebunden werden sollen, geöffnet und somit betrachtet werden (1.) und das Include-Element kann expandiert werden (2.). Expandieren überführt die

4. Implementierung des Programms

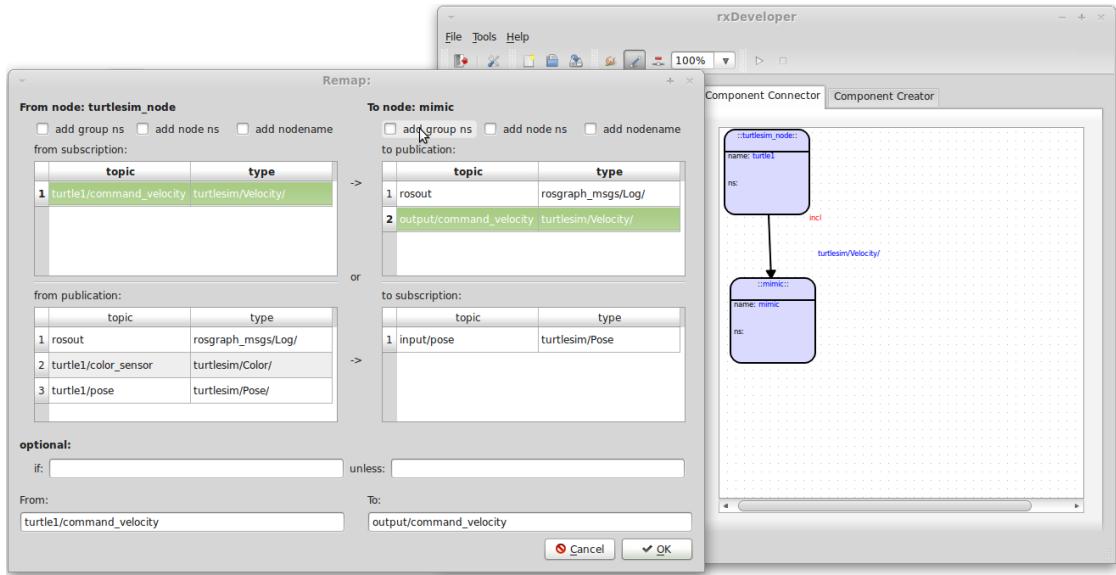


Abbildung 4.12.: rxDeveloper - Remap-Topic-Editor

inkludierte Datei in die *rxDeveloper*-Darstellung und entspricht somit einem Import der Elemente der Datei in das Programm. Befindet sich das Include-Element dabei in einem Group-Element, so werden auch die Elemente aus der Datei in das Group-Element gesetzt.

4.3.6. Testing/Debugging

Wie bereits in Kapitel 4.3.4 beschrieben können Launchfiles vor der Speicherung testweise ausgeführt werden. Mögliche Remap-Fehler werden in einer Informationsleiste angezeigt und können vom Benutzer kontrolliert werden. Aus der Menüleiste heraus können unter dem Punkt „Tools“ außerdem einige bekannte graphische Hilfsprogramme gestartet werden, die den Berechnungsgraphen und die Kommunikation analysieren. Zu diesen ROS-Tools gehören *rxgraph*, *rviz*, *rxloggererlevel*, *rxconsole* und die *dynamic reconfigure gui*, die dem Entwickler eine gewohnte Arbeitsumgebung bieten. Ein weiteres Debuggingtool ist *roswtf*, dessen Analyse nach Ausführung in der Informationsleiste erscheint.

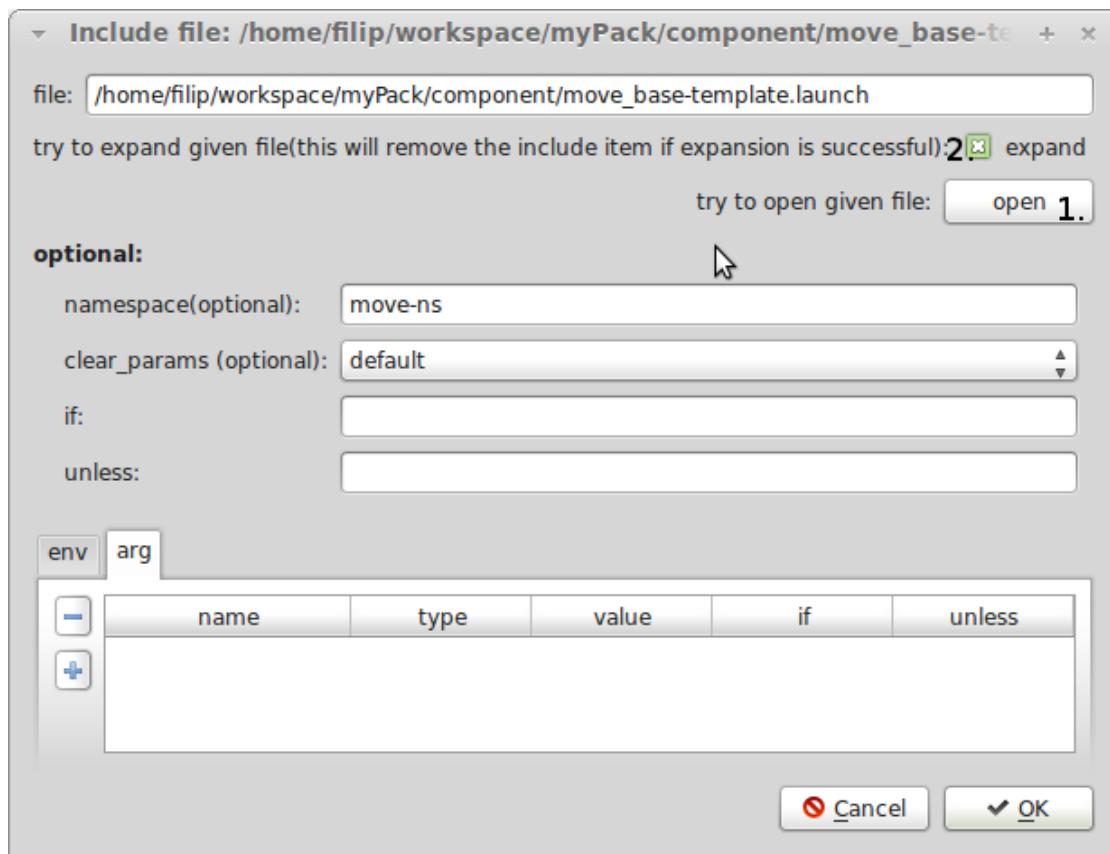


Abbildung 4.13.: rxDeveloper - Include-Editor: 1. Öffnen der angegebenen Datei;
2. Extrahieren der Informationen aus der Datei

4.3.7. Der Component Creator

Abbildung 4.14 zeigt den *Component Creator*, der dafür gedacht ist Pakete zu verwalten. Sobald der Benutzer vom *Component Connector* in den *Component Creator* wechselt wird aus der Arbeitsfläche ein *Packagebrowser* (1.) mit einigen Erstellungsmöglichkeiten für Dateien (2.) und Pakete (3.). In der Seitenleiste werden nun die Pakete angezeigt (4.), die vom, in den Einstellungen angegebenen, Arbeitsverzeichnis (*working directory*) aus erreichbar sind. Legt der Benutzer ein neues Paket an (3.), so wird dieses im *working directory* erstellt und ist umgehend in der Seitenleiste (4.) verfügbar. Per Mausklick wird ein hier ausgewähltes Paket im *Packagebrowser* (1.) dargestellt. Ordner und Dateien können über das Kontextmenü oder durch die Buttons im ausgewählten Paket angelegt, geöffnet oder entfernt werden. Wählt man die Aktion zum Erstellen einer neuen Spezifikationsdatei, so wird diese automatisch korrekt in einem Unterordner "node" im

4. Implementierung des Programms

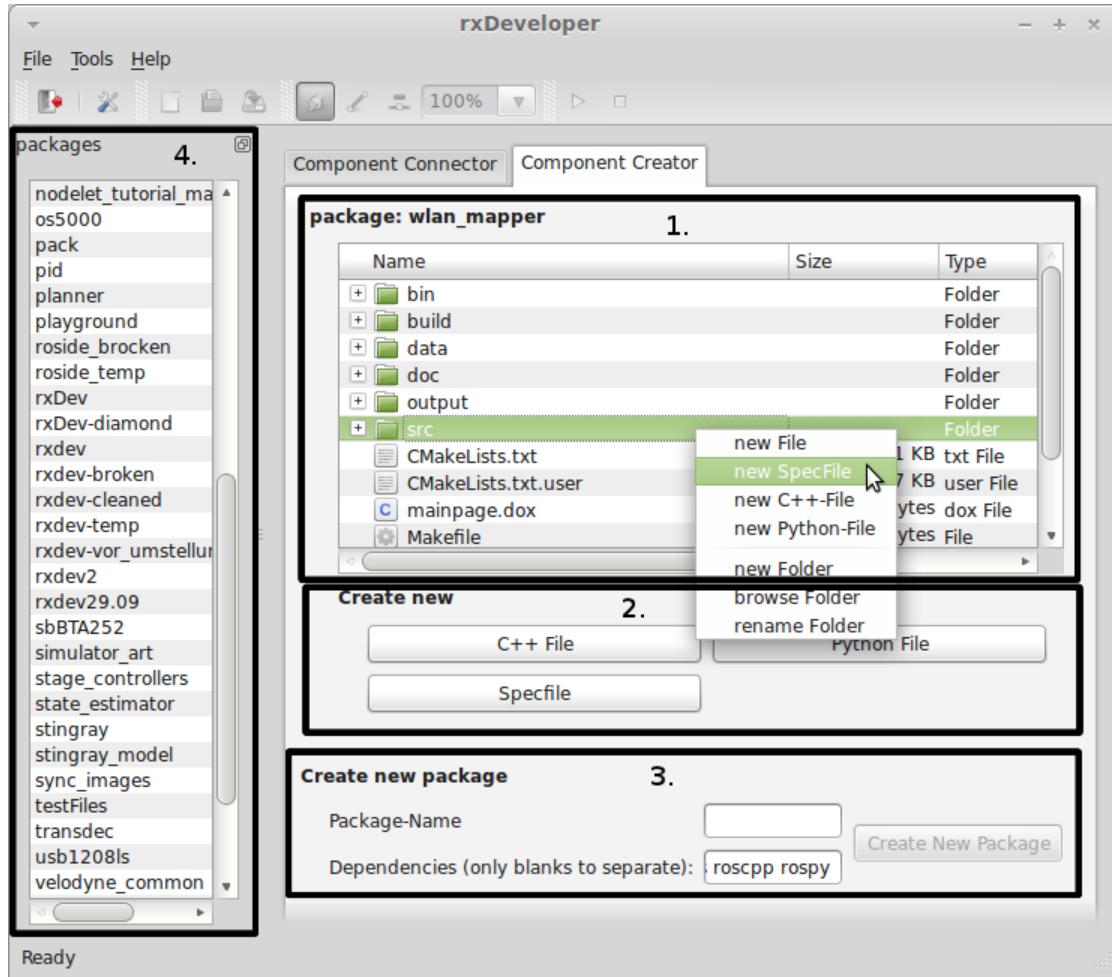


Abbildung 4.14.: rxDeveloper - Component Creator: 1. Package-Browser; 2. Anlegen neuer Dateien; 3. Anlegen eines neuen ROS-Paketes; 4. Seitenleiste mit vorhandenen Paketen zur Auswahl

Paket angelegt und kann von da an im *Component Connector* verwendet werden. Die Spezifikationsdetails müssen hierfür im *Specfile-Editor* (siehe Kapitel 4.3.8) eingetragen werden, der sich automatisch bei einer Neuerstellung einer Spezifikationsdatei öffnet. Die Aktionen zum Erstellen von neuen Quelltextdateien in C++ oder Python legen leere Dateien im ausgewählten Ordner an. Dem Benutzer werden somit sämtliche Freiheiten bei der Ordnerstruktur gelassen.

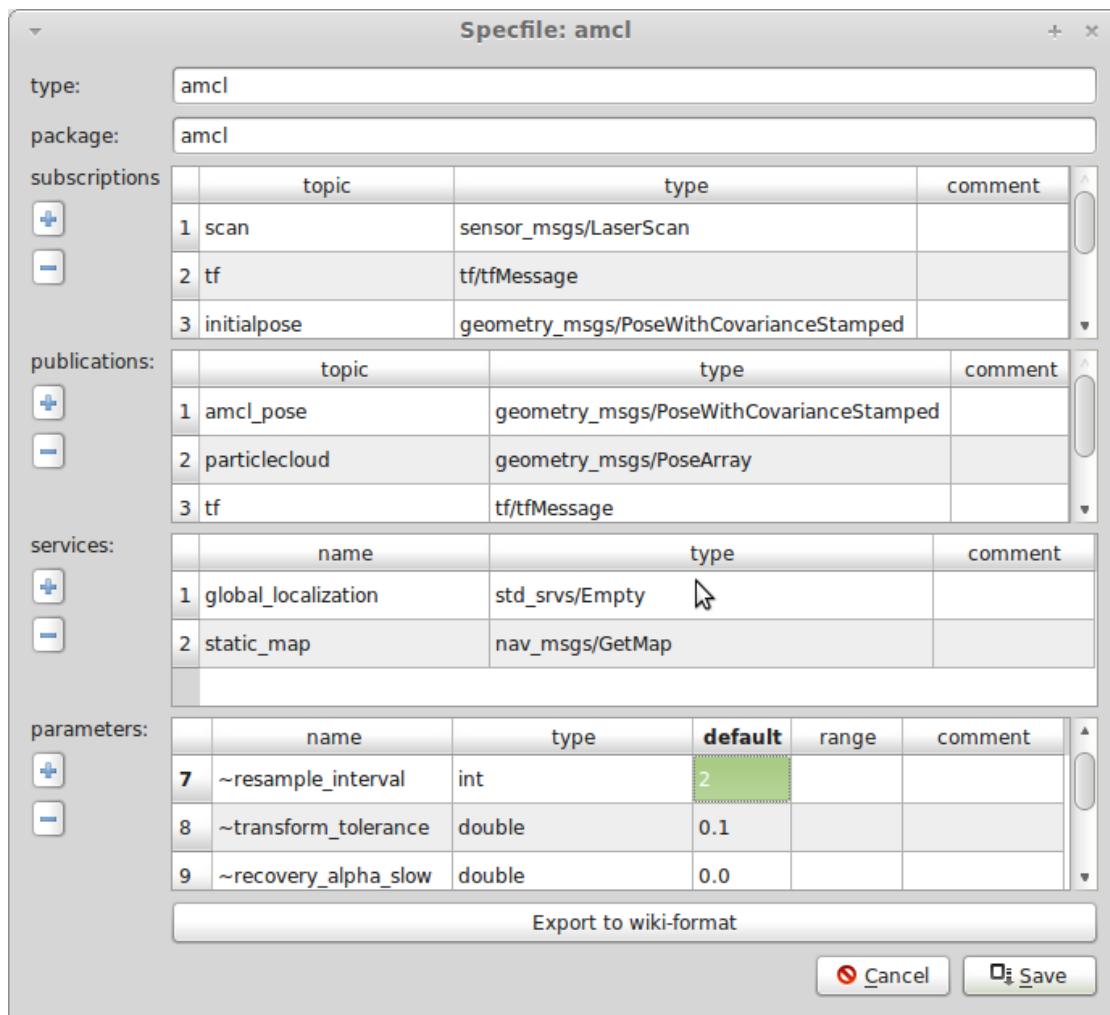


Abbildung 4.15.: rxDeveloper - Specfile-Editor

4.3.8. Specfile-Editor

Der *Specfile-Editor*, dargestellt in Abbildung 4.15, bietet eine komfortable Möglichkeit die YAML-formatierten Nodespezifikationsdateien zu manipulieren. Durch ihn können neue Spezifikationsdateien im Component Creator (siehe Kapitel 4.3.4) angelegt werden oder existierende verändert werden. Das Verändern vorhandener Spezifikationsdateien kann sowohl aus dem Component Connector als auch aus dem Component Creator (siehe Kapitel 4.3.7) erfolgen. Neben der Eingabe der Pflichtangaben „*type*“ und „*package*“ (1.) können auf übersichtliche Weise *subscriptions*, *publications*, *services* und *parameters* gelöscht, hinzugefügt und editiert werden (2.). Ein zusätzliches Feature stellt der Wiki-Export-Knopf (3.) dar. Nach

4. Implementierung des Programms

Betätigung wird eine Textdatei dargestellt, welche die getätigten Eingaben im Format der ROS-Wiki-Seite präsentiert. Dieser Text kann dann in die Zwischenablage kopiert und anschließend auf der Wiki-Seite des Pakets eingefügt werden. Diese Komfortfunktion erleichtert dem Node-Entwickler die Dokumentationsarbeit und ist ein weiteres Argument für die Verwendung von Spezifikationsdateien.

5 Kapitel 5

5 Evaluation und Bewertung

In diesem Kapitel wird die Verwendung des *rxDevelopers* näher betrachtet und die Leistungsfähigkeit überprüft. Zunächst wird in Kapitel 5.1 getestet, ob der *rxDeveloper* in der Lage ist, sämtliche Element- und Attribut-Kombinationen für Launchfiles zu erkennen und zu erstellen. In Kapitel 5.2 wird anschließend die Verwendung der Spezifikationsfiles betrachtet. Kapitel 5.3 bestätigt die Plattformunabhängigkeit des *rxDevelopers* bevor eine abschließende Bewertung des erzeugten Programms in Kapitel 5.4 erfolgt

5.1. Evaluation der Vollständigkeit

Um zu überprüfen, ob der *rxDeveloper* in der Lage ist Launchfiles vollständig zu verwenden und zu erstellen, wurden zunächst Auflistungen aller Launchfile-Möglichkeiten erstellt. Hierfür mussten folgende Punkte beachtet werden und systematisch aufgelistet werden:

- Alle Attribut-Möglichkeiten für einen jeweiligen Tag
- Alle Möglichkeiten für Tags in anderen Tags

Durch die objektorientierte Programmierung der graphischen Elemente im *rxDeveloper* und dem Aufbau des Parsers und Emitters für die Launchfiles im Programm müssen nicht alle Kombinationsmöglichkeiten einzeln betrachtet werden. Attribute werden einzeln gelesen, gesetzt und geschrieben, falls sie legalen Kombinationen entsprechen. Daher genügt es sicherzustellen, dass für alle Tags jedes erlaubte Attribut mit jedem möglichen Wert aufgelistet ist. Das Ergebnis ist Tabelle 5.1, die 78 Tags und ihre optionalen Attribute entsprechend legaler Verwendungen enthält. Des Weiteren müssen Tags, die anderen Tags untergeordnet werden können, in ihrer untergeordneten Rolle überprüft werden. Es müssen somit nicht alle vier hierarchischen Klassen (siehe Kapitel 2.4.2) in allen Kombinationen aufgeführt werden, sondern nur die jeweils direkten Unterordnungen. Alle weiteren

5. Evaluation und Bewertung

Verschachtelungen funktionieren nach dem gleichen Prinzip und müssen nicht einzeln betrachtet werden. Eine Darstellung der direkten Unterordnungen von Tags erfolgte bereits mit Tabelle 2.4, welche die 30 zu überprüfende Möglichkeiten beschreibt.

In mehreren Stufen wurden alle Möglichkeiten auf ihre Kompatibilität zum *rxDeveloper* überprüft:

1. Test mit existierenden Launchfiles aus der zugrunde liegenden ROS-Installation und der Arbeitsgruppe „Autonome Intelligente Systeme“.
 - Laden der existierenden Launchfiles und Überprüfung der korrekten Darstellung im *rxDeveloper*.
 - Speichern der Darstellung der zuvor geladenen Launchfiles und Vergleich der Ausgabe mit der Ursprünglichen Datei.
2. Erstellung eigener Launchfiles für noch nicht überprüfte Tag- und Attribut-Möglichkeiten.
 - Laden der erstellten Launchfiles und Überprüfung der korrekten Darstellung im *rxDeveloper*.
 - Speichern der Darstellung der zuvor geladenen Launchfiles und Vergleich der Ausgabe mit der Ursprünglichen Datei.
3. Systematischer Test der Erstellung aller aufgeführten Möglichkeiten.

Die existierenden und die neu erstellten Dateien sowie die über das Programm abgespeicherten Versionen dieser Dateien können im Appendix A.2, beziehungsweise unter der dort verlinkten Webseite, betrachtet werden.

Tag-Typ	Tag-Kombinationen
launch	launch launch + deprecated
group	group group + ns group + ns + clear_params='true' group + ns + clear_params='false' group + if group + unless

Tag-Typ	Tag-Kombinationen
node	node node + args node + respawn='true' node + respawn='false' node + launch-prefix node + machine node + output='screen' node + output='log' node + required='true' node + required='false' node + cwd='node' node + cwd='ROS_HOME' node + if node + unless node + ns
test	test test + launch-prefix test + required='true' test + required='false' test + ns test + args test + cwd='node' test + cwd='ROS_HOME' test + time-limit test + retry test + if test + unless
param	param + value param + value + type param + textfile param + binfile param + if param + unless param + command
remap	remap + if remap + unless remap

<i>Tag-Typ</i>	<i>Tag-Kombinationen</i>
machine	machine machine + default='true' machine + default='false' machine + default='never' machine + ros-root machine + ros-package-path machine + user machine + password machine +if machine + unless machine + timeout
arg	arg arg + default arg + if arg +unless arg + value
env	env env + if env + unless
rosparam	rosparam + file + command='load' rosparam + file + command='dump' rosparam + file + ns rosparam + raw YAML-text rosparam + param rosparam + param + delete rosparam + if rosparam + unless
include	include include + ns include + clear_params='true' include + clear_params='false' include + if include + unless

Tabelle 5.1.: Launchfile-Tags + eventuelle Attribute mit Wert (falls distinkтив), gruppiert nach dem Tag-Typ.

Ergebnisse

Zu 1.) Dieser Test enthielt 33 Launchfiles aus der Arbeitsgruppe und 13 weitere aus ROS. Durch den Test wurden häufige und aus der Praxis stammende Möglichkeiten getestet. Diese Launchfiles deckten 27 von 78 Möglichkeiten aus Tabelle 5.1 und 16 der 30 Unterordnungsmöglichkeiten aus Tabelle 2.4 ab. Die Darstellung nach dem Öffnen deckte die gleichen Möglichkeiten ab und die Speicherung führte zu funktionell äquivalenten Launchfiles.

Dabei ist anzumerken, dass manche der Dateien noch einen Tag enthielten, der in aktuellen ROS-Versionen nicht mehr unterstützt wird. Bei diesem Tag handelt es sich um den <master>-Tag, der häufig inklusive eines Attributes zum automatischen Laden des Master-Services auftrat (<master auto="start"/>). Dieser Tag wurde nicht dargestellt und somit auch nicht beim Speichern erzeugt. Da es sich hierbei um einen nicht mehr unterstützten Tag handelt, wird dieser allerdings auch nicht in der Liste der Möglichkeiten aufgeführt und findet auch keine weitere Beachtung in der Auswertung.

Zu 2.) 11 Dateien wurden erstellt, wovon zehn die 51 restlichen Möglichkeiten aus Tabelle 5.1 und eine die 14 verbleibenden Möglichkeiten aus Tabelle 2.4 enthält. Die Dateien, die entsprechend Kapitel 2.4 Launchfile-konform sind, wurden nur mit dem Ziel der Überprüfung der Möglichkeiten erstellt und ergeben kein sinnvolles, funktionierendes Programm. Auch hier deckte die Darstellung nach dem Öffnen die gleichen Möglichkeiten ab und die Speicherung führte zu äquivalenten Launchfiles.

Zu 3.) Da das Öffnen und Abspeichern aller Möglichkeiten in den ersten beiden Stufen bereits erfolgte, wurde in der dritten Stufe die Erstellung aller dieser Möglichkeiten im *rxDeveloper* systematisch überprüft. Hierbei ließen sich sämtliche Möglichkeiten erfolgreich darstellen und fehlerhafte Kombinationen wurden durch die Element-Editoren verhindert.

5.2. Nutzung der Spezifikationsfiles

Ein weiterer Verwendungstest bezieht sich auf die Spezifikationsdateien und wurde parallel durchgeführt. Im Lieferumfang des *rxDevelopers* befinden sich bereits 15 Spezifikationsfiles, die aus Testzwecken oder Aufgrund einer vermeintlichen häufigen Benutzung ausgewählt und erstellt wurden. Bei den zuvor durchgeföhrten

Überprüfungen der Launchfile-Möglichkeiten wurden Node-Elemente, für die solche Spezifikationsdateien hinterlegt wurden, automatisch erkannt und so deren Funktionalität dem Programm zur Verfügung gestellt. Alle per Drag & Drop auf der Arbeitsfläche erzeugten Nodes, welche aus Spezifikationsdateien generiert wurden, stellten ebenso die angegebene Eigenschaften von Parametern, Services, Publications und Subscriptions zur Verfügung. Hierdurch wird nachgewiesen, dass das Konzept der Spezifikationsfiles korrekt umgesetzt wurde. Daraus entstehen viele Vorteile für den Benutzer, da er während der Erstellung der Launchfiles auf diese Informationen zurückgreifen kann und durch den *rxDeveloper* Hilfestellungen erhält.

5.3. Plattformunabhängigkeit und Verfügbarkeit

Aufgrund der Rückmeldungen aus der Community nach der Bekanntmachung des *rxDevelopers* wurden einige Plattform- und ROS-Kombinationstests nötig, die zu einer Verbesserung des Programms führten und mehr Anwendern die Benutzung ermöglichen. So Unterstützt der *rxDeveloper* sowohl die aktuelle ROS-Version „Electric Emit“ als auch die noch von vielen Benutzern erfragte Vorgängerversion „Diamondback“. Getestet wurden dazu die folgenden Betriebssysteme bzw. Distributionen:

- Ubuntu 10.04
- Ubuntu 10.10
- Ubuntu 11.04
- Ubuntu 11.10
- Mac OS X

Ubuntu ist die einzige, von ROS als unterstützt deklarierte, Linux-Distribution, weshalb andere Linux-Distributionen nicht getestet wurden. Grundsätzlich läuft der *rxDeveloper* jedoch auf jedem Betriebssystem, welches von Qt und ROS unterstützt wird und auf dem die Bibliotheken YAML-CPP und TINYXML kompilierbar sind. Sobald ROS vollständig unter Microsoft Windows lauffähig ist, spricht auch von Seiten des Programms nichts gegen eine Verwendung unter Windows. Eine Verbesserung Windows-Unterstützung ist jedoch erst für die nächste Versi-

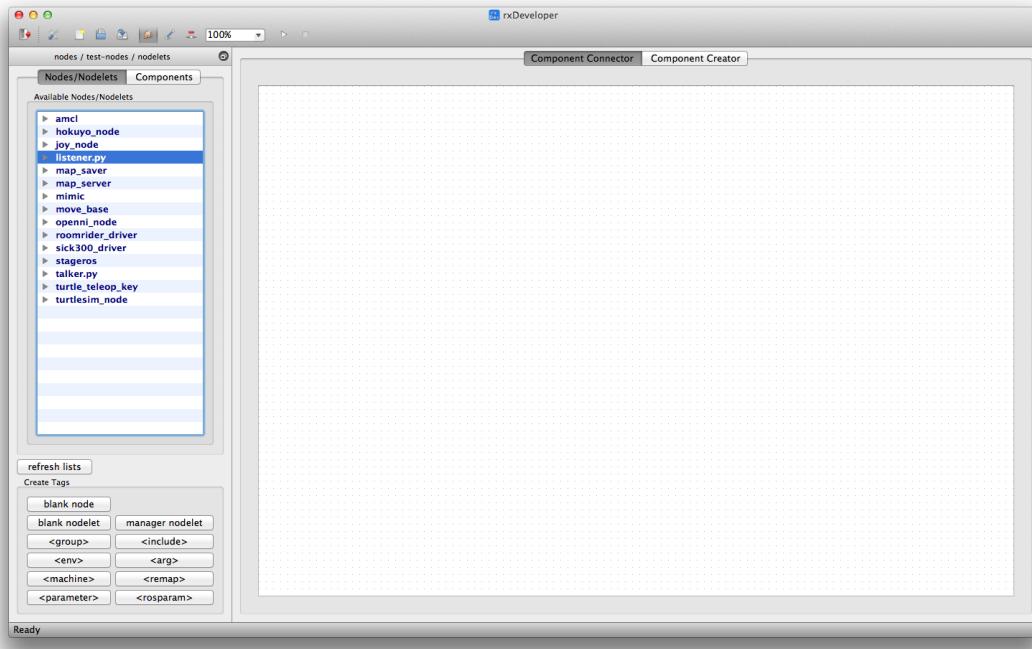


Abbildung 5.1.: rxDeveloper - Unter Mac OS X

on von ROS geplant. Abbildung 5.1 zeigt das Programm unter Mac OS X, die Abbildungen in Kapitel 4.3 wurden alle unter Linux erstellt. Alle getesteten Systeme präsentierte ein voll funktionstüchtiges Programm, was auch durch die Entwicklergemeinschaft in Kommentaren und Zuschriften bestätigt wurde.

5.4. Bewertung

Die Tests haben gezeigt, dass der *rxDeveloper* in der Lage ist alle gültigen Launchfiles anzusehen, diese zu erzeugen und abzuspeichern. Er erfüllt somit alle Kriterien der Vollständigkeit in Bezug auf Launchfiles, welche in der Anforderungsanalyse (siehe Kapitel 4.1.3) erhoben wurden. Weitere Kriterien, wie die Erstellung des *rxDevelopers* als eigenständiges und plattformunabhängiges Programm, wurden durch geeignete Bibliotheken und die vollständige Implementierung in Qt erfüllt. Das Programm ermöglicht die Erstellung von Launchfiles sowohl mit Hilfe von Spezifikationsdateien als auch ohne diese. Assistenzfunktionen bieten eine Reihe von Hilfestellungen bei der Erzeugung von Launchfiles und beschleunigen so den Ent-

5. Evaluation und Bewertung

wicklungsprozess. Es wurden Erstellungsmöglichkeiten für ROS-Pakete und Dateien implementiert. Auch Spezifikationsdateien können im Programm editiert oder neu erstellt werden und die Informationen über die Nodes werden dem Benutzer übersichtlich während der Entwicklung präsentiert. Das Ausführen von Launchfiles sowie das Aufrufen von Debugging-Tools wurde vereinfacht und so der Testprozess vereinfacht. Alle Anforderungen an das Programm wurden erfüllt und die Benutzbarkeit und Nützlichkeit nachgewiesen.

6 Kapitel 6

Zusammenfassung und Ausblick

In diesem Kapitel werden die Ergebnisse der Arbeit zusammengefasst. Des Weiteren soll ein Ausblick auf zukünftige Weiterentwicklungsmöglichkeiten für das erstellte Programm und seine Konzepte gegeben sowie andere Anwendungsfälle diskutiert werden.

6.1. Zusammenfassung

Das Ziel dieser Diplomarbeit war es, eine graphische Benutzeroberfläche für das Roboterframework ROS zu erstellen, die den Benutzer bei der Entwicklung von komponentenbasierten Softwaresystemen unterstützt. Als Basiskonzept fungierten die ROS-Launchfiles, die den Start von Komponenten, deren Parametrisierung und ihre Kommunikation festlegen und gemeinsam mit weiteren Konfigurationen zu einem Softwaresystem bündeln. Diese Dateien können durch das Programm gelesen, geschrieben und neu erstellt werden. Zu diesem Zweck wurde eine Umgebung erstellt, die zwischen visuell erstellten Diagrammen und der textuellen Sprache der Launchfiles übersetzt.

Um die Bedürfnisse und Arbeitsgewohnheiten der Anwender zu evaluieren, wurden Befragungen durchgeführt und regelmäßig Feedback aus der Community über Programmpräsentationen eingeholt. Aus den gewonnenen Erkenntnissen wurde ein detaillierter Anforderungskatalog zusammengestellt. Hiermit wurde sichergestellt, dass ein Produkt entsteht, welches durch seine Funktionalität und Betriebssystemunabhängigkeit eine große Benutzergruppe anspricht und ROS-Neulingen den Einstieg erleichtert. Ebenso wurde durch die Beteiligung der Community gewährleistet, dass das Programm leicht von interessierten Entwicklern weiterentwickelt werden kann, da ROS-Prinzipien für Software beachtet werden.

Mit Hilfe des Programms können Launchfiles für ROS-basierte Robotersysteme erstellt und bearbeitet werden. Das Hauptaugenmerk liegt hierbei auf der Interprozesskommunikation bereits vorhandener Komponenten und deren Parametrisie-

6. Zusammenfassung und Ausblick

rung. Es unterstützt jedoch sämtliche Möglichkeiten, die in Launchfiles enthalten sein können. Um die Komponenten innerhalb der Launchfiles leicht und typgerecht miteinander zu verbinden, wurde mit den sogenannten Spezifikationsdateien ein offenes Interface-Dateiformat entwickelt. Die Spezifikationsdateien ermöglichen die Hauptunterstützungsmöglichkeiten bei der Erstellung von Launchfiles und erleichtern das Erkennen und Verhindern einer fehlerhaften Verwendung von ROS-Komponenten. Neben Informationen zu den Kommunikationsschnittstellen dieser ROS-Ressourcen bieten die Spezifikationsdateien auch Informationen, die die Parametrisierung der Komponenten erleichtern. Das Konzept der Spezifikationsdateien wurde umfassend in den *rxDeveloper* integriert, so dass auch neue Spezifikationsdateien leicht erstellt und verwendet werden können. Trotz der Vorteile, welche die Dateien im Programm bieten, lässt sich der *rxDeveloper* auch ohne diese benutzen und behält seine Vollständigkeit im Bezug auf die Verwendung von Launchfiles.

Außer den Unterstützungsmöglichkeiten für die Komponenten, wurden noch viele weitere Assistenzfunktionen implementiert, die für die Erstellung von korrekten Launchfiles sorgen. Diese Assistenzfunktionen erkennen eine fehlerhafte Benutzung und weisen den Benutzer darauf hin. Eine lokale Ausführung des erstellten oder geladenen Launchfiles wird über die Programmoberfläche ebenso ermöglicht, wie das Starten von Debugging-Tools, die weiteren Aufschluss über die Funktionalität des Softwaresystems geben.

Der *rxDeveloper* wurde so entwickelt, dass er sich wie andere Programme in ROS starten lässt und sich gut in die ROS-Umgebung integriert. Er ist ein eigenständiges Programm mit nur wenigen externen Abhängigkeiten, die zudem noch automatisch durch ROS-Tools installiert werden können. Umfassende Tests wurden durchgeführt, um die Vollständigkeit und Benutzbarkeit des *rxDeveloper* zu belegen. So wurde die Verwendung aller Kombinationen von Anweisungen, die in einem Launchfile enthalten sein können, in verschiedenen Test erfolgreich überprüft. Auf diese Weise wurde nachgewiesen, dass die Inhalte von Launchfiles korrekt gelesen und geschrieben werden können und auch eine Neuerstellung für alle Anweisungskombinationen innerhalb des Programms möglich ist. Des Weiteren wurde das Programm unter verschiedenen Betriebssystemen erfolgreich ausgeführt und die betriebssystemunabhängige Verwendung nachgewiesen. Insgesamt wurden alle Anforderungen an das Programm erfüllt.

Das Feedback aus der Community lässt darauf schließen, dass der *rxDeveloper* bereits jetzt, kurz nach der Veröffentlichung, von einigen Benutzern verwendet wird und auf großes Interesse stößt. Viele Communitymitglieder bestätigten, dass der *rxDeveloper* eine sinnvolle Ergänzung zu den bereits vorhandenen graphischen

ROS-Tools darstellt und die Benutzbarkeit von ROS vor allem für Anfänger drastisch erhöht. Circa 40 Zuschriften über E-Mail und Kommentare in Foren und Webseiten gingen hierzu ein. Die, für die Veröffentlichung erstellte Projektseite wurde bereits über 500 Mal aufgerufen und das Tutorialvideo über 380 Mal betrachtet (Stand: 15.02.2012). Da das Programm per SVN über die Projektseite interessierten Benutzern zur Verfügung gestellt wird, kann über die genaue Downloadzahl keine Aussage gemacht werden.

6.2. Ausblick

Der *rxDeveloper* wurde entsprechend der ROS-Philosophie, nach der ein Programm genau eine Aufgabe erledigen soll, entwickelt. Dennoch ließe sich sein Aufgabenfeld leicht vergrößern. Ein Beispiel hierfür wären Erweiterungen um Möglichkeiten zur Entwicklung von Quelltext und somit für die Erstellung von neuen Komponenten. Dabei muss kein vollständiger Quellcode-Editor implementiert werden. Vorstellbar wäre zum Beispiel ein Template-System basierend auf den Spezifikationsdateien. Die Spezifikationen der Schnittstellen innerhalb der Dateien könnten genutzt werden, um Quelltext automatisch zu generieren. In diesem Fall müsste der Anwender lediglich die Informationen zu den Schnittstellen definieren und könnte anschließend daraus automatisch Node-Quelltext generieren, ähnlich wie es bei der CORBA-IDL (Object Management Group, 2011) möglich ist.

Ein Weiteres Tool für die Spezifikationsdateien könnte ein ROS-Wiki Importer/Exporter sein. Mit einem solchen Tool könnten Spezifikationsdateien aus Wiki-Seiten erstellt werden, wenn diese die benötigten Informationen für Komponenten enthalten. Ebenso wäre der umgekehrte Weg möglich, bei dem automatisch Wiki-Seiten mit den Daten aus den Spezifikationsdateien aktualisiert werden. Einen ersten Ansatz dafür stellt bereits der in Kapitel 4.3.8 vorgestellte Spezifikationsdatei-Editor dar, mit dem optional Wiki-Quelltext generiert werden kann.

Auch die Spezifikationsdateien selbst bieten Raum für Erweiterungen. Für den Start eines Nodes benötigte andere Nodes könnten in den Spezifikationsdateien als Abhängigkeiten angegeben werden und so dem *rxDeveloper* noch weitere unterstützende Fähigkeiten geben. Mit diesen Informationen könnten im Programm Warnungen ausgegeben werden, falls ein kreierter Launchfile nicht alle abhängigen Komponenten enthält. Außerdem können andere Programme erzeugt werden, die Informationen aus Spezifikationsdateien benötigen und um eventuelle eigene Informationen erweitern.

6. Zusammenfassung und Ausblick

Die erstellte Software bietet bereits ein Konzept an, um in Richtung Komponentenkomposition in ROS einen Schritt weiter zu gehen. Die in Kapitel 4.3.3 gezeigte Funktionalität zum direkten und schnellen Verwenden häufig benutzter Launchfiles in Form eines Includes kann sehr gut dafür genutzt werden, um aus atomaren Komponenten neue Komponenten zusammenzusetzen und diese dem Benutzer zur Wiederverwendung zur Verfügung zustellen. Hersteller von Development Kits oder von Roboterhardware allgemein könnten diese Komponentendateien für ihre Hardware mitliefern und so dem Entwickler mit einer Basiskonfiguration viel Einarbeitungszeit abnehmen. Mit den Komponentendateien könnten die verschiedenen Treiber eines Development Kits oder Robotersystems gestartet und parametrisiert werden, damit einzelnen Hardwarekomponenten miteinander Kommunizieren können. Der Benutzer müsste nur noch die angedachte Funktionalität für den Roboter implementieren und das Softwaresystem aus der Komponentendatei hinzufügen.

7

Kapitel 7

Verzeichnisse

7.1. Definitionen und verwendete Abkürzungen

Definitionen

Berechnungsgraph Ein laufendes Softwaresystem basierend auf ROS wird als Berechnungsgraph bezeichnet, dessen Knoten die Softwarekomponenten sind und dessen Kanten die Kommunikationswege implizieren.

Community Mit Community oder ROS-Community ist stets der internationale Personenkreis gemeint, der ROS benutzt und eventuell eigene ROS-Module entwickelt.

Dynamic reconfigure Nodes Nodes, die zur Laufzeit partiell konfiguriert werden können.

Key/Value-Paar Zuordnung eines Wertes (Value) zu einem Schlüsselbegriff (Key).

Mapping Ist hier im Bezug auf YAML-Syntax zu verstehen und beschreibt die Zuordnung eines Key/Value-Paares.

Node Mit Node werden die ROS-Prozesse bezeichnet, die einen Knoten im Berechnungsgraphen darstellen. Nodes sind Softwarekomponenten oder auch Tools oder Treiber.

Nodelets Ermöglicht die Ausführung von mehreren Algorithmen in einem Prozess, ohne dass Daten zwischen den Algorithmen mit Hilfe der CPU kopiert werden müssen (engl.: „zero copy transport“).

Launchfile Launchfiles beschreiben textuell die Struktur des ROS-Berechnungsgraphen und beinhalten neben den Softwarekomponenten unter

anderem Parameter und Details über die Startumgebung.

Package Als Package wird in ROS eine Ordnerstruktur bezeichnet, die Ressourcen, wie Nodes, Quelltext, Messagedefinitionen, Daten, etc., enthält.

Peer-to-Peer-Modell Bezeichnet die Kommunikation zwischen gleichberechtigten Kommunikationspartnern. Bei ROS agieren Prozesse häufig nach diesem Modell.

Publisher Der Publisher ist ein Node, der Daten auf ein Topic schreibt und so veröffentlicht.

Remapping Um nicht nur auf Ressourcen innerhalb oder oberhalb des eigenen Namespaces zugreifen zu können, existiert eine Möglichkeit, Namen zur Laufzeit umzubenennen.

rxDeveloper Der rxDeveloper ist das im Zuge dieser Arbeit entstandene Programm.

Schnittstellendefinitionssprache Eine deklarative formale Sprache, die eine Sprachsyntax zur Beschreibung von Schnittstellen einer Softwarekomponente bietet. Neben den Schnittstellen, einschließlich der zugehörigen Datentypen, können hiermit auch Parameter und vorhandene Methoden von Objekten definiert werden.

Sequenz Ist hier im Bezug auf YAML-Syntax zu verstehen und beschreibt eine Liste von Gruppen aus Key/Value-Paaren.

Spezifikationsdatei Spezifikationsdateien beschreiben, basierend auf einer IDL, die Eigenschaften einer Softwarekomponenten. Dazu gehören mögliche Parameter, asynchrone und synchrone Kommunikationsmöglichkeiten, sowie die Packetzugehörigkeit eines ROS-Nodes.

Subscriber Der Subscriber bezeichnet einen Node, der auf ein, von einem anderen Node publiziertes, Topic zugreifen möchte.

Topic Um den Inhalt der Messages zu identifizieren und die Kommunikation zu organisieren werden die Messages durch ein Transportsystem, den Topics, geleitet.

Abkürzungen

API	Application Programming Interface
CLI	Command-line Interface
GUI	Graphical User Interface
IDE	Integrated Development Environment
IDL	Interface Definition Language
IPC	Inter-process Communication
OOP	Object-oriented Programming
ROS	Robot Operating System
UML	Unified Modeling Language
XML	Extensible Markup Language

XML-RPC Extensible Markup Language Remote Procedure Call

7.2. Abbildungen

- | | | |
|------|--|---|
| 1.1. | Anstieg der Roboter-Population 2006-2008. (Quelle: http://spectrum.ieee.org , Artikel: „World Robot Population Reaches 8.6 Million“ [Stand: 07.02.2012]) | 2 |
| 1.2. | Hierarchische Roboterprogrammierstruktur, nach Han <i>et al.</i> (2010). Die jeweils untere Ebene bildet die Grundlage für die darüber liegende Ebene. | 3 |
| 1.3. | Grundfunktionen des geplanten Programms. In der <i>graphische Darstellung</i> (blauer Rahmen) werden Elementen aus Launchfiles in Form eines Graphen mit Knoten und Kanten dargestellt, um den ROS-Berechnungsgraphen zu simulieren. | 7 |

7. Verzeichnisse

2.1. (a) Flowchart-Darstellung eines einfachen Algorithmus: Gehe solange vorwärts, bis du 20 Schritte getan hast!	14
(b) Datenflussdiagramm eines einfachen Algorithmus: 1) Wende einen Farbfilter auf das aktuelle Bild der Videodaten an! 2) Drehe das Bild! 3) Gib es auf dem Bildschirm aus!	14
2.2. State Machine-Darstellung eines einfachen Algorithmus: „1) Bewege dich vorwärts! 2.a) Wenn du links eine Wand siehst, drehe dich 90 Grad nach rechts oder 2.b) Wenn du rechts eine Wand siehst, drehe dich 90 Grad nach links! Fahre fort mit Punkt 1)!“	15
2.3. UML-Diagramm einer Komponentenverknüpfung über Interfaces. <i>Komponente 1</i> bietet ein Interface an, welches <i>Komponente 2</i> anfordert.	18
2.4. Synchrone (oben) und asynchrone (unten) Kommunikation in ROS, Quelle: http://www.ros.org/wiki/ROS/Concepts [Stand: 07.02.2012]	29
3.1. RobotFlow - ”22/04/2002 : Test network with mapping algorithm”, Quelle: http://robotflow.sourceforge.net/ [Stand: 07.02.2012]	43
3.2. RCX-Code, Quelle: Biggs und Macdonald (2003)	45
3.3. Robolab, Quelle: http://www.nxt-in-der-schule.de/lego-mindstorms-education-nxt-system/nxt-software/robolab-2.9 [Stand: 07.02.2012]	46
3.4. NXT-G, Quelle: http://www.generationrobots.de/roboerprogrammierung-mit-nxt-g-und-microsoft-vpl,fr,8,42.cfm [Stand: 07.02.2012]	47
3.5. Nao Choregraphe, Quelle: http://www.aldebaran-robotics.com/en/Discover-NAO/Software/choregraphe.html [Stand: 07.02.2012]	47
3.6. Urbi - Gostai Studio, Quelle: http://www.gostai.com/products/jazz/gostai_studio/ [Stand: 07.02.2012]	48
3.7. OPRoS - Component Authoring Tool, Quelle: Jang <i>et al.</i> (2010, S.651)	49
3.8. OPRoS - Component Composer, Quelle: Jang <i>et al.</i> (2010, S.651) .	50
3.9. rxgraph, Quelle: http://www.ros.org/wiki/rxgraph [Stand: 07.02.2012]	51
4.1. Prototypen der graphischen Darstellung im Programm zum Zeitpunkt der Umfrage	56
4.2. Schematische Darstellung der Elemente bei einem Remap. <i>Node 1</i> publiziert ein Topic, welches von <i>Node 2</i> abgerufen wird.	69

4.3. rxDeveloper - Component Connector: 1. Arbeitsfläche; 2. Seitenleiste; 3. mögliche Launchfile-Elemente; 4. verfügbare Komponenten; 5. Tab für abgespeicherte zusammengesetzte Komponenten	71
4.4. rxDeveloper - Seitenleiste: (a) Node-Tab - 1. Tag-Buttons; 2. Browser für vorhandene Nodes; (b) Component-Tab	72
4.5. rxDeveloper - Toolbar und Menü: 1. Drag&Drop-Modus; 2. Remap-Modus; 3. Delete-Modus; 4. Gruppe (neu, öffnen, speichern) für Launchfiles; 5. Start/Stop; 6. Zoom Arbeitsfläche	73
4.6. rxDeveloper - Informationsleiste: 1. Informationsleiste; 2. missglückter Topic-Remap; 3. geglückter Topic-Remap	73
4.7. rxDeveloper - ROS-Prozesse: a) Standard-Node erstellt aus einer Spezifikationsdatei; b) Test-Node erstellt aus einer Spezifikationsdatei; c) Nodelet erstellt aus einem Template; d) Nodelet-Manager erstellt aus einem Template; e) Standard-Node erstellt aus einem Template mit Angabe eines Namespaces; g) Group-Element mit Namespace welches b) und d) beinhaltet; f) Darstellung eines Topic-Remaps von einem Publish-Topic in a) auf ein Subscribe-Topic in b) und zugeordnet in a) (impliziert durch rotes "incl")	74
4.8. rxDeveloper - weitere Launchfile-Elemente: a) verschiedene Parameter-Tags, b) Machine-Tag, c) verschiedene Arg-Tags, d) unterschiedliche Rosparam-Tags, e) Include-Tag, f) allgemeiner Remap-Tag (im Gegensatz zum expliziten Topic-Remap zwischen Nodes, siehe Abb. 4.7), g) Env-Tag	75
4.9. rxDeveloper - verschachtelte Elementen: a) Group-Element, welches b) und c) beinhaltet. Elemente in einem Group-Bereich, werden mit kleinen Klammern dargestellt; b) Node-Element im Group-Element, der jeweils mindestens einen Remap-, einen Param- und einen Rosparam-Tag enthält (impliziert durch "contains: [element-type]"); c) Param-Tag im Group-Element.	77
4.10. rxDeveloper - Node-Editor: 1. Elementattribute; 2. Verwaltung der untergeordneten Elemente; a. neues untergeordnetes Element hinzufügen; b. untergeordnetes Element löschen	78
4.11. rxDeveloper - Parameter-Assistent	79
4.12. rxDeveloper - Remap-Topic-Editor	80
4.13. rxDeveloper - Include-Editor: 1. Öffnen der angegebenen Datei; 2. Extrahieren der Informationen aus der Datei	81
4.14. rxDeveloper - Component Creator: 1. Package-Browser; 2. Anlegen neuer Dateien; 3. Anlegen eines neuen ROS-Paketes; 4. Seitenleiste mit vorhandenen Paketen zur Auswahl	82

7. Verzeichnisse

4.15. rxDeveloper - Specfile-Editor	83
5.1. rxDeveloper - Unter Mac OS X	91

7.3. Tabellen

2.1. Inhalt eines ROS-Package und zugehörige Standard-Ordner	26
2.2. Grundtypen für Nachrichten in ROS und ihre Entsprechungen in C++ und Python (nach Quelle: http://ros.org/wiki/msg [Stand: 07.02.2012])	28
2.3. Arrays aus Grundnachrichtentypen in ROS und ihre Entsprechungen in C++ und Python (nach Quelle: http://ros.org/wiki/msg [Stand: 07.02.2012])	28
2.4. Mögliche Positionen von Tags in Launchfiles (Spalten entsprechen möglichen übergeordneten, Zeilen eventuell untergeordneten Tags) .	38
4.1. Mögliche Schlüsselwörter einer Spezifikationsdatei	64
5.1. Launchfile-Tags + eventuelle Attribute mit Wert (falls distinkтив), gruppiert nach dem Tag-Typ.	88

7.4. Listings

2.1. Remapping	30
2.2. Launchfile-Tag Struktur	32
2.3. Verschachtelung von Tags	38
2.4. Node starten	39
2.5. Namespaces Includes und Remaps	39
2.6. Parameter	39
2.7. Lauffähiges Group- und Remapping-Beispiel	40
4.1. Nodespezifikationsdatei - Konzept zum Zeitpunkt der Umfrage . .	55
4.2. Nodespezifikationsdatei - Schema	63
4.3. Nodespezifikationsdatei - Beispiel	64
4.4. Nodespezifikationsdatei - Dateipfadkonvention	66
4.5. Programmstart per rosrun	68
4.6. Komponentendateien - Dateipfadkonvention	70

7.5. Literaturverzeichnis

- [Beder 2012] BEDER, J.: *yaml-cpp*. Stand: 07. Februar 2012. – URL <http://code.google.com/p/yaml-cpp/w/list>
- [Ben-Kiki *et al.* 2009] BEN-KIKI, Oren ; EVANS, Clark ; NET, Ingy döt: *YAML Ain't Markup Language (YAMLTM) Version 1.2*. Oktober 2009. – URL <http://yaml.org/spec/1.2/spec.pdf>
- [Biggs und Macdonald 2003] BIGGS, Geoffrey ; MACDONALD, Bruce: A Survey of Robot Programming Systems. In: *in Proceedings of the Australasian Conference on Robotics and Automation, CSIRO*, 2003, S. 27
- [Bischoff *et al.* 2002] BISCHOFF, Rainer ; KAZI, Arif ; SEYFARTH, Markus: The MORPHA style guide for icon-based programming. In: *Proc. 11th IEEE Int Robot and Human Interactive Communication Workshop*, 2002, S. 482–487
- [Brooks *et al.* 2005] BROOKS, A. ; KAUPP, T. ; MAKARENKO, A. ; WILLIAMS, S. ; OREBACK, A.: Towards component-based robotics. In: *Proc. IEEE/RSJ Int. Conf. Intelligent Robots and Systems (IROS 2005)*, 2005, S. 163–168
- [Brugali und Scandurra 2009] BRUGALI, D. ; SCANDURRA, P.: Component-based robotic engineering (Part I) [Tutorial]. In: *IEEE Robotics & Automation Magazine* 16 (2009), Nr. 4, S. 84–96
- [Brugali und Shakhimardanov 2010] BRUGALI, D. ; SHAKHIMARDANOV, A.: Component-Based Robotic Engineering (Part II). In: *IEEE Robotics & Automation Magazine* 17 (2010), Nr. 1, S. 100–112
- [Ceriani und Migliavacca 2010] CERIANI, Simone ; MIGLIAVACCA, Martino: Middleware in robotics. 2010. – Forschungsbericht
- [Cote *et al.* 2004] COTE, C. ; LETOURNEAU, D. ; MICHAUD, F. ; VALIN, J.-M. ; BROSSEAU, Y. ; RAIEVSKY, C. ; LEMAY, M. ; TRAN, V.: Code reusability tools for programming mobile robots. In: *Proc. IEEE/RSJ Int. Conf. Intelligent Robots and Systems (IROS 2004)* Bd. 2, 2004, S. 1820–1825
- [Cote *et al.* 2006] COTE, Carle ; BROSSEAU, Yannick ; LÉTOURNEAU, Dominic ; RAÏEVSKY, Clément ; MICHAUD, François: Robotic Software Integration Using MARIE. In: *International Journal of Advanced Robotic Systems* 3 (2006), 3,

7. Literaturverzeichnis

Nr. 1, S. 055–060

[Cousins 2010] COUSINS, Steve: Welcome to ROS Topics [ROS Topics]. In: *Robotics Automation Magazine, IEEE* 17 (2010), march, Nr. 1, S. 13 –14. – ISSN 1070-9932

[Gerkey und Conley 2011] GERKEY, Brian ; CONLEY, Ken: Robot Developer Kits. In: *Robotics & Automation Magazine, IEEE* 18 (2011), Nr. 3

[Gerkey *et al.* 2003] GERKEY, Brian P. ; VAUGHAN, Richard T. ; HOWARD, Andrew: The Player/Stage Project: Tools for Multi-Robot and Distributed Sensor Systems. In: *In Proceedings of the 11th International Conference on Advanced Robotics*, 2003, S. 317–323

[Gill und Smart 2002] GILL, Christopher D. ; SMART, William D.: Middleware for robots. In: *Distributed and Embedded Systems Papers from the* (2002), S. 1–5

[Gostai 2012] GOSTAI: *Gostai Studio*. Stand: 07. Februar 2012. – URL <http://www.gostai.com/urbistudio.html>

[Gumbley und MacDonald 2005] GUMBLEY, Luke ; MACDONALD, Bruce A.: Development of an Integrated Robotic Programming Environment. In: *Australasian Conference on Robotics and Automation 2005* (2005)

[Han *et al.* 2010] HAN, Soohee ; KIM, Mi sook ; PARK, Hong S.: Components and an effective IDE of Open software Platform for Robotics Services. In: *Proc. Int Control Automation and Systems (ICCAS) Conf*, 2010, S. 393–398

[Henning 2008] HENNING, Michi: The rise and fall of CORBA. In: *Commun. ACM* 51 (2008), August, S. 52–57. – ISSN 0001-0782

[Jang *et al.* 2010] JANG, Choulsoo ; LEE, Seung-Ik ; JUNG, Seung-Woog ; SONG, Byoungyoul ; KIM, Rockwon ; KIM, Sunghoon ; LEE, Cheol-Hoon: OPRoS: A New Component-Based Robot Software Platform. In: *ETRI Journal* vol.32, no.5 (2010), S. pp.646–656

[Kim und Jeon 2008] KIM, Seung H. ; JEON, Jae W.: Using visual programming kit and LEGO Mindstorms: An introduction to embedded system. In: *Proc. IEEE Int. Conf. Industrial Technology ICIT 2008*, 2008, S. 1–6

[Kramer und Scheutz 2007] KRAMER, James ; SCHEUTZ, Matthias: Development

environments for autonomous mobile robots: A survey. In: *Autonomous Robots* 22 (2007), S. 132

[Macdonald *et al.* 2003] MACDONALD, Bruce ; YUEN, David ; WONG, Sylvia ; WOO, Evan ; GRONLUND, Rowan ; COLLETT, Toby ; TREPANIER, Felix-Etienne ; TRÉPANIER, Félix étienne ; BIGGS, Geoff: Robot Programming Environments. In: *in ENZCon2003 10th Electronics New Zealand Conference, (University of*, 2003

[Makarenko *et al.* 2006] MAKARENKO, Alexei ; BROOKS, Alex ; KAUPP, Tobias: Orca: Components for robotics. In: *Conference on Intelligent Robots* (2006), S. 163–168

[MathWorks 2012] MATHWORKS: *Simulink - Simulation und Model-Based Design*. Stand: 07. Februar 2012. – URL <http://www.mathworks.de/products/simulink/index.html>

[Mohamed *et al.* 2008] MOHAMED, N. ; AL-JAROODY, J. ; JAWHAR, I.: Middleware for Robotics: A Survey. In: *Proc. IEEE Conf. Robotics, Automation and Mechatronics*, 2008, S. 736–742

[Namoshe *et al.* 2008] NAMOSHE, Molaetsa. ; TLALE, N. S. ; KUMILE, C. M. ; BRIGHT, G.: Open middleware for robotics. In: *Proc. 15th Int. Conf. Mechatronics and Machine Vision in Practice M2VIP 2008*, 2008, S. 189–194

[NI 2012] NI: *National Instruments LabVIEW*. Stand: 07. Februar 2012. – URL <http://www.ni.com/labview/d/>

[Nokia 2012] NOKIA: *Qt*. Stand: 07. Februar 2012. – URL <http://qt.nokia.com/>

[Object Management Group 2011] OBJECT MANAGEMENT GROUP, Inc. (.: *Common Object Request Broker Architecture (CORBA) Specification Version 3.2*, November 2011. – URL <http://www.omg.org/spec/CORBA/3.2/Interfaces/PDF>

[Park und Han 2009] PARK, Hong-Seong ; HAN, Soohee: Development of an Open Software Platform for Robotics Services. In: *Proc. ICCAS-SICE*, 2009, S. 4773–4775

[Plasil und Stal 1998] PLASIL, Frantisek ; STAL, Michael: *An Architectural View*

7. Literaturverzeichnis

- of Distributed Objects and Components in CORBA, Java RMI, and COM/DCOM.* 1998
- [Quigley *et al.* 2009] QUIGLEY, Morgan ; CONLEY, Ken ; GERKEY, Brian P. ; FAUST, Josh ; FOOTE, Tully ; LEIBS, Jeremy ; WHEELER, Rob ; NG, Andrew Y.: ROS: an open-source Robot Operating System. In: *ICRA Workshop on Open Source Software*, 2009
- [Roy *et al.* 1998] ROY, G. G. ; KELSO, J. ; STANDING, C.: Towards a visual programming environment for software development. In: *Proc. Int Software Engineering: Education & Practice Conf*, 1998, S. 381–388
- [Shakhimardanov und Prassler 2007] SHAKHIMARDANOV, A. ; PRASSLER, E.: Comparative evaluation of robotic software integration systems: A case study. In: *Proc. IEEE/RSJ Int. Conf. Intelligent Robots and Systems IROS 2007*, 2007, S. 3031–3037
- [Shakhimardanov *et al.* 2011] SHAKHIMARDANOV, Azamat ; HOCHGESCHWENDER, Nico ; RECKHAUS, Michael ; KRAETZSCHMAR, Gerhard K.: Analysis of software connectors in robotics. In: *Proc. IEEE/RSJ Int Intelligent Robots and Systems (IROS) Conf*, 2011, S. 1030–1035
- [Siciliano und Khatib 2008] SICILIANO, Bruno ; KHATIB, Oussama ; SICILIANO, Bruno (Hrsg.) ; KHATIB, Oussama (Hrsg.): *Handbook of Robotics*. Springer, 2008
- [Thomason 2012] THOMASON, Lee: *TinyXML*. Stand: 07. Februar 2012. – URL <http://sourceforge.net/projects/tinyxml/>
- [Zwintzscher 2004] ZWINTZSCHER, Olaf: *Software-Komponenten im Überblick: Einführung, Klassifizierung und Vergleich von JavaBeans, .Net, EJBs, Corba, UML 2.* 1. W3l, 12 2004

A

Anhang A

Anhang

A.1. Umfragedetails

rxDeveloper - online survey

Total records in this survey	23
Completed	23

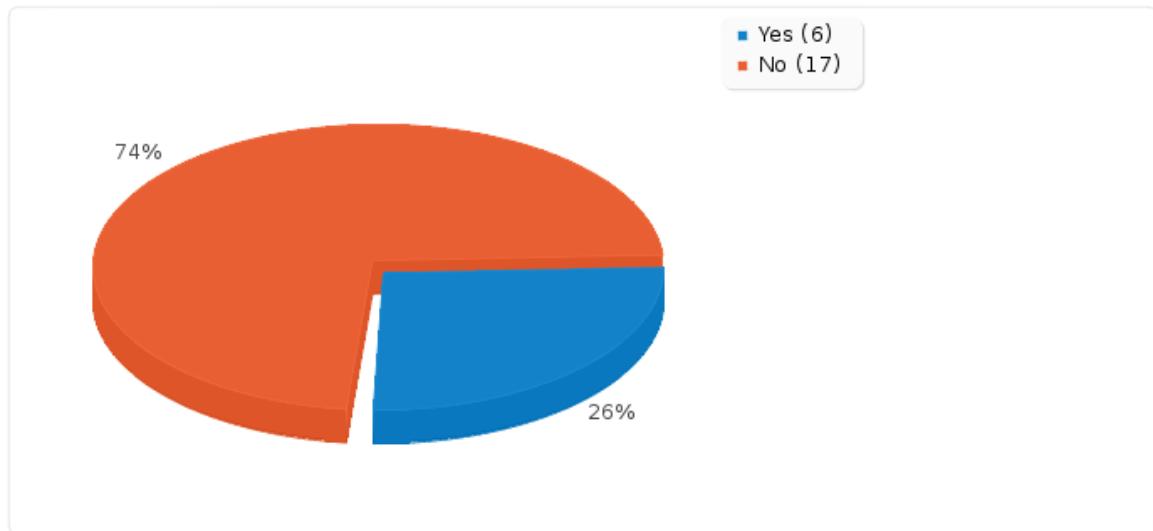
Fragengruppe 1: „The node specfiles“

Field summary for 1

Should dependencies be part of the spec-files?

(e.g., a node is always launched together with another node)

Answer	Count	Percentage
Yes (Y)	6	26.09%
No (N)	17	73.91%

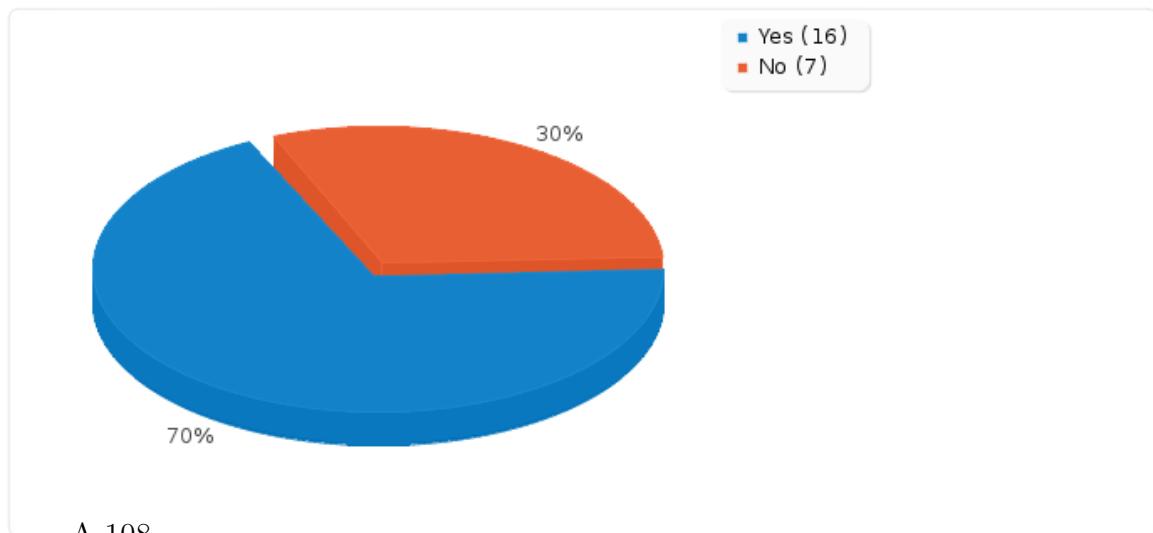


Field summary for 2

Would you support me by creating specfiles for some nodes?

(This survey is anonymous)

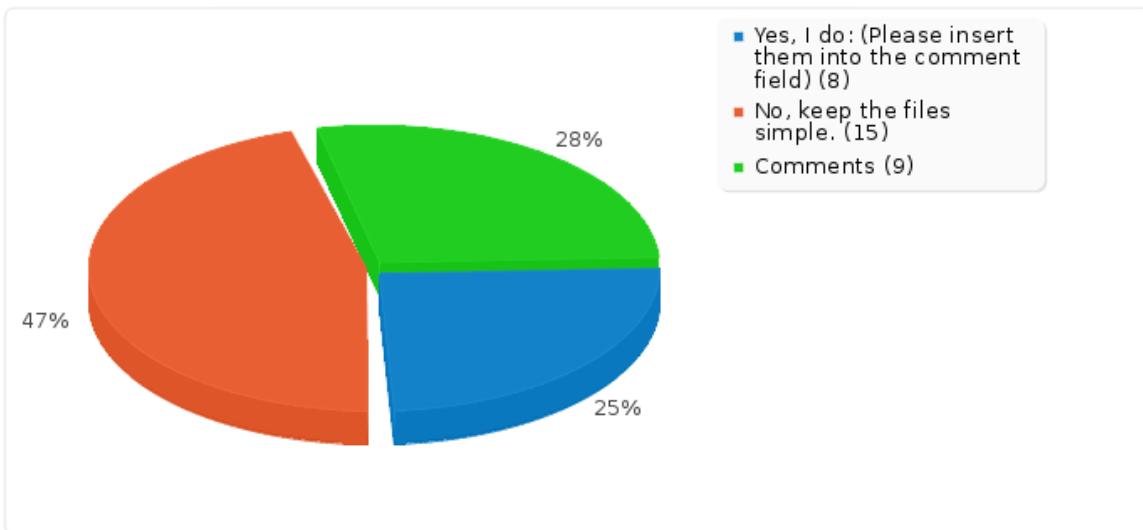
Answer	Count	Percentage
Yes (Y)	16	69.57%
No (N)	7	30.43%
No answer	0	0.00%



Field summary for 3

Can you think of other information that should be part of spec-files

Answer	Count	Percentage
Yes, I do: (Please insert them into the comment field) (A1)	8	34.78%
No, keep the files simple. (A2)	15	65.22%
Comments	9	39.13%
No answer	0	0.00%



Kommentare zur Fragengruppe 1:

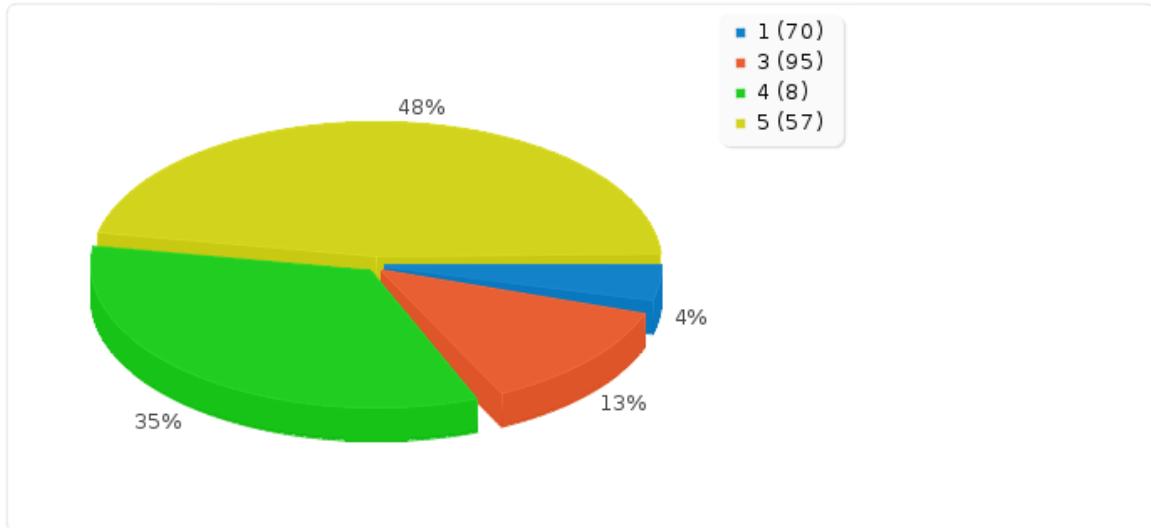
1. allow for actions such that you could map result msg to a goal msg from action to action
2. actionlib actions (both client and server); dynamic_reconfigure parameters?
3. At least, the parser should not throw an error when additional fields are specified in the yaml file. This allows users to add arbitrary information they might need for a specific application.
4. This wasn't mentioned up to this point, but I'd like to stress that it would be extremely useful to have spec files autogenerated (It's not clear to me if this is intended, so I thought I'd mention it)
5. But make them optional!
6. Node versions or message versions may be useful.
7. - services both provided and called - TF frames broadcast and used (actually, not 100% sure this is needed/useful but give it some thought)
8. Should topic hints be in there anywhere? I am thinking of things like image_transport compression type. Maybe it isn't important.
9. "rosnode info" is a possible source of information. The only thing that you'll have to know is whether you can start the node in a ros "sandbox" without destroying hardware (this should not happen, even for drivers).

Fragengruppe 2: „Features“

Field summary for 1(SQ001)

Please select the importance for the different features. (1 = lowest; 5 = highest)
[edit node parameters]

Answer	Count	Percentage
1 (1)	1	4.35%
2 (2)	0	0.00%
3 (3)	3	13.04%
4 (4)	8	34.78%
5 (5)	11	47.83%
Sum (Answers)	23	100.00%
Number of cases	23	100.00%
No answer	0	0.00%
Arithmetic mean	4,22	
Standard deviation	1	

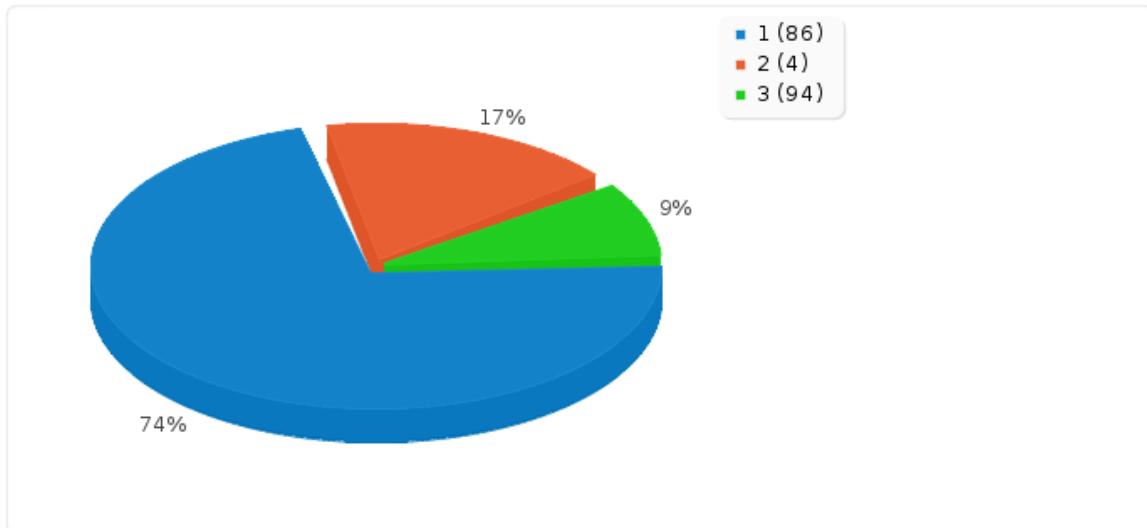


A.1. Umfrage details

Field summary for 1(SQ002)

Please select the importance for the different features. (1 = lowest; 5 = highest)
[edit node sourcecode internally]

Answer	Count	Percentage
1 (1)	17	73.91%
2 (2)	4	17.39%
3 (3)	2	8.70%
4 (4)	0	0.00%
5 (5)	0	0.00%
Sum (Answers)	23	100.00%
Number of cases	23	100.00%
No answer	0	0.00%
Arithmetic mean	1,35	
Standard deviation	0,65	

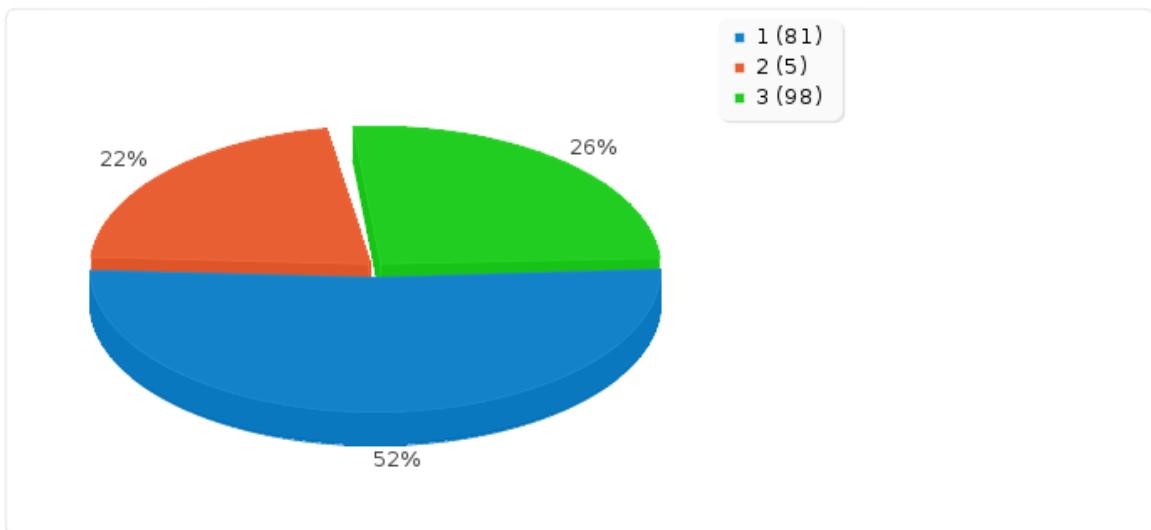


Anhang A. Anhang

Field summary for 1(SQ003)

Please select the importance for the different features. (1 = lowest; 5 = highest)
[edit node sourcecode with chosen editor]

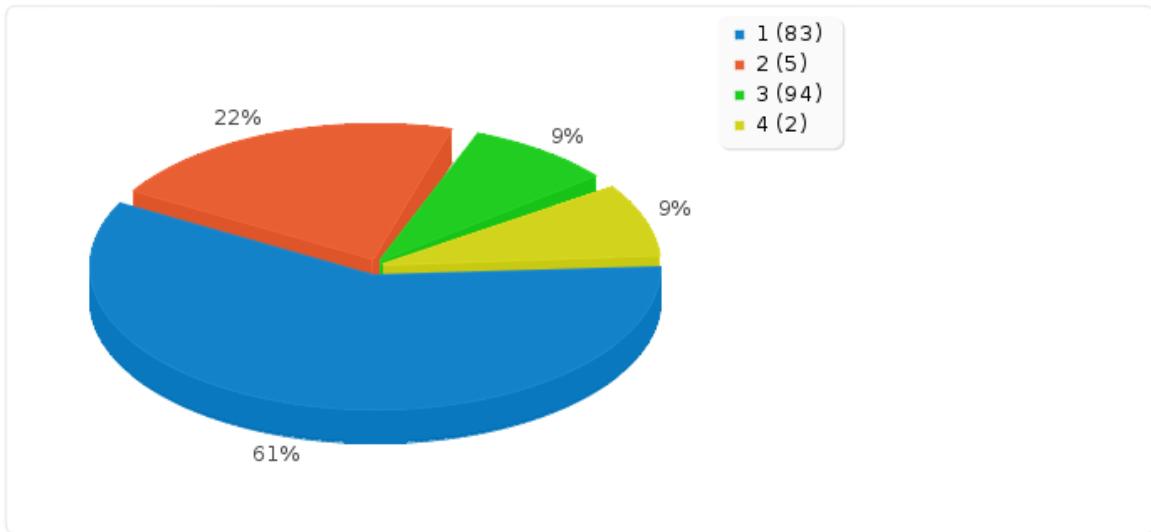
Answer	Count	Percentage
1 (1)	12	52.17%
2 (2)	5	21.74%
3 (3)	6	26.09%
4 (4)	0	0.00%
5 (5)	0	0.00%
Sum (Answers)	23	100.00%
Number of cases	23	100.00%
No answer	0	0.00%
Arithmetic mean	1,74	
Standard deviation	0,86	



Field summary for 1(SQ004)

Please select the importance for the different features. (1 = lowest; 5 = highest)
[create new packages]

Answer	Count	Percentage
1 (1)	14	60.87%
2 (2)	5	21.74%
3 (3)	2	8.70%
4 (4)	2	8.70%
5 (5)	0	0.00%
Sum (Answers)	23	100.00%
Number of cases	23	100.00%
No answer	0	0.00%
Arithmetic mean	1,65	
Standard deviation	0,98	

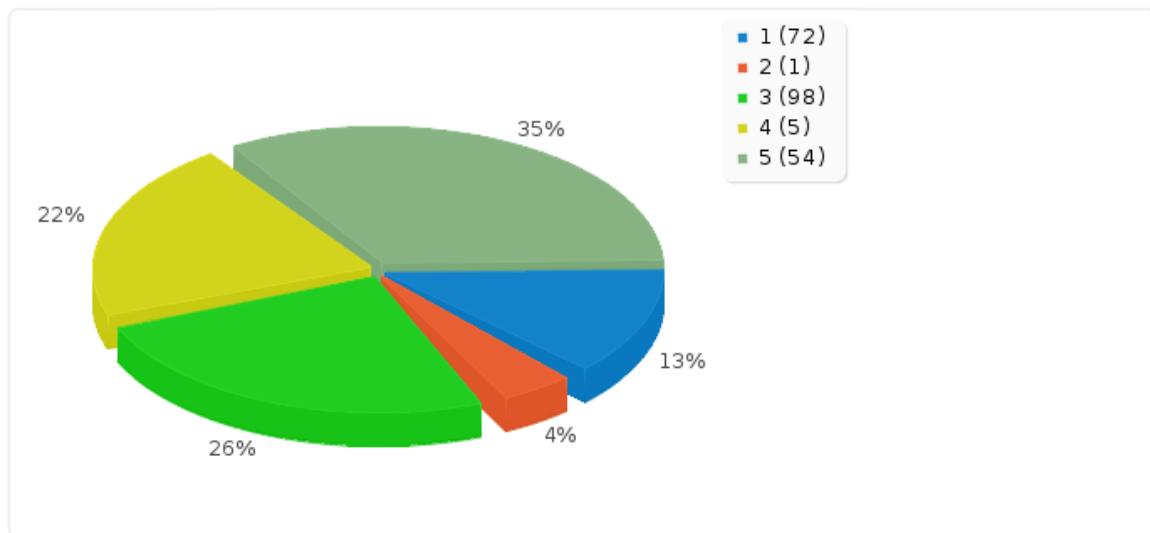


Anhang A. Anhang

Field summary for 1(SQ005)

Please select the importance for the different features. (1 = lowest; 5 = highest)
[edit specfiles]

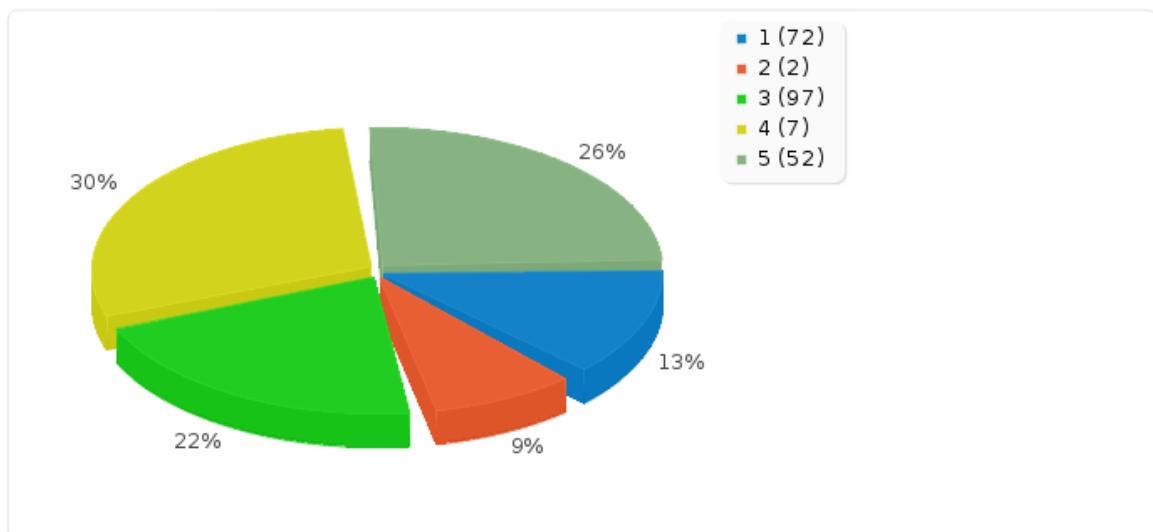
Answer	Count	Percentage
1 (1)	3	13.04%
2 (2)	1	4.35%
3 (3)	6	26.09%
4 (4)	5	21.74%
5 (5)	8	34.78%
Sum (Answers)	23	100.00%
Number of cases	23	100.00%
No answer	0	0.00%
Arithmetic mean	3,61	
Standard deviation	1,37	



Field summary for 1(SQ006)

Please select the importance for the different features. (1 = lowest; 5 = highest)
 [create new specfiles]

Answer	Count	Percentage
1 (1)	3	13.04%
2 (2)	2	8.70%
3 (3)	5	21.74%
4 (4)	7	30.43%
5 (5)	6	26.09%
Sum (Answers)	23	100.00%
Number of cases	23	100.00%
No answer	0	0.00%
Arithmetic mean	3,48	
Standard deviation	1,34	

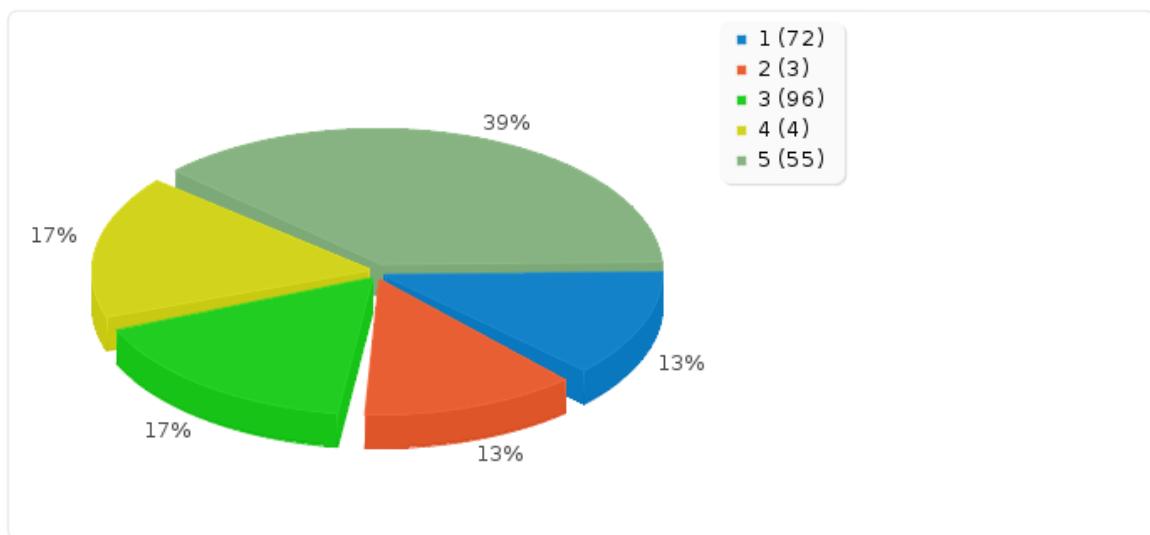


Anhang A. Anhang

Field summary for 1(SQ007)

Please select the importance for the different features. (1 = lowest; 5 = highest)
[import specfile information from launchfiles]

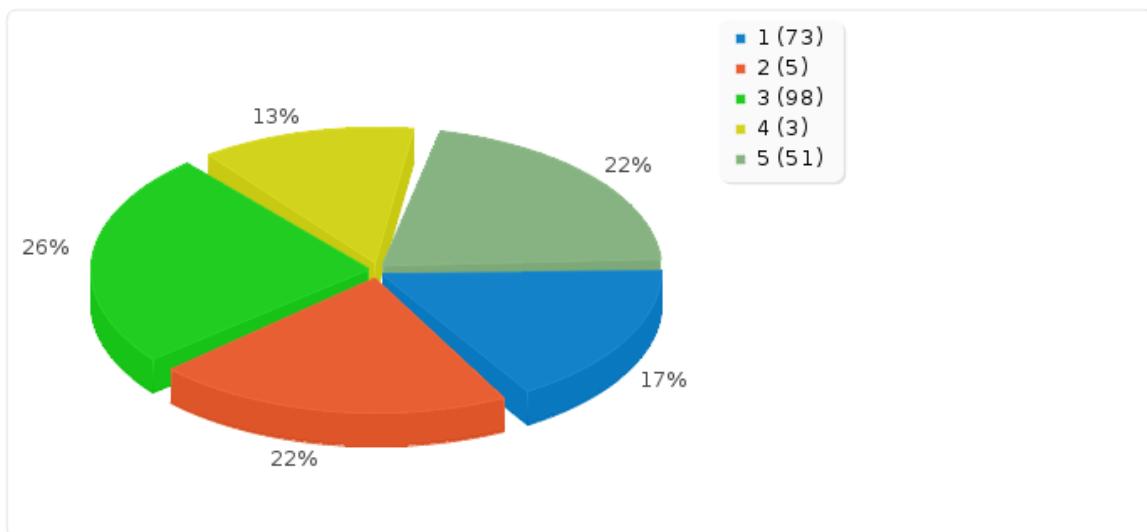
Answer	Count	Percentage
1 (1)	3	13.04%
2 (2)	3	13.04%
3 (3)	4	17.39%
4 (4)	4	17.39%
5 (5)	9	39.13%
Sum (Answers)	23	100.00%
Number of cases	23	100.00%
No answer	0	0.00%
Arithmetic mean	3,57	
Standard deviation	1,47	



Field summary for 1(SQ008)

Please select the importance for the different features. (1 = lowest; 5 = highest)
 [check for required dependencies while creating new launchfiles]

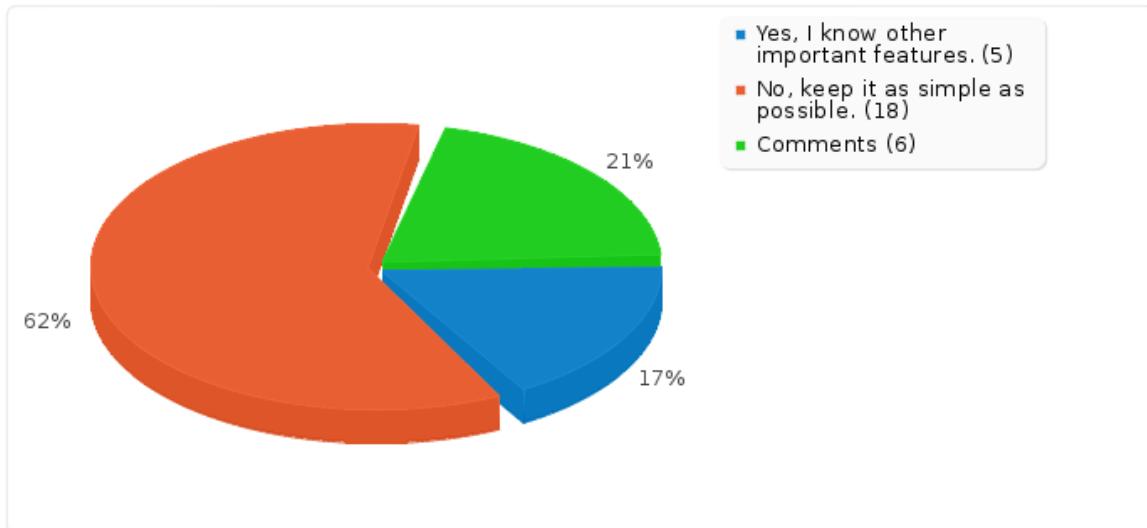
Answer	Count	Percentage
1 (1)	4	17.39%
2 (2)	5	21.74%
3 (3)	6	26.09%
4 (4)	3	13.04%
5 (5)	5	21.74%
Sum (Answers)	23	100.00%
Number of cases	23	100.00%
No answer	0	0.00%
Arithmetic mean	3	
Standard deviation	1,41	



Field summary for 2

Other important features?

Answer	Count	Percentage
Yes, I know other important features. (A1)	5	21.74%
No, keep it as simple as possible. (A2)	18	78.26%
Comments	6	26.09%
No answer	0	0.00%



Kommentare zu Fragengruppe 2:

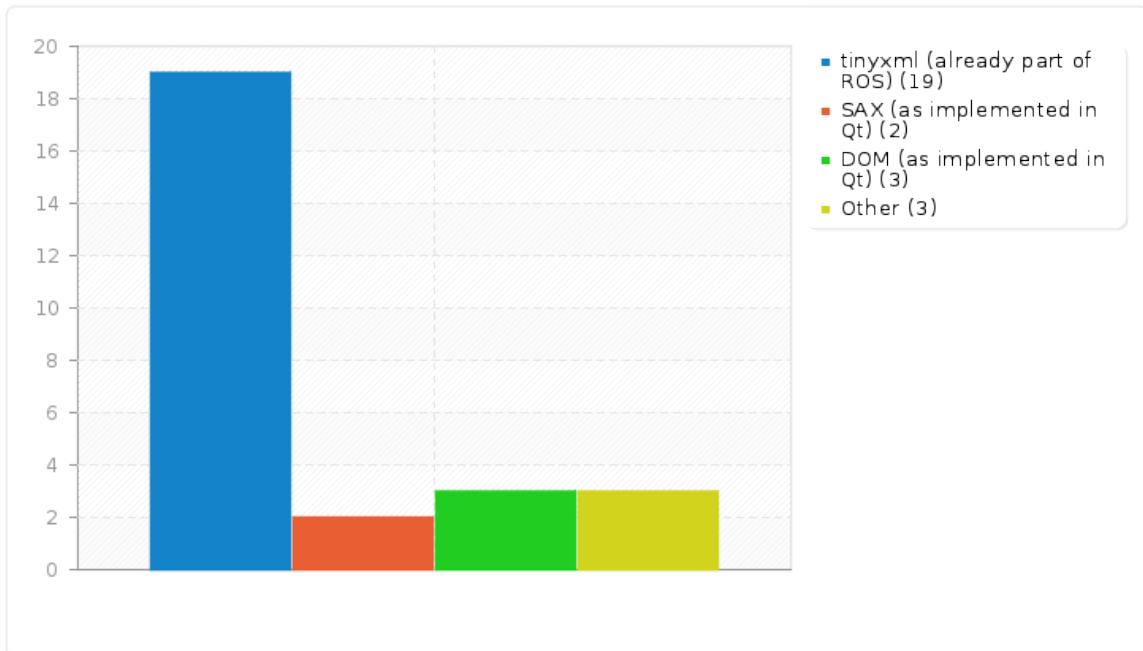
1. to create on the fly node blocks similar to matlab to labview allow for smach containers (iterators, concurrencies, etc) type matching on the connection construction so you can't connect topics with type mismatches
2. * Edit remappings * Edit additional params such as launch-prefix * Edit machines * Manage includes correctly * Detect duplicate nodes, i.e. duplicate includes and give exact error location and maybe a parse stack * Detect errors in yaml syntax inside <rosparam> tags and give correct error location
3. Consideration of Nodelets (e.g. similar features for them)
4. autogenerate spec files from source
5. It would fit in the ROS/Unix mentality to keep the tool as simple as possible.
6. allow parametrization as can be done with the \$(arg) and \$(env) substitutions in launch files

Fragengruppe 3: „Libraries“

Field summary for 1

Which library should be used for launchfile parsing and writing (XML)?

tinyxml (already part of ROS)	19	82.61%
SAX (as implemented in Qt)	2	8.70%
DOM (as implemented in Qt)	3	13.04%
Other	3	13.04%

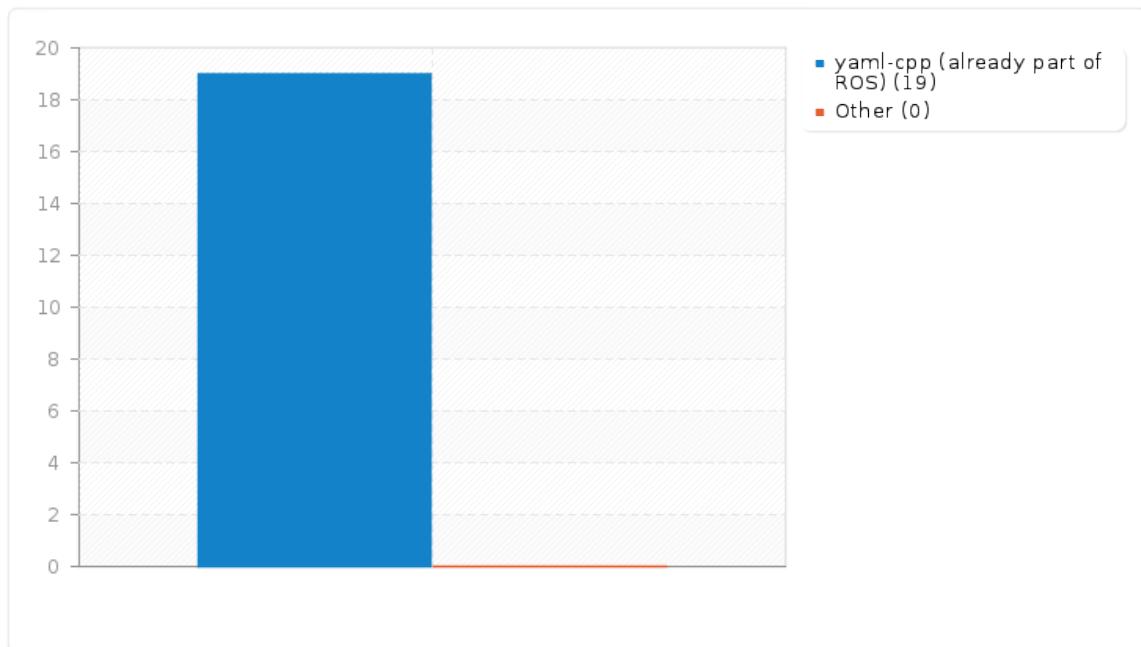


Anhang A. Anhang

Field summary for 2

Which library should be used for spec-file and parameterfile parsing (YAML)?

yaml-cpp (already part of ROS)	19 82.61%
Other	0 0.00%

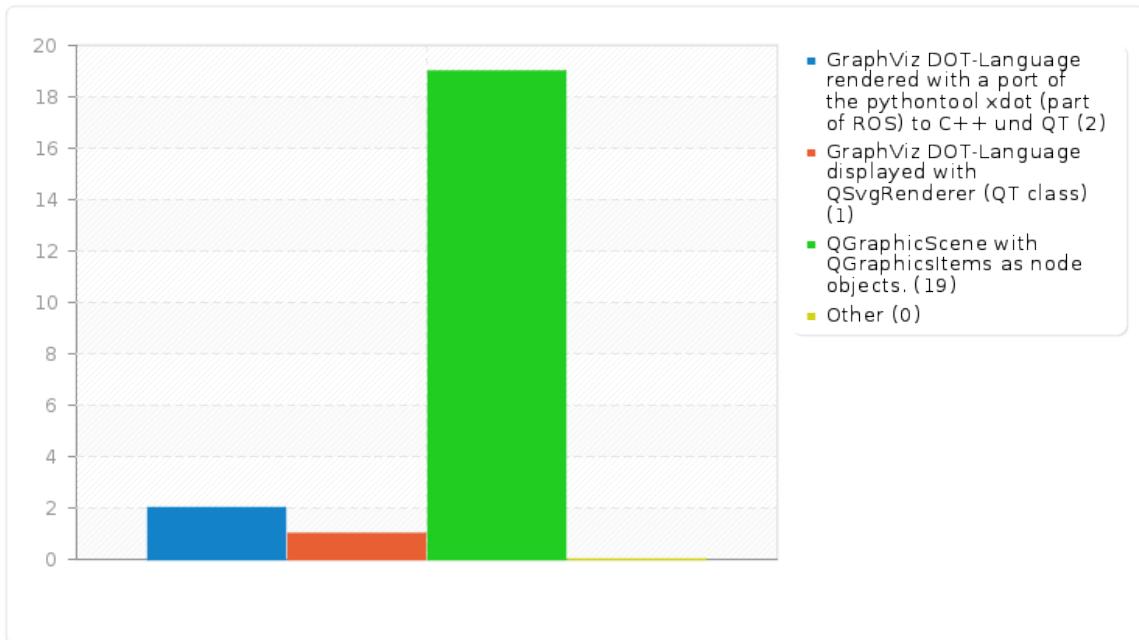


Field summary for 3

How should the graphical representation of the nodes be implemented? My first Idea was to make the tool similar to rxgraph and internally use the DOT-format for that purpose. This way the rendering would always be the same. A QGraphicsScene seems more flexible and the graphic items are objects.

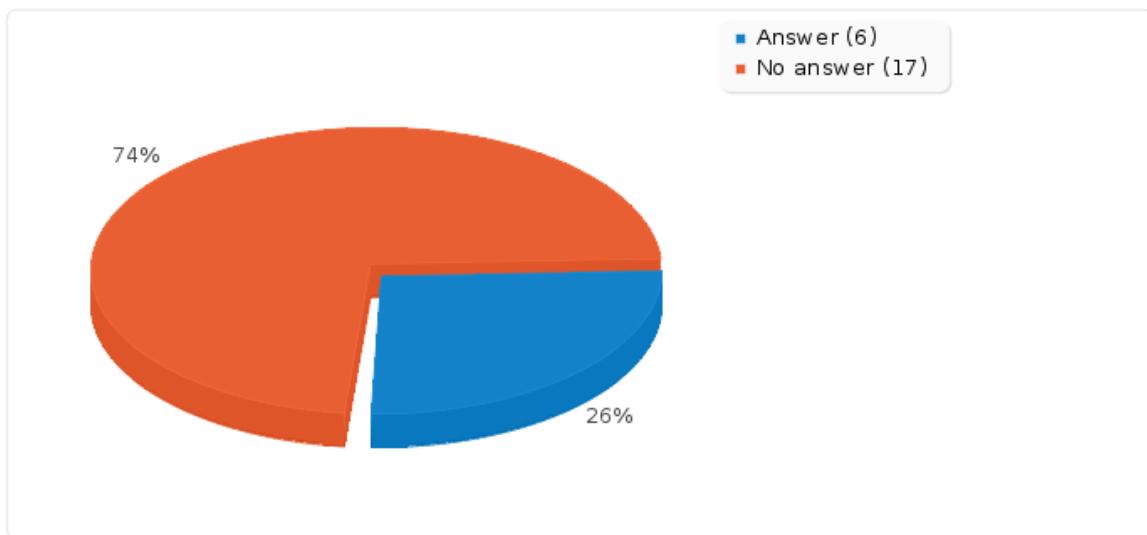
GraphViz DOT-Language rendered with a port

of the pythontool xdot (part of ROS) to C++ und QT	2	8.70%
GraphViz DOT-Language displayed with		
QSvgRenderer (QT class)	1	4.35%
QGraphicScene with QGraphicsItems		
as node objects.	19	82.61%
Other	0	0.00%



Field summary for 4
Other ideas or remarks?

Answer	6 26.09%
No answer	17 73.91%



Kommentare zu Fragengruppe 3:

1. no GraphViz please
2. please not graphviz it is better if the connections between nodes is explicit similar to labview or matlab where you can see that the output of a block is specifically connected to an input of another block
3. Take whatever visualization suits you. It would probably be useful to connect nodes by drag & drop.

Allgemeine Kommentare:

1. It would be nice for the spec files to be parsed by the wiki, to generate a ROS interface section.
2. Keep it simple and "do one thing well"! I think a graphical launchfile editor would be great for visualizing which node in an elaborate filter chain connects to which, also with remapped topics. One example of that can be found here: "pr2_arm_navigation/pr2_arm_navigation_perception/launch/laser+stereo-perception.launch".
3. Could the spec file maybe be replaced by an extension to the manifest file? I consider adding another file to the standard package layout problematic. As far as I know, additional tags in manifest files don't lead to errors, so maybe integrating the spec file there would work.
4. Great idea, the specfiles would actually make writing nodes easier too--you could have a library that instantiates subscribers and publishers based on the specfile. However, you should focus on the core functionality first. Better to have fewer things working really well than lots of things buggy. Good luck!

A.2. Projektseite, Quellcode und Testfiles

Webseite

Die Projektseite des *rxDevelopers* befindet sich unter:

<http://code.google.com/p/rxdeveloper-ros-pkg/>

Hier findet sich Anleitungen und auch ein Videotutorial. Des Weiteren wird hier eine Plattform für Fehlermeldungen und das Programm selbst zur Verfügung gestellt. Über diese Webseite wird die Weiterentwicklung des *rxDevelopers* erfolgen.

Quellcode des Programms

Der Quelltext des Programms zum Zeitpunkt der Abgabe befinden sich unter folgender Adresse oder kann auf der Projektseite per SVN bezogen werden:

<http://rxdeveloper-ros-pkg.googlecode.com/files/Quelltext.tar.gz>
SHA1 Checksum: d86990eb76f11cd64be3a7977b4a2cda11c0c2b8

Testfiles

Die Testfiles aus Kapitel 5.1 befinden sich unter:

<http://rxdeveloper-ros-pkg.googlecode.com/files/Testfiles.tar.gz>
SHA1 Checksum: c09945637de6a6bfe7eec6dca64bc370abfd7b44

Anhang A. Anhang

