



Technische Universität München





Distributed Multimodal Information Processing
Group

Prof. Dr. Matthias Kranz

Diplomarbeit

Towards End-User Programming of complex Robotic
Systems and Intelligent Environments

Programmierung von komplexen Robotersystemen und
intelligenten Umgebungen durch End-Benutzer

Author:	Christian Schraml
Matriculation Number:	
	
	
	
Advisor:	Prof. Matthias Kranz
Begin:	17.01.2011
End:	17.07.2011

The thesis investigates the possibility to introduce End-User Programming to complex robotic systems and Intelligent Environments.

ROS, the Robot Operating System¹, is selected as middleware for distributed sensor-actuator systems that focuses on expert-users. To reach more users and make it attractive to end-users, the ROS-VP (visual programming) toolkit is developed to enable the creation of a ROS program employing a visual programming language.

The toolkit's concept is based on the integration and automatic analysis of existing ROS packages, easy to use configuration and connection of software modules as well as the provision of two packages that contain operations for strings and numbers. Visually created ROS programs can be exported, stored and executed without further conversion.

The prototype is implemented into Eclipse to keep a familiar user interface and simplify the integration into the existing development process. Two evaluated use cases visualize the capabilities of the application and show that it is suited for both novice and expert-users.

¹For details: <http://www.ros.org>, accessed June 30th 2011

Diese Diplomarbeit untersucht die Möglichkeit, End-Benutzern die Programmierung von komplexen Robotersystemen oder intelligenten Umgebungen zu ermöglichen.

ROS, das Robot Operating System², wird als Middleware für verteilte Sensor-Aktor-Systeme eingesetzt, welches auf erfahrene Entwickler ausgerichtet ist. Um mehr Nutzer zu erreichen und es insbesondere für End-Benutzer attraktiver zu machen wird das ROS-VP (visuelle Programmierung) Toolkit entwickelt welches Nutzern erlaubt ROS grafisch zu programmieren.

Das Konzept des Toolkits integriert und analysiert existierende ROS Pakete und erlaubt einfache Konfiguration und Verbindung von Software-Modulen. Zusätzlich werden zwei ROS Pakete, welche die grundlegende Verarbeitung von Zahlen und Strings erlaubt, zur Verfügung gestellt. Alle mit dem ROS-VP Toolkit erstellten ROS Programme können exportiert werden und sind ohne weitere Umwandlung direkt ausführbar.

Der Prototype integriert das Konzept in Eclipse um eine bekannte Benutzeroberfläche zu verwenden und es sich leichter in existierende Entwicklungsprozesse einbinden lässt. Zwei evaluierte Anwendungsfälle zeigen, dass der Prototype sowohl Anfängern als auch Spezialisten einen Mehrwert bieten kann.

²Für details: <http://www.ros.org>, Zugriff am 30. Juni 2011

Contents

Contents	4
1 Introduction	7
1.1 Motivation	8
1.2 Objective	9
1.3 Structure	10
2 Fundamentals and related work	11
2.1 End-User Programming	11
2.1.1 Comparing End-User and Professional Programming	11
2.1.2 Types of End-User Programming languages	12
2.1.3 Visual programming	13
2.2 Ubiquitous computing	15
2.2.1 Intelligent Environments	16
2.2.2 Robot Operating System (ROS)	17
3 Requirements analysis	20
3.1 Use cases	20
3.1.1 Explanations	20
3.1.2 Number processing for end-users (Stephanie)	22
3.1.3 Extension of ROS programs (Alexander)	23
3.1.4 Rapid prototyping for expert-users (Florian)	23
3.1.5 Support for custom packages (Martina)	24
3.2 User needs	24
3.2.1 Toolkit for user innovation	24
3.2.2 Integration into existing development process	25
3.3 Requirements	27
3.3.1 Functional requirements	28
3.3.2 Non-functional requirements	29
4 State of the art	30
4.1 Imperative programming	30
4.1.1 LEGO Mindstorms NXT	30
4.2 Dataflow programming	32
4.2.1 Microsoft Robotics Developer Studio	32

4.2.2	Simulink	33
4.3	Event-driven programming	34
4.3.1	App Inventor	34
4.3.2	StarLogo TNG	35
4.3.3	Scratch	36
4.4	Results	38
5	Concept for End-User Programming in ROS	41
5.1	User interface	41
5.1.1	Integration into Eclipse	42
5.1.2	Visual programming language	43
5.2	Edit mode	44
5.2.1	Integration of ROS packages	44
5.2.2	Connection and configuration of nodes	46
5.2.3	Integration of ROS launch files	47
5.3	Active mode	48
5.4	Provision of basic nodes	48
5.4.1	Numbers package	49
5.4.2	Strings package	50
6	Prototype	52
6.1	Software architecture	52
6.1.1	Information exchange of user interface components	52
6.1.2	Graphical object architecture)	54
6.1.3	Internal ROS representation	57
6.2	Implementation	58
6.2.1	Configuration wizard	58
6.2.2	Visual programming editor	60
6.2.3	Action bar and context menu	60
6.2.4	Views of the ROS-VP toolkit	62
6.2.5	Package for numbers and strings	63
6.3	Evaluation of two use-cases	64
6.3.1	Simple math program	64
6.3.2	Image processing program	66
7	Conclusion	68
7.1	Discussion	69
7.2	Future work	70
	List of Figures	72
	List of Tables	74
	A Source code	76

CONTENTS

6

Bibliography

77

Chapter 1

Introduction

Nowadays, mobile devices offer a variety of applications and sensors that are used to support the user in his¹ daily tasks, like accessing data from any place as long as an Internet connection is present [4]. In spite of the amount of far more than half a million applications for mobile devices [17], programming environments are developed to allow end-users create their own applications. As an example, the App Inventor for Android allows novice programmers to create their own applications and use them on their phone [54]. Two essential trends can be found in this development: End-User Programming and ubiquitous computing.

End-User Programming describes the possibility that enables non-professional programmers to create or modify a software artifact [34]. According to a study by Scaffidi et al. [46], the number of end-user programmers is at least four times higher than the number of professional ones. Applications that integrate End-User Programming have the advantage of letting the user extend and customize existing functionalities according to his preferences [16]. As an example, Von Hippel [59] mentions spreadsheets, that allow users to create their custom models as they know the most about their specific needs. Without End-User Programming, the creation would require the involvement of developers. If the user's creation can be shared in an active community, the whole application or system may be improved [58].

Ubiquitous computing has according to Weiser [63] the purpose to augment the environment with computational resources. Users are provided with services as well as information when and where they are desired. This development is a consequence, from the fact that processors, connectivity, memory as well as battery and displays are becoming better, cheaper and more versatile [32]. To support the user in his daily tasks, these devices must be interconnected and be able to exchange information between each other [64], as a connection to the Internet would allow. This trend is supported by Cloud Computing, which offers Software and Hardware as a Service, as Salesforce.com [31] or Amazon EC2 showed [28]. Cloud Computing allows access to data and services at any time as long as a connection to the Internet is present and scaled at the user's demand [18].

¹The male pronoun represents male and female users alike

One subset of ubiquitous computing are Intelligent Environments. These are highly embedded, interactive spaces that bring computers in the real world [12]. One exemplary scenario for Intelligent Environments is a smart home that supports its residents pro-actively and context-aware during the activities of daily living [44]. The idea is, that home appliances, from radiators over dishwashers up to simple lightbulbs, are interconnected with each other over standards like ZigBee, Ethernet or WLAN [56]. Combined with Cloud Computing services, Intelligent Environments can inform users about their energy consumption² or their predicted travel time based on the traffic situation [44]. Roalter et al. [44] showed that, in the area of Intelligent Environments, no standardized middleware is available and that the whole "market" is fragmented. They proposed the usage of ROS, the Robot Operating System, as a middleware for Intelligent Environments. In contrast to the fact that ROS is originally programmed for robots [41], its capabilities to connect different sensors, actuators or devices via a peer-to-peer network and enable information exchange between software modules make ROS suitable for Intelligent Environments [44]. Consequently, I use the middleware in this thesis and investigate the possibility to introduce End-User Programming to ROS.

1.1 Motivation

ROS focuses currently on expert-users, like developers of robotic systems. It has a steep learning curve even for average developers as its documentation and tutorials are still limited. As ROS is open-source software, a community emerged that actively contributes and shares over 3000 different packages³. These packages contain software modules, so-called nodes, that can be used to create a ROS program. ROS has as of now no consistent naming scheme, standardized internal structure or requirements for a proper documentation for community-contributed modules. In order to use a new node, the user has to look into the source code and documentation to understand the node. Especially to connect different nodes to each other and to configure it, knowledge about the node is required.

Eisenberg [16] argues that enabling end-users to program and modify a system on their own is preferable as the developers are not able to foresee all requirements and customize every feature for the user. In addition, Von Hippel [58] showed that user communities can play an important role to improve a product. The integration of end-users let the community grow and they might develop for new scenarios and share their experience to increase the functionalities of an application [59]. Von Hippel [58] argued that an active community is an important success factor as the example of the Apache Web server showed. Developed as an open-source project it was able to get, in spite of competitors like Microsoft, more than half of the market.

²For details: <http://www.greenpocket.de/produkte/greenpocket-losung/>, accessed July 16th 2011

³For details: <http://www.ros.org/browse/list.php>, accessed July 11th 2011

The integration of end-users in ROS programming is at the moment difficult, because tasks like the creation of ROS programs or new nodes require experience. Yet, current trends in information technologies make it necessary and desirable to integrate end-users. One way would be to implement tools that support the user and reduce the complexity. ROS offers a variety of tools that can be used at run-time of a ROS program. These tools allow for example to investigate if the software modules are properly linked to each other and what messages are transmitted. For the creation of new modules, there are also tools like compiling software packages. All ROS tools are accessed via the command-line and require some basic knowledge. The configuration and creation of a ROS program is currently not supported by any tools, like semi-automatic generation of launch files. The user has to check the source code on his own and create the file in a text editor. In my opinion, this currently limits the possibilities for rapid-prototyping of new ROS programs and in particular sets a high barrier to entry for new users.

Reducing the complexity to create launch files and thereby making ROS more accessible might attract new users to the community. I develop an application in this thesis to simplify this creation process for end-users as well as for expert-users. As the user interface has to support the different levels of experience I use visual programming. Visual programming is one type of End-User Programming that allows users to build their program by placing connecting blocks [16]. According to Myers [40], visual programming has advantages in comparison to traditional programming languages as for example textual based ones. The information representation feels more natural and allows the user to model complex systems with concurrent tasks without using a complicated syntax. Especially in the area of robotic and control systems, visual programming is already present in applications like LEGO Mindstorm NXT or Microsoft Robotics Developer Studio (see Chapter 4 for a comparison of programming approaches).

ROS is considered as a suitable middleware for Intelligent Environments which focuses on developers. To open ROS towards End-User Programming, the creation process of a new ROS program is currently seen as major issue. I argue, that this problem can be solved by an application that uses visual programming to support the user in the creation of a ROS program.

1.2 Objective

The objective of this thesis is to simplify the creation and modification of ROS programs. Therefore, the ROS-VP (visual programming) toolkit is conceptualized and implemented. It enables End-User Programming during the creation process of a new ROS program.

This approach should make ROS more attractive to end-users, as they are able to create their own program without being a professional developer. The developed toolkit reuses and automatically analyzes existing ROS packages and does not require the creation of new ones. Users can recombine these modules to build customized ROS programs that fit their needs. The ROS-VP toolkit applies visual programming to allow the user to focus on the creation of a system and sticks to existing ROS artifacts like launch files. This allows the seamless integration of the toolkit into existing development processes. In addition, expert-users can apply the toolkit to rapid prototype new ROS programs.

The scope of this toolkit is to support the user from the point he is aware about his specific needs and which software modules he requires up to the point where he exports his program and has a launch file. The ROS-VP toolkit does not try to simplify the creation of new software modules. In addition, as the run-time support is already available using various ROS tools, this is therefore not focus of this thesis but used wherever appropriate. As the creation process of these launch files is very complex, the ROS-VP toolkit is not able to cover every feasible option. Its implementation supports the features that I see as critical to support end-users and allow the creation of most programs.

1.3 Structure

The following chapter provides a common understanding in the research topics End-User Programming as well as ubiquitous computing. Therefore, both terms are defined and related research is presented. The third chapter defines four use cases for the ROS-VP toolkit, followed by general user needs. Then, precise requirements for the concept are derived. A summary on the current state of the art of application development with a visual programming language is presented.

The concept for the ROS-VP toolkit is presented in chapter five. It is based on visual programming, two different operation modes, namely edit and active mode, and the provision of basic operations for strings and numbers. The implemented concept is described in chapter six. The focus lies on the technology that is used as well as the software engineering concepts. At the end of this chapter, the prototype is evaluated.

This thesis concludes with a summary of the ROS-VP toolkit and its contribution to End-User Programming for ubiquitous computing. The concept and its implementation are critically discussed and possible future steps to improve the developed toolkit are presented.

Chapter 2

Fundamentals and related work

The ROS-VP toolkit combines concepts of End-User Programming and ubiquitous computing. To ensure a common knowledge, the key principles of the the two research topics are explained in the following. Accordingly, End-User Programming is defined and compared to professional programming, followed by the different types of End-User programming. After a definition of ubiquitous computing, Intelligent Environments are described and exemplified. The chapter is concluded with an explanation of the Robot Operating System .

2.1 End-User Programming

End-User Programming (EUP), also called End-User Development, is defined by Lieberman et al. [34] as follows:

”End-User Development can be defined as a set of methods, techniques, and tools that allow users of software systems, who are acting as non-professional software developers, at some point to create, modify or extend a software artifact.”

Consequently, End-User Programming includes for example all users who program or record macros in spreadsheets, build a CAD model (e.g. 3D model of cars) or a website with web authoring tools [16]. In the following, End-User Programming is compared to professional programming and advantages of EUP are explained. The different types to implement EUP are outlined in the following, concluded with a detailed description of visual programming.

2.1.1 Comparing End-User and Professional Programming

A way to distinguish between end-user and professional programmers is that end-user programmers are people, who do not write programs as their primary function [16]. They only use it to support their primary goal, like creating a financial model or an informative

website. Myers et al. [39] argue that professionals write or maintain programs as their primary job function. To be more precise, professional programming languages can only be used and understood, if the user is part of the programming community or had significant training. EUP languages can be conceived by end-users without specific experience with this language. End-user projects are usually of a very limited scope, implemented by the user supporting his own goal [16]. Professional programmers usually work in a team to collaboratively reach their targets .

To comprehend how important End-User Programming is, a study [46] from 2008 in the USA showed that over 50 million people use tools like spreadsheets and databases, 12 million call themselves programmers, but only 3 million people are indeed professional programmers. This means that the number of end-user programmers is between 12 and 50 million. The study also showed that this group is expected to increase. Accordingly, the integration of End-User Development will become more important and a selling argument for future products. Another reason besides the sheer mass of users for the implementation of an EUP language is that users are enabled to add his own functionalities [16]. This means that applications have more features that are not directly implemented by the developers. If these new modules can be shared and exchanged, users can use and be inspired by each others concepts and improve existing software [59].

2.1.2 Types of End-User Programming languages

As the definition of End-User Programming is very broad, there are various concepts to implement a suitable programming language. To classify those, Eisenberg [16] suggests the following three categories:

Implicit or Hidden languages: In this category, the user is not expected to have any contact with the programming language, neither is able to see it. One of its typical applications are macros, which are recorded when the user performs the task on his own [21]. Typical examples, which uses this method, are spreadsheet applications, where users can store regular tasks and execute them by clicking a simple button. That way, the end-user has no exposure to any kind of textual or visual languages. On the contrary, it is challenging to implement complex programs that are not sequential as these programs require access to the generated code [21]. Therefore, hidden languages are suited for simple tasks or programs.

Domain- or interface-specific languages: More complex programs, like the creation of mathematical models [2], require domain- or interface-specific languages. These are variations of existing professional programming languages, modified to fit the requirements of a specific application [16], like allowing the user to edit only a single function and adding custom functions with a sounding naming scheme. MatLab for example combines standard C-Code with custom functions to access mathematical

functions and create programs [2].

Visual or iconic languages: A very broad definition of visual programming (VP) is that it allows the user to "specify a program in a two-(or more)-dimensional fashion" [40]. A more practical approach is that a visual programming language allows a user to build his program with graphical elements that can be recombined [16]. Therefore, a collection of elements or blocks that represent a specific function, are offered. These are associated to each other like using a directed graphs or by sticking them together. This way, the user can construct his program using provided elements. As an example, Simulink (see Section 4.2.2 for details) has a visual programming editor that allows the user to design mathematical models [2].

These three language types can be combined with each other to enable EUP. For example, Simulink uses primarily a visual language, but custom blocks are defined with a domain-specific language. Besides the above mentioned types, recent research includes tangible objects to create a program. Therefore, I add the category tangible programming language to the existing classification and explain it in the following:

Tangible programming language: In spite of typical End-User Programming, tangible programming languages can be used without directly interacting with a computer. Therefore, digital information are represented by tangible objects like cubes or small models, employing the concepts of tangible user interfaces as introduced by Ishii and Ullmer [26]. That way the user can interact and create a program by recombining the tangible objects like stacking, turning or repositioning them [25, 66]. As an example, a system for classrooms was developed to allow children program simple robots [24]. The children can stick elements together to build a control flow diagram (Figure 2.1 shows a selection of the tangible elements) that is digitalized and can be converted into a program. Another approach uses magnetic blocks that are positioned on plates. Based on this, a robot is programmed to do the defined sequence of actions [51, 52]. Tangible programming is limited considering its complexity and scalability. Tangible programming languages are usually used in demonstrative or educational contexts and are suitable to teach children the basic concepts of programming [24].

2.1.3 Visual programming

Out of the above mentioned types of End-User Programming, visual programming is selected to implement EUP into the ROS-VP toolkit because of the following benefits in comparison to other programming languages. They are described by Myers [40]:

Information representation: Graphical programming uses and presents data in a format closer to the way objects are manipulated in the real world and to the user's mental representation of problems [40]. As a result, users understand and process the given information easier, making it especially suitable for non-programmers or novice programmers [43].



Figure 2.1: A collection of tangible programming parts Horn and Jacob [24] used in their concept to control a robot. These objects can be connected, photographed and converted into a program

Higher abstraction: Visual programming languages provide a higher-level of abstraction than textual ones [40], deemphasizing the used syntax. Especially during debugging, graphics can present additional information about the programs state and its current variables than textual based displays can. Complex programs that use concurrent tasks or have real-time requirements are more easily represented employing graphics [40]. For example, visual programming can visualize two systems that run concurrently by drawing them next to each other.

Direct manipulation: Shneiderman [50] argues that it is a more satisfying experience to operate with visible objects. The user is more motivated to create a program by directly manipulating objects with the mouse, like placing or connecting blocks, and not typing code.

Myers [40] also investigated the challenges for visual programming that need to be solved. The most important ones are mentioned and explained as follows:

Representation of large programs: Large programs do not fit properly on a single screen. They require extensive scrolling, which limits the overview and might decrease the user's understanding [40]. This issue can be resolved by a dedicated algorithm to layout the objects on the desktop, zooming combined with dynamic abstraction or

grouping of objects.

Difficult to build editors: The usage of visual programming requires the creation of a dedicated editor. The challenge is that the visually constructed system is not compiler-ready as typical textual code [40]. A module to process and convert the creation into a compilable textual format is required.

Lack of program portability: The output of a visual programming editor is only machine readable and can not be opened without a dedicated application [40]. If the user only wants to simply show or send the program to another person, he is limited to a printout or a PDF/image file.

2.2 Ubiquitous computing

In 1991, Marc Weiser stated that processors will in the future not only be integrated in personal computers or mainframes, but appear in a large variety of devices [63]. This development enables the integration of computing power into the physical environment to support the user. He called this trend ubiquitous computing. These devices are virtually invisible to the user, meaning that he can interact with them without special input devices. His vision is that the workplace consists of hundreds of little displays and computers that enhance a user's productivity [64].

Weiser [63] classified ubiquitous computing devices based on their size in three categories to show different usage scenarios: inch-scale devices can be used with one hand, foot-scale ones require two hands and yard-scale devices are stationary. The active badge is an example for the first category. It is able to automatically open security doors or to configure the telephone to forward the user's calls [63]. A foot-scale device is the developed ParcTab, a touchscreen outfitted device for office-tasks [65], like organizing emails or taking notes. These features can nowadays be found in tablets like the iPad¹ that have a similar form factor. The yard-scale category is intended for stationary devices like the LiveBoard [65], an electronic whiteboard that is connected with the computer.

Kranz and Schmidt [32] argue in their paper that the technological progress supports the emerging of ubiquitous computing. In particular, the computing power is increasing while the energy consumption is decreasing. The storage and memory size grows, while prices are lowering. Interconnectivity is guaranteed as new protocols and modules allow the gadgets to be connected everywhere for shrinking prices to the Internet or other devices. Especially the interconnection of devices was according to Weiser [64] a major issue, since a substantial wireless bandwidth as well as the possibility to switch between networks is required. The

¹For details: <http://www.apple.com/ipad/>, accessed July 16th 2011

mentioned awareness of its location and environment is enabled by the increasing variety of low-cost sensors. Finally, the paper explains that the opportunities to interact with the environment via actuators and displays are decreasing in price although more varieties are offered.

As a next step, Intelligent Environments are explained and examples of implemented solutions are given. As the establishment of these environments have specific software requirements this chapter is concluded with a description of ROS, a middleware for distributed sensor-actuator systems.

2.2.1 Intelligent Environments

Ubiquitous computing focuses on integrating computers seamlessly into the physical world to support users in their daily tasks. An Intelligent Environment, being for example a home that is "aware" of its residents [29], is part of Weiser's vision. It can be enabled by equipping rooms with sensors and actuators, like pressure plates that detect the presence of residents [29], moisture sensors in plants that notice when they need water² or lightbulbs that can be controlled wirelessly [9].

As an example, the Smart Floor project [29] outfitted rooms with force-sensitive load tiles. This allows to distinguish persons based on their weight. As a consequence, it is possible to track the position of a person and for example identify when a resident has not yet left his bed and might have a health problem. This is especially for elderly support a promising and interesting technology. In the course of the Aware Home project [29], keys, glasses and other easily lost objects were combined with tags. The rooms themselves had sensors to identify whether any tags are nearby. In case of losing these objects, the user could be guided to their positions by tracing their tags.

More recent research, focusing on supporting elderly, investigates how existing homes can be upgraded. Linner et al. [35, 36] had the idea to build service modules, walls or complete rooms in their old house equipped with sensors, actuators and displays. This allows the residents to communicate to other persons, like their relatives or their physician by using video walls. In addition, the inhabitants can monitor their health signs like heart rate, body weight or blood pressure with interconnected sensors. The furniture itself can transform to support the user. For example, a kitchen that is hidden in the wall will show up in case the user wants to cook [35]. If retracted, the kitchen cleans and sanitizes itself. The paper also mentions overhead lifting, which is remote controlled by the user and supports handicapped people.

²Botanicalls, <http://http://www.botanicalls.com/>, accessed July 2nd 2011

Apart of elderly support, there are other concepts to implement Intelligent Environment into kitchens, living rooms or the work environment. In the paper of Kranz et al. [33], context-aware kitchen utilities like knives can identify ingredients and suggests additional spices or ingredients that augments the meal. Another concept presented in that paper was to weave electronics into furniture, allowing for example to control the multimedia center from the couch. Besides support at home, there are also concepts to augment the user at work. The research project Labscape [7] focused on enhancing tasks in laboratories, where due to contamination rules, data logging is a major issue. An automatic system was developed, that has the ability to identify reagent bottles with barcode and RFID chips and workers with proximity tags. Additionally, devices like an electronic pipette transmits the amount of used liquid replaced the old one. These information were combined on a touchscreen with a dataflow graph that contained additional information about the intended experiment.

The Cognitive Office project [44] showed how a one-person room could be transformed into an Intelligent Environment. This project included a wide variety of sensors and actuators that were interconnected, starting from power switches, light and temperature sensors over ultrasonic sensors up to traffic RSS feeds or Twitter. The integration of the personal calendar allowed in this project to calculate the travel time, based on the current traffic situation. Lights were automatically activated when a person is present or an intelligent coffee cup warned the user when the coffee is no longer hot.

2.2.2 Robot Operating System (ROS)

As there is not yet a standardized middleware for Intelligent Environment, different solutions are available. Their common purpose is to simplify the interconnection of all sensors with the application logic [44]. In addition, the usage of decoupled software modules are another requirement for the effective implementation of an Intelligent Environment. Suitable middlewares that are available and used in the context of research would be MundoCore [1], GridKit [6] or ROS [41].

Roalter et al. [44] argues that Intelligent Environments have various similarities with robots in their requirements for a middleware. Both use distributed systems, both have the ability to act automatically based on the sensor input and their analysis to enrich the interaction with the user. The Robot Operatings System fits the needs of Intelligent Environments that are modeled as a robot without the ability to move and are therefore called Immobile Robots or "ImmoBots".

ROS implements three modern concepts for distributed software architectures: decentralized peer-to-peer networking, publish-subscribe information distribution and bi-directional

services between components [44]. The different decentralized nodes are running their own software concurrently and are linked with each other by using the services provided by ROS [41]. For this thesis, the usage of decentralized software modules is of main interest. These so-called nodes can be interconnected with each other using the above mentioned publish-subscribe information distribution. This means, that nodes send or receive messages using topics. On a single topic, several nodes can send and receive messages without the knowledge of each other. The nodes implement functions starting with retrieving information of sensors (e.g. capturing camera images³) to processing the information (e.g. grayscaling the image⁴) and showing the results (e.g. viewing the image⁵). These software modules are already partly provided by ROS, others are actively developed and shared within the ROS community. Users can reuse or modify existing nodes and reduce the development effort for a new system significantly. In addition, the complexity of the overall system is reduced as the nodes can be seen as black boxes since only the function and its in- and outputs are important.

As programming a complex robot requires collaboration in the development of the software modules, ROS organizes and structures its software modules into packages. ROS packages are basically directories that contain nodes and messages as well as an XML file to describe its content and dependencies [41]. A collection of packages is usually combined to a stack that allows further structuring.

Quigley et al. [41] define five key characteristics of ROS:

Peer-to-peer: Robots, and especially Intelligent Environments, usually consist of multiple computers that are interconnected by a fixed or wireless network [44]. To enable their communication, ROS connects processes at runtime in a peer-to-peer topology, allowing to distribute the tasks on various machines [41].

Multi-lingual: ROS enables programmers to use their desired language, and can be programmed in C++, Python, Lisp or Octave [41]. All compile options include these languages and the development of ROS for Java⁶ is ongoing as shown at the Google I/O 2011 conference.

Tools based: The operating system is designed around a mini kernel and uses tools for further functionalities like analysis at run-time [41].

Thin: ROS re-uses code from various open-source projects and incorporates them in its nodes. As an example, ROS uses OpenCV [8] to implement its vision algorithms, a library which is widely used for image recognition.

³For details: http://www.ros.org/wiki/uvr_camera, accessed June 30th 2011

⁴For details: http://www.ros.org/wiki/image_proc, accessed June 30th 2011

⁵For details: http://www.ros.org/wiki/image_view, accessed June 30th 2011

⁶rosjava, <http://code.google.com/p/rosjava/>, accessed June 30th, 2011

Free and open-source: In contrast to proprietary environments as Microsoft Robotics Studio, ROS is designed as a free and open source software, which greatly enhances its capabilities for debugging on any level of the software stack [41]. In addition, the contribution of the community is an important element of the success of open-source software [59].

To conclude, ROS is due to its abstraction of software into nodes and interconnection capabilities suitable for Intelligent Environments. Its main issues are that ROS targets at the moment expert-users as its documentation is limited. The user has as a consequence in the beginning a very steep learning curve until he can create his own projects. As I consider this as the major challenge to become more successful, this thesis will present a concept to enable End-User Programming for ROS.

Chapter 3

Requirements analysis

The goal of this thesis is to enable End-User Programming for complex robotic system and Intelligent Environments. As this definition is vague, this chapter will clarify the specific requirements. Four use cases are presented that explain how the ROS-VP toolkit should be utilized. In addition general user needs are presented with the focus on toolkits as a source of user innovation and the proper integration into existing development processes.

3.1 Use cases

According to the classification by Cockburn [11], all four uses cases are casual, meaning that they describe and summarize the scenario. For a better understanding, I explain in a first step the preconditions, the artifacts and the actors of the use cases. That way, the scope of the ROS-VP toolkit is laid out. The first use case is about the end-user Stephanie, who wants to create a simple ROS program. The second one focuses on Alexander, who imports, understands and modifies an existing ROS program. The third use case presents the expert-user Florian, who customizes ROS packages and uses the ROS-VP toolkit for rapid prototyping of new ROS programs. The fourth shows how the ROS developer Martina enhances her packages by configuring them for the ROS-VP toolkit.

3.1.1 Explanations

The preconditions of all use cases are, first of all, that ROS is installed and properly configured. Secondly, the user has successfully installed the ROS-VP toolkit. And thirdly, it must be ensured, that the ROS-VP toolkit is able to access the ROS package directories, to parse all containing files and to create a file in each directory.

The use cases mention four different artifacts that are described in the following paragraph:

ROS-VP toolkit: This toolkit is developed in this thesis to enable End-User Programming for complex robot systems and Intelligent Environment. It employs the idea of visual programming to create a ROS program.

ROS-VP packages: These are two packages provided with the ROS-VP toolkit. They offer basic operations for strings and numbers and are suited for end-users to create simple ROS programs.

Launch file: A launch file¹ is the artifact, used by ROS to start a program. It is an XML file in which the configuration of nodes and its parameters are defined. To allow a smooth integration into the existing development process, this file acts as the format the created ROS program can be exported to. To allow their modification, these files can also be imported into the toolkit.

ROS-VP package file: To allow editing and configuration of ROS packages for expert-users, the ROS toolkit creates the ROS-VP package file after a package was analyzed. It is an XML file, called `vp_package.xml`, that contains all extracted information and allows expert-users to alter its content.

The level of experience of the actors in the use cases differ, as the use cases visualize different scenarios the toolkit can be applied to. As a consequence, I named the users differently and characterize them as follows:

Stephanie: Stephanie studies mechanical engineering and wants to control a quadcopter using ROS. To understand the core principles, she has finished the ROS tutorial². For the beginning Stephanie has no intention to create her own package or nodes, but to reuse existing ones.

Alexander: Alexander studies electrical engineering and attends a class about Intelligent Environments. In its course, ROS was introduced as middleware and he used ROS in a homework where he created a launch file. Additionally, Alexander used Simulink to create simulations for control systems and learned Python in a class about distributed computing.

Florian: Florian studies electrical engineering and was introduced to ROS in the course of a research project. The project implemented ROS to control a robot and Florian modified ROS nodes and created several launch files. Florian is an experienced programmer and used visual programming techniques in Simulink.

Martina: Martina works at the informatics department and researches in the field of image recognition. Therefore, she created a ROS package in her departments stack where she contributes several ROS nodes. This package is shared within the ROS community³. She can be considered an expert-user and professional programmer.

¹For details: <http://www.ros.org/wiki/roslaunch/XML>, accessed July 10th 2011

²For details: <http://www.ros.org/wiki/ROS/Tutorials>, accessed June 17th 2011

³For details: <http://www.ros.org/browse/list.php>, accessed July 10th 2011

Each of the persons wants to use the ROS toolkit for different purposes. However in each use case, only one person is involved.

3.1.2 Number processing for end-users (Stephanie)

Stephanie's goal is to get accustomed to ROS and considers the ROS-VP toolkit as a possibility to simplify the launch file creation. As her quadcopter primarily uses number messages, she wants to start with a simple ROS program, which processes numbers using basic operations (see Figure 3.1).

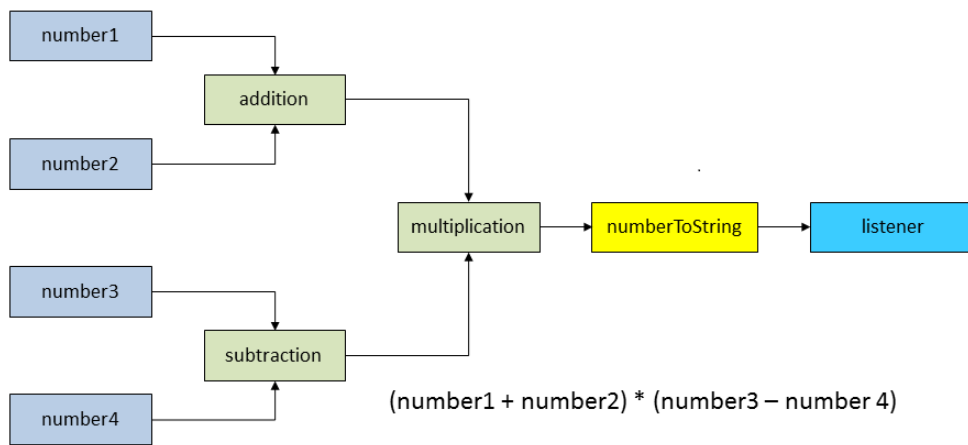


Figure 3.1: Showing a basic dataflow (for details on the coloring scheme see Section 5.2.1) to solve the equation and logging the converted results

As Stephanie has no real knowledge about ROS packages and nodes, she wants to start with the ROS tutorial package and the ROS-VP packages for strings and numbers to create her dataflow. Therefore, she starts the built-in wizard of the ROS-VP toolkit, defines the required parameters and selects all three packages. The editor consists of a palette containing the ROS nodes as well as the active area. The palette is filled with the three selected packages, which the ROS-VP toolkit analyzes by parsing the source code of the nodes and extracting the required information about publishers and subscribers. In the loaded palette, Stephanie identifies the required nodes and places. Stephanie places, configures and connects the required nodes to build her ROS program. As soon as she is satisfied with her first creation, she exports the creation into a launch file. Stephanie starts the ROS master node and executes the launch file using a terminal. To visualize the running system and its nodes, she switches the ROS-VP toolkit to active mode. This way, she can ensure that her creation is running according to her design.

In a second iteration, Stephanie switches back to edit mode and imports the previous

created launch file. At first, she corrects the minor issues that she identified in the active mode. Next, she adds additional nodes to extend the existing ROS program. As an example, Stephanie wants to understand how the dataflow can be controlled. She replaces a number generator node with a random one and adds a control flow block which redirects all messages that do not match the condition. After further iterations, Stephanie is satisfied with her result and is now able to accomplish a more challenging task.

3.1.3 Extension of ROS programs (Alexander)

Alexander wants to modify an existing launch file with the ROS-VP toolkit. He is not the creator of the original launch file and therefore has only a rough idea about its behavior.

Thus, after the configured ROS-VP toolkit is started, he imports the existing launch file. The editor fills itself with the nodes of the file and Alexander changes the layout of the nodes. As he wants to get an idea of the system, he views which messages are used by the topics and nodes and the configuration of the nodes. Additionally, he opens the source code of the nodes to understand how the inputs and outputs of the nodes are linked.

After Alexander understood how the program can be modified, he adds new nodes and remaps the topics of the existing nodes. In the end, he exports the system into a launch file and uses it for his own projects.

3.1.4 Rapid prototyping for expert-users (Florian)

Florian wants to use the ROS-VP toolkit for the creation of launch files. He already used the ROS toolkit several times before and is therefore accustomed to it.

As a first step, Florian wants to customize the packages he uses the most. Thus, he enters the package folder and identifies the relevant ROS-VP package file. He opens it and adds the missing information, as some of the publishers and subscribers were not identified by the toolkit. In addition, he modifies the category attribute of the nodes to change their coloring within the editor. After he is finished with the modifications, he starts the configured ROS-VP toolkit. The custom ROS-VP package file is loaded and the nodes are colored based on their defined category.

The goal of Florian is to create a ROS program that processes an image stream. Therefore, he connects the nodes and configures them with param attributes. Florian also wants to use a parameter server, therefore he simply places that block in the editor and adds the

parameters to the server. After he is satisfied with the created system, he exports it as a launch file.

3.1.5 Support for custom packages (Martina)

As Martina wants to support the ROS-VP toolkit, she integrates the ROS-VP package file into her package and shares it. Her package contains nodes, which do not use the standard publishing and subscription syntax and the toolkit is not able to extract all information. Martina wants to modify the file and share the corrected version of it.

Martina starts the configured ROS-VP toolkit that analyzes her package and automatically creates the ROS-VP package file. She opens the file in a text editor and adds unidentified topics by herself and categorizes her nodes. After she is finished, she includes the file in the uploaded package and shares it with the community. As a consequence, other users of the ROS-VP toolkit have the properly configured ROS-VP package file and do not need to configure it on their own.

3.2 User needs

Besides the four use cases described above, the ROS-VP toolkit has to fulfill general user needs. These needs are divided into two categories. The first one focuses on the aspect that the toolkit acts as a source for user innovation. The second one shows the needs expert-users have to integrate the ROS toolkit into their existing development process.

3.2.1 Toolkit for user innovation

The ROS-VP toolkit should open ROS to a wider audience, by enabling a broad community to create launch files without having an in-depth knowledge of ROS. Von Hippel [60] defined user needs for toolkits to allow user innovation. These cover the most important aspects to allow end-users the proper usage of the ROS-VP toolkit. Based on von Hippel's work there are five aspects which are crucial to the toolkit's success:

Learning through trial and error: The user should have the opportunity to execute several trial-and-error cycles, using the toolkit [61]. This way, the user can experience the consequences of his design choices and is able to alter the result accordingly. Besides the export of the ROS program as a launch file, the ROS-VP toolkit must allow to re-import it and visualize the running ROS program.

Appropriate solution space: The solution space encompasses the user's freedom of choice, by providing a set of limiting factors [61]. This means that the ROS-VP toolkit should

hide features from new users to allow faster learning. Further options for experienced users are outsourced to separate views or outsource options to the ROS-VP package file.

User friendly tools: To guarantee user friendly tools, it is important that the required skills are already available to the user [60]. The user should not need to adapt to a specific design language. The toolkit can match this need by using a visual programming language, which is closer to the mind mapping of problems (see Section 2.1.3). As the state of the art analysis (see Chapter 4) shows, there are already plenty of applications, which use visual programming editors to include end-users.

Commonly used modules: The provision of commonly used modules enables the user to concentrate on his own design [61]. For the ROS-VP toolkit, this requires the integration of already available packages and nodes that are shared in the community. In addition, two packages enable basic operations for strings and numbers like multiplications of two numbers or comparing strings are provided. That way, the user can focus on building his desired ROS program and has not to develop his own nodes.

Result creation: The toolkit must generate results, which can be integrated into the existing workflow [61]. As any required conversion effort reduces the benefit for the user, the output artifact should be usable without any modifications. For the ROS-VP toolkit, the integration of launch files is the best choice. These files already exist in the development process and the user can apply them without prior conversion or modifications.

3.2.2 Integration into existing development process

In contrast to end-users, experienced users have different needs that should also be fulfilled by the ROS-VP toolkit. As they have established their own ROS development process, a seamless integration is required. Therefore, Szekely [55] proposes requirements for prototyping tools, which are partly suitable for the ROS toolkit. In combination with the user needs concerning minimal additional effort and maximal usability of results [47] the following four user needs are identified and clustered:

Minimal Conversion Effort: The development process is limited by budget and time constraints. Reducing the needed conversion effort is important as otherwise the user has no interest in testing the ROS-VP toolkit. On one hand, this means, that the installation and setup process is fast and simple. On other hand, the toolkit should focus on the ease of learning and reducing the time to create the first satisfying results.

Quick Setup: A quick setup especially in the beginning is important to reduce the barriers of usage [47]. There should be no need for lengthy installation instructions or user manuals to set it up. In addition, the user should not have the need

to install other packages that are not delivered with the toolkit. This can be accomplished by using automatic installation or the integration into an existing application. For example, if the ROS-VP toolkit is an Eclipse plug-in, the user can install it using a simple url [49].

Ease of Learning: The time to understand the basic concepts of the toolkit should be limited [55]. The user interface and the usage concepts should be self-explanatory and intuitively usable. Utilizing elements that are known to the user reduces the time to adapt. The ROS-VP toolkit could for example use the same visual programming language as existing applications do (Chapter 4 shows a selection of applications with a visual programming editor).

Integration into Existing Process: Existing development processes should not have the need to change because of the ROS-VP toolkit. Accordingly, a smooth integration into the existing processes can be achieved by using existing artifacts for the input as well as for the output.

Use of Existing Artifacts as Input: The already established development process creates artifacts [47]. Concerning ROS, the existing processes uses packages that contain nodes and messages as well as launch files to create a ROS program. The toolkit must be able to integrate as a consequence existing packages and launch files.

Applicability of Resulting Artifacts: After using the toolkit the output artifact should be implementable into the existing process [55]. This can be accomplished by utilizing file formats that are used in the development process or are easily adjustable by the user like XML. ROS programs are defined using a launch file. The ROS-VP toolkit must be able to export a created ROS program to a launch file that allows the user to execute it without any further conversion.

3.3 Requirements

Based on the use cases and user needs, I derive the specific requirements for the ROS-VP toolkit. These requirements are classified according to Sommerville [53] into functional and non-functional requirements:

Functional requirements: Services or features that characterize the toolkit's behavior are part of this category [53]. Users themselves are the main beneficiaries of these requirements and future development will add new functional requirements to enhance the toolkit.

Non-functional requirements: These requirements define constraints on requirements for the software architecture or other limitations in the development process [53]. Future development and maintenance has a major influence on this topic, as the developer of the ROS-VP toolkit himself and not the user is the main beneficiary of proper defined non-functional requirements.

Both requirement categories are prioritized according to the MoSCoW method that was developed by Clegg and Barker [10]. This scheme rates all requirements based on four different classifications:

Must: Fulfilling requirements of this category is essential for the success of the ROS-VP toolkit. In case of non-compliance, the user's as well as other stakeholders' needs are not matched and the whole application can be considered as a failure.

Should: These requirements are important, but not crucial for the toolkits success. Completion of these features can be integrated in a later iteration, while only a work-around is implemented in the first prototype.

Could: All requirements of this category can be considered as features that are nice to have. They would enhance the application and should be part of the ROS-VP toolkit. In case of underestimated time constraints, these requirements can be postponed without any work-around.

Won't: These requirements are not considered to be delivered for the first version of the ROS-VP toolkit and might be implemented in future iterations of the toolkit.

3.3.1 Functional requirements

The functional requirements are prioritized in Table 3.1. Each line consists of a short version which summarizes the requirement, a description and finally a priority based on the MoSCoW scheme. These requirements act as the foundation for the concept of the ROS-VP toolkit.

Short version	Description	Priority
Visual programming	The editor uses a visual programming language	Must
Integration of packages	Existing and custom ROS packages are integrated	Must
Export of launch files	Created ROS programs can be exported as launch files	Must
Connection of nodes	ROS nodes can be connected to topics using the ROS-VP toolkit	Must
Basic ROS packages	Two ROS packages with basic operations for numbers and strings are delivered	Must
Import of launch files	Existing ROS launch files can be imported into the toolkit	Should
Open source code	The ROS-VP toolkit can find and open the source code of nodes	Should
Configure nodes	Using the ROS-VP toolkit, nodes can be configured with parameters like their name or param attributes	Should
ROS-VP package file	Collected information about a ROS package are stored into a customizable XML files	Should
Run-time support	The execution of launch files and a visualization of running nodes is supported	Could
Automatic categorization	The nodes are analyzed and automatically categorized	Won't
Node identification	The ROS-VP toolkit offers support to find a node by the user's description	Won't

Table 3.1: Functional requirements for the ROS-VP toolkit

3.3.2 Non-functional requirements

In Table 3.2 the non-functional requirements are listed and prioritized. Like the functional requirements, each requirement consists out of a short version, its description and its priority. These requirements are especially important for the software architecture of the concept.

Short version	Description	Priority
Common user interface	The ROS-VP toolkit employs a common user interface	Must
Modularity	Software engineering frameworks are employed to structure the software architecture	Must
Reuse of tools	The toolkit reuses existing software artifacts if possible	Should
Ease of deployment	The ROS toolkit can be installed without expert knowledge	Should
Quick setup	Installation does not need prior configuration of the computer	Should
Use of existing knowledge	Expert users should be able to use it without the need to learn new concepts	Should

Table 3.2: Nonfunctional requirements for the ROS-VP toolkit

Chapter 4

State of the art

Having outlined the requirements, this chapter offers an overview of available applications that use a visual programming language. The selection is not exhaustive but do cover different types of visual programming that are analyzed for the concept of this thesis: imperative, dataflow and event-driven programming. The six selected programs have different areas of application, ranging from robotics for non-professionals or professionals over mobile application programming and to the creation of games and animations for children. At the end of this chapter, I analyze the fit of each application for the ROS-VP toolkit and draw a conclusion.

4.1 Imperative programming

Imperative programming defines an application as a sequence of commands that is executed and changes the programs state [48]. Dabek et al. [13] argue, that this can be complex and might lead to buggy software. Consequently, this way of programming is suited for simple scenarios that are properly defined.

4.1.1 LEGO Mindstorms NXT

LEGO Mindstorms NXT (NXT) is a robotics solution that can be used in combination with usual LEGO bricks. The system consists of a processing unit that can be combined with sensors or actuators and the NXT-G application to program the robot [20]. NXT is developed in a way that users are free to design any kind of robot by using common LEGO bricks. Due to the high degree of flexibility, NXT is utilized from hobbyist, researchers or educational organizations. For expert-users, other applications like MatLab/Simulink or Microsoft Robotics Studio are available to program their NXT robot [30].

The NXT-G programming environment uses a visual programming language that is based on National Instruments' LabView[30]. The programming logic is a sequential flow,

meaning that one block is done at a time until the next one is activated. This is visually supported, as all blocks are placed on a "line". Complex systems can be realized by having two or more independent "lines" that run concurrently, like a blinking light and the motor itself. The flow of data can be modeled to the extend that variables can be handed from one block to another. The blocks are delivered with the software, and do cover various areas, like movement of actuators, processing of sensors and conditional operations [30]. The example visualized in Figure 4.1 shows a LEGO NXT robot that moves until an obstacle in front of it appears. At that point the robot brakes and shots at the wall.

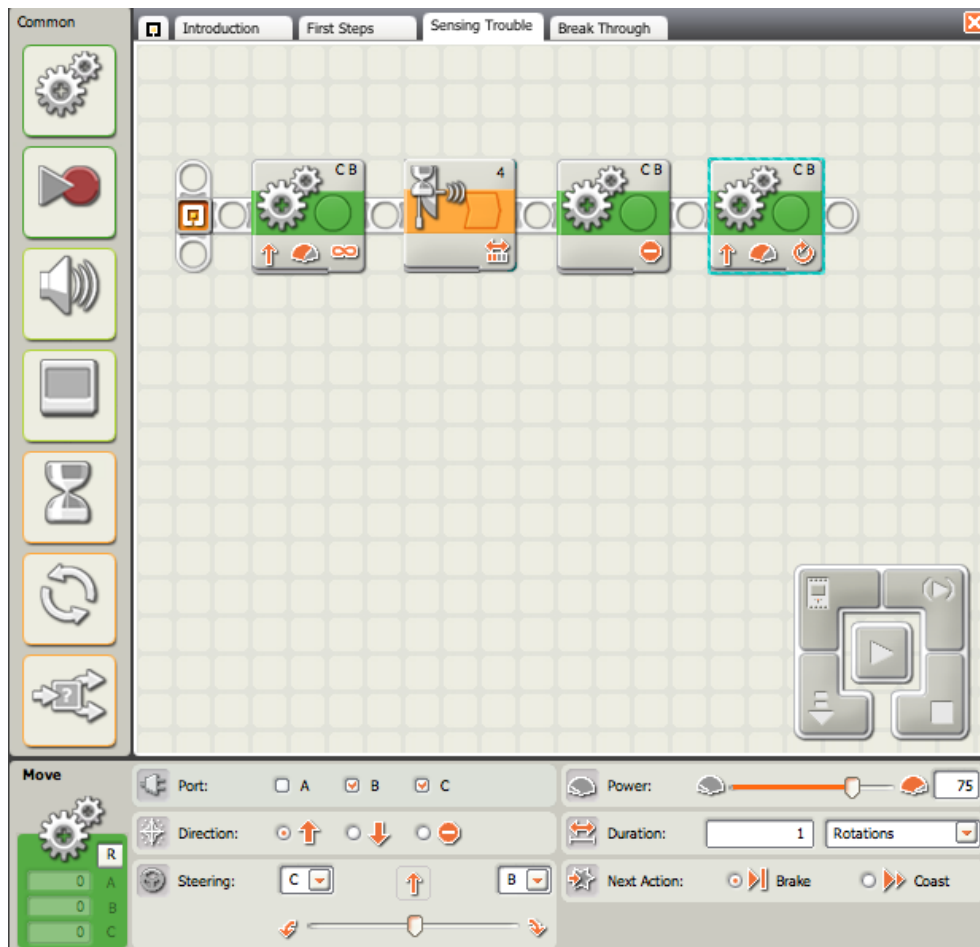


Figure 4.1: The user interface of the NXT-G programming environment that is part of LEGO Mindstorms NXT. The visualized sequence shows a program, that moves forward until an obstacle appears, breaks and then activates the second actuator

NXT-G offers only simple blocks that shows its focus on non-professionals. As its programming logic is based on imperative programming, the user interface provides a line which is used to add the blocks to. Custom blocks or even libraries are not supported by the NXT-G programming environment. Appearance of blocks, like colors or small icons are

adjusted based on the category. In addition blocks like loops change their forms, allowing the user to place blocks inside of it, supporting the user to understand their function. The user interface offers a view at the bottom of the screen that contains further options.

4.2 Dataflow programming

Dataflow programming focuses on the way data is processed and exchanged [23]. This allows the proper mapping of concurrent sensor read-outs and their processing. The modules work solely based on the received data and the sequence of execution can not be predicted.

4.2.1 Microsoft Robotics Developer Studio

Microsoft Robotics Developer Studio (RDS) is developed to allow its usage on any type of robot, as long as a cooperation with Microsoft is present. It is not limited by the processor architecture, supporting 8-bit to 32-bit processor or multi-processor robots [30]. As an example, the above mentioned LEGO Mindstorms NXT robot can be controlled with RDS. The program, that is created with the visual programming editor, can be run and debugged inside a simulator.

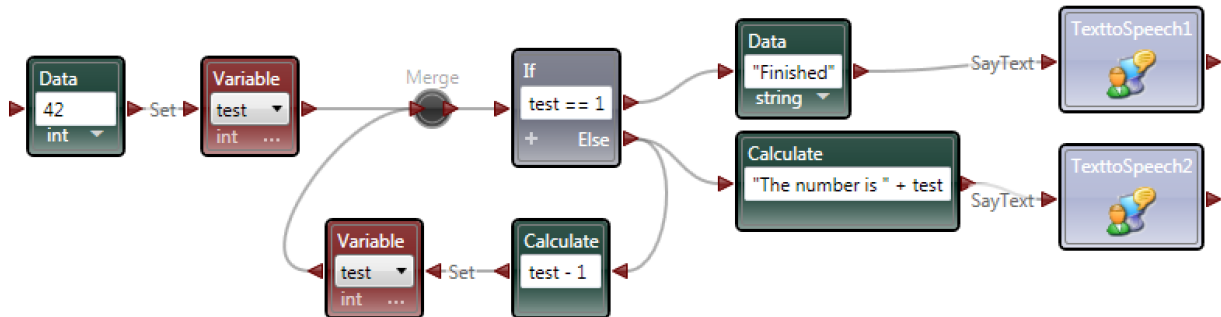


Figure 4.2: A simple program created in Microsoft Robotics Developer Studio that checks the value of a number. If the value of the variable "test" is equal to 1, the word "Finished" is spoken by the computer. In any other case, the number is decremented and its value is said by the computer

A robot is programmed using the visual programming editor, by dragging and dropping of blocks and connecting these with directed graphs [30]. Each node has a text box for the output of a signal as visualized in Figure 4.2. As a consequence, conditional nodes have for each rule a text box and an output that is used in case the condition is matched. The blocks themselves are partly delivered by Microsoft, partly by other developers or

can be created by the user himself using the .NET framework [22, 27]. In spite of programming a robot with existing blocks, creating new ones require programming experience.

RDS is suitable for novice and professionals programmers in industry or research context [27]. The robot system is modeled with blocks that are connected with directed graphs, showing the flow of data. In case multiple in- or outputs are available, a pop-up menu appears to let the user select the right one. Custom libraries can be created by the user and are supported by the application. The appearance of blocks, in particular the color, the size and possible icons, change depending on the block's type.

4.2.2 Simulink

Simulink is an extension of the math application MatLab. Its purpose is to model complex systems in physics, financial or technical context [2]. These models are dataflow driven, meaning that graphical elements are connected by the flow of data.

The user interface is a visual programming editor, providing a pop-up menu which offers a variety of predefined blocks. These blocks can be dropped onto the editor and connected with directed graphs in case the data types match. Besides the huge library, users can create customizable blocks and program them by using the same syntax as MatLab, a domain-specific C-Code [2]. Other toolboxes are offered that allow the user to create for example a finite state machine inside of the visual programming editor [2].

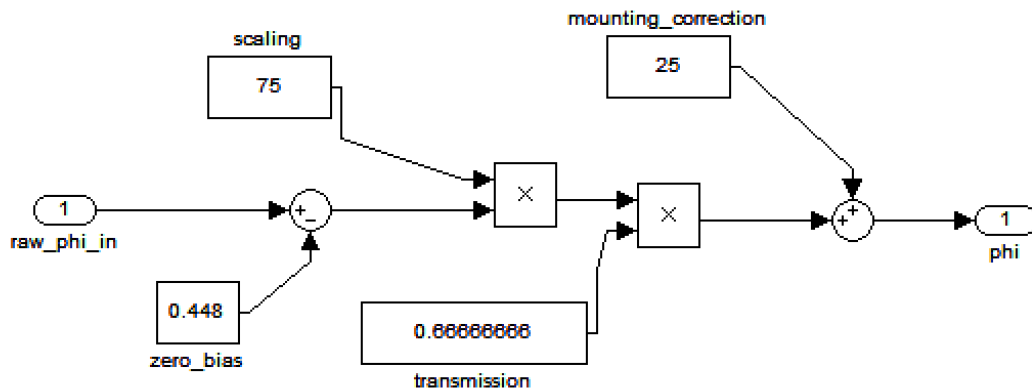


Figure 4.3: The visual editor of Simulink shows a program to convert the measured angular value to corrected one. Therefore the program adds offsets and scales the raw value

The target audience of Simulink are professionals, which have to model complex systems.

The blocks can be combined by drawing the directed graphs between blocks, defining the flow of data. Custom libraries are not supported by the application, but custom blocks are enabled. The appearance of blocks is only partly adapting to the function of the node, changing their shape or adding anchors to visualize an additional in- or output. Most nodes are white rectangles labeled with their purpose or value.

4.3 Event-driven programming

After imperative as well as dataflow programming was investigated, the third analyzed type is event-driven programming. Thus the program is not sequential from beginning to end, but handles events like pressing the space bar on the keyboard [13]. In an application with a graphical user interface, its logic is triggered based on the inputs of the user. This means, that the flow of information can not be predicted.

4.3.1 App Inventor

Google's App Inventor for Android (AIA), is an application that employs a visual programming language to build simple Android applications. It is designed for end-users that have basic knowledge about programming [54] and is usable in the browser after installing a package, while compilation and storing of the creation is done on a Google server. The App Inventor is based on a two-step process. In the first step, the user interface is defined using the application designer. In a second step, the application logic has to be defined using an event-driven visual programming editor. After the program is created, the user can test it in an emulator or export and install it on their Android phone [37].

For this thesis, only the blocks editor for the application logic will be analyzed. After a user interface is created using the editor, the user must create the logic that handles the events [37]. In contrast to the previous visual programming languages, AIA offers blocks that are added to each other. The user has a selection of event handlers that are based on the components in the user interface. For example Figure 4.4 shows the event handler in case the button1 is pressed. The user defines by placing blocks in this event handlers, what should happen. The shape and color of a block indicates its category. This shows the user which block is expected. As an example, variables are blocks with a knob on the left side.

The App Inventor targets users that are not yet accustomed to build Java applications but already have basic programming knowledge. This makes it useful for educational purposes or for end-users that wants to create simple applications. The blocks are combined to build a program by stacking them up and placing them into cutouts of each other. Custom

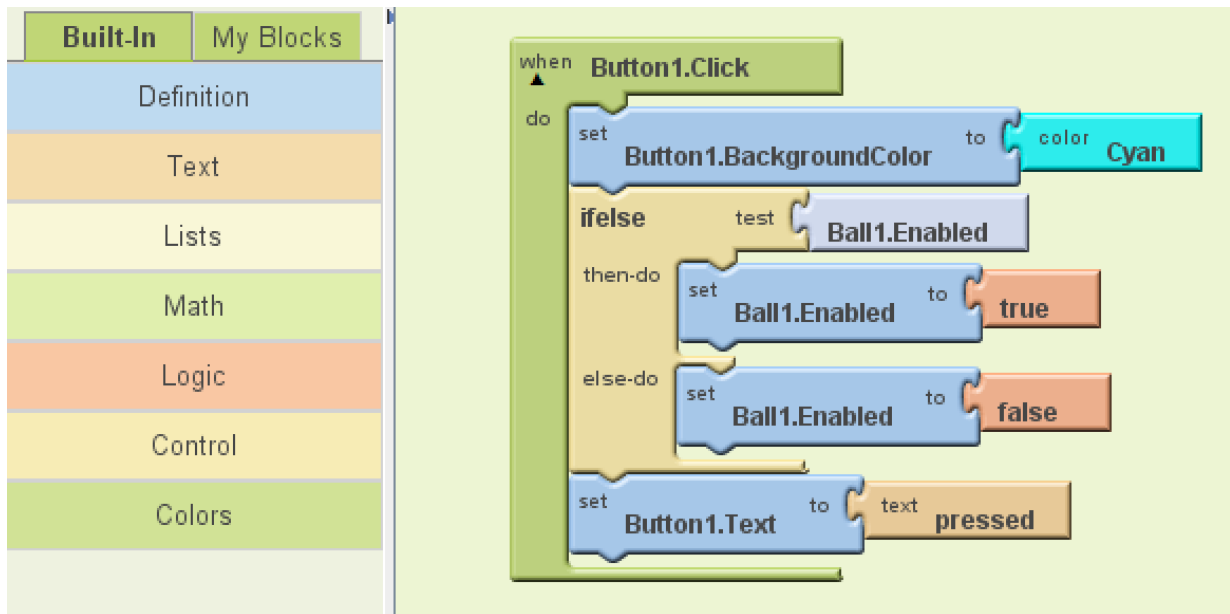


Figure 4.4: App Inventor for Android block that enables or disables a ball and changes the color and text of the pressed button

libraries or blocks are not supported, as it focuses on end-users. The form, color and the cutouts change from one block to another, showing which inputs are required. By adding a block at the appropriate cutout, the blocks are combined. The user interface does not offer further functions to configure a block.

4.3.2 StarLogo TNG

Another example for event-driven programming is the StarLogo TNG project. It was developed at the MIT to eliminate the need for textual input for the text-based programming environment StarLogo [45]. StarLogo TNG (the next generation) extends the language by adding a visual programming layer on top to make it more attractive to end-users. The user has a palette of predefined blocks, recombines them and develops a program to control a 3D agent.

StarLogo TNG's programming environment itself consists primarily of a palette of blocks and an editor area, where recombination allows the creation of a program (see Figure 4.5). The blocks have forms and colors based on their functions. Rounded corners symbolize booleans, hexagons variables with a value, rectangle with an arrow at their bottom center are commands like movement and allow stacking them up to build a sequential command chain. More complex ones like control flow blocks or functions, have cutouts to show what kind of blocks they require. This concept is very similar to the one of the App

Inventor (see Section 4.3.1). The coloring itself is also used to further classify the blocks. As an example control blocks are colored orange, movements red and math calculations are blue. The blocks resize themselves if needed, so that for example an "if"-block can contain various conditions and is not limited to a single one. The user can pick the blocks based on a predefined set of events. The cutout also directly shows the user if the code is not yet complete, limiting the possibilities for error [45, 62].

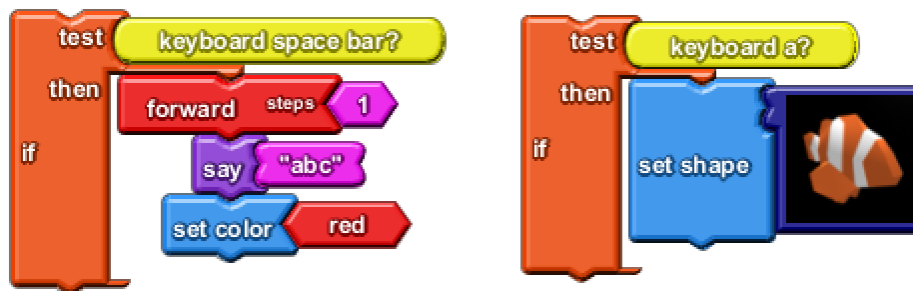


Figure 4.5: The StarLogo TNG visual editor shows how pressing the space bar or the "a"-key is handled. Using the space bar moves the agent and let it say "abc", a keystroke on the "a"-key changes the shape of the 3D agent to a fish

StarLogo TNG is created for end-users to build simple 3D applications like games and to understand the needed concepts. The applicable scenario is limited to controlling a 3D agent in a virtual environment. That is why this application is also suitable for school and educational purpose. Due to the simplification of the programming and the built-in error prevention by its shapes, StarLogo TNG offers students a more structured programming environment than textual based ones [5]. Blocks are placed in cutouts or added to each other. Neither custom libraries nor programmable blocks are supported. The appearance of the blocks is massively based on their function as well as on the required input types. The coloring is based on the categories, like event handlers, movements or mathematical operations. The application does not offer an additional menu to configure blocks.

4.3.3 Scratch

My third example for event-driven programming is the Scratch project. Developed at the MIT Media Lab, it enables children with an age between 8 and 16 to create and share stories, games and animations over the Internet [57]. Its main purpose is to allow the creation of content that the target audience likes with a simple user interface. Besides providing a programming environment for children, Scratch enables the users to share their creation with each other [43].

The Scratch user interface consists of a library of graphical blocks, an editor for sequential programming of actions or graphics, a library for sprites and a preview window [43]. The blocks are snapped to each other and the user is able to create scripts. As an example, the user defines what happens, when on the keyboard the up arrow or a button is pressed (see Figure 4.6). Each possible event receives a block to define the reaction of the object. Therefore, each object has its own editor to create its scripts. The library contains besides the basic operations, like control flow or math blocks, blocks to play a sound or move objects in the animation. The shape indicates the type of the block and cutouts show what kind of blocks it requires. The creation can always be previewed and evaluated.

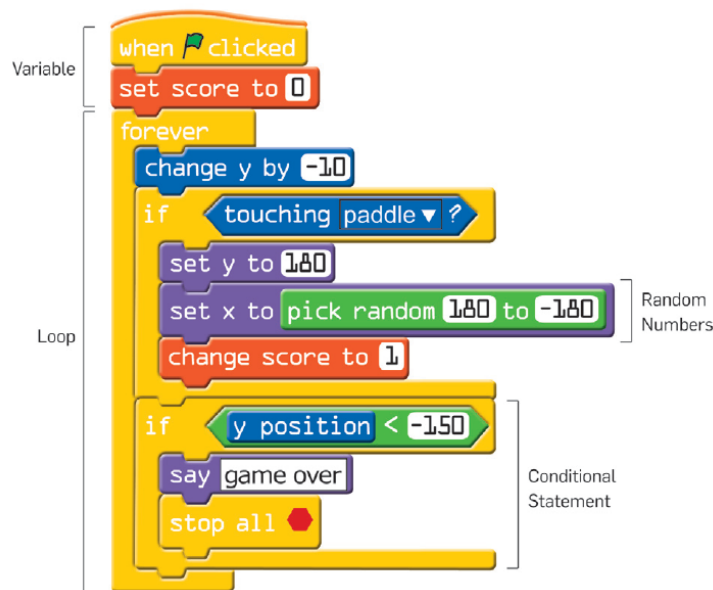


Figure 4.6: A block from Scratch that is executed when the green flag button is pressed. The loop runs forever and changes the position of an object until the y-position is too low and the game is stopped [43]

Scratch is a very simple application, using the same concepts of visual programming as StarLogo TNG and App Inventor, to enable creation of websites and its content for children. As the two previous applications, the blocks are combined by adding them together and filling cutouts. As this system is created for children, the user is not encouraged to create their own blocks or even libraries. As the identification of blocks seem to be important for the target audience, different colors and shapes are used. The applications uses views to create new sprites or to set the name of a variable, but to me, it seems that option views to configure blocks are not intended.

4.4 Results

To compare the six analyzed applications, I selected seven criteria, presented in the following, and extracted these information to Table 4.1. Based on this, I decide which application should be used as best practice and acts as a basis for the ROS-VP toolkit and its visual programming language.

Area of application: This category describes the application's typical utilization scenario, like complex robotics systems.

Type: The type represents the kind of programming style that is used. The possible answers hereby are imperative, dataflow or event-driven.

Audience: The audience refers to the target audience of the application. This might be professionals, like researchers developing new control systems or employees to develop a new program, or non-professionals that includes educational purposes or hobbyists.

Connecting nodes: Analyzing the visual programming languages, the six application use different concepts to combine blocks. One concept is to simply place them on a line representing the sequence of execution, another one connects the blocks by lines to show the dataflow and the third stacks and snaps blocks to each other and places them into cutouts.

Custom libraries: This criteria analyzes if custom libraries or blocks are supported. Therefore, it is possible to offer an interface to create blocks as part of a library and import them or to allow the user to program the logic of single blocks.

Block categorization: This category refers if the majority of blocks that are used in the editor are categorized and have different appearances according to their purpose. The analyzed visual programming languages employs blocks that may change their color, shape or even both.

Option views: Some of the applications offer option views, menus that allow the user to configure some blocks. These views must be used to configure a block and not just to select a sound file or create an image.

The analysis of the above applications showed three types of programming types, that can be implemented with visual programming languages: imperative, dataflow and event-driven programming. After the three types are compared to each other, it is obvious, that only dataflow programming is used to create complex systems and is used by end-users and professionals. These are also the only kind of applications that have a true support for custom libraries or blocks. In addition, the characteristics of ROS (see Section 2.2.2), especially the fact that the ROS nodes are connected to each other with topics, show that the middleware itself is dataflow driven. Therefore, the visual programming concepts of Microsoft Robotics Studio as well as Simulink are included in the development of the ROS-VP toolkit. Applied on the concept, it means that the user should be able to import

Applications	Lego Mindstorms NXT	Microsoft Robotics Developer Studio	Simulink	App Inventor for Android	StarLogo TNG	Scratch
Developer	LEGO	Microsoft	MathWorks	Google	MIT	MIT
Area of application	Simple robotic systems	Complex robotic systems	Complex mathematical models	Android application	3D games	Interactive animations
Type	Imperative	Dataflow	Dataflow	Event-driven	Event-driven	Event-driven
Audience	Non- professional	Professional	Professional	Non- professional	Non- professional	Non- professional (children)
Connecting nodes	Placing on a Line	Directed graphs	Directed graphs	Stacking	Stacking	Stacking
Custom libraries	No	Yes	Blocks	No	No	No
Block categorization	Shape, color	Color	Partly shape	Shape, color	Shape, color	Shape, color
Option views	Yes	Yes	Yes	No	No	No

Table 4.1: Summary of visual programming applications

custom libraries that contain blocks. These use custom coloring scheme like applied by RDS and are connected with directed graphs that symbolize the flow of information. The configuration of blocks is done by using a separate options views.

Chapter 5

Concept for End-User Programming in ROS

This chapter covers the concept of the ROS-VP toolkit. It is designed to match the previously defined functional (see Table 3.1) and partly the non-functional requirements (see Table 3.2), which were prioritized with a Must or Should. In addition, the best practices that were concluded at the end of the state of the art analysis are implemented (see Section 4.4).

The concept is explained in four sections. The first one describes the user interface. Focus lies on the integration into Eclipse using the provided components and the implemented visual programming language. The second and third section describe the two different modes of the toolkit: the edit and the active mode. The edit mode allows to create and modify a ROS program, while the active mode shows the ROS program at run-time. The last section explains the two packages, one has nodes to process and generate numbers and one offers basic operations for strings.

5.1 User interface

Two requirements are identified for the user interface: users should be familiar with the user interface and the editor must employ a visual programming language. The first point, the provision of a familiar user interface, will be solved by integrating the toolkit as a plug-in into Eclipse, a widespread programming environment [42]. The second part of this section describes the concept of the visual programming editor that incorporates the best practices from the summary in Chapter 4.

5.1.1 Integration into Eclipse

To understand the reasoning for the integration of the ROS-VP toolkit into Eclipse, the paper of des Rivieres and Wiegand [14] provides an overview of Eclipse as an integrated development environment (IDE). It was established as open-ended and language neutral programming environment that can be used on different operating systems like Windows or Linux. Eclipse itself offers primarily generic functionalities that can be extended in form of plug-ins, developed in Java and with application specific frameworks. As it is an open-source project, a large community exists that actively contributes various plug-ins, for example new languages like Python¹ and L^AT_EX² or features like the integration of subversion³. Eclipse's versatile features enables users to keep their familiar programming environment even though they program in Java or Python and write their documentation in L^AT_EX. The Eclipse adoption study [42] shows that in 65% of the questioned companies, Eclipse was used. This means that the user interface of Eclipse is already familiar to potential users and might already be applied in the existing ROS development process to create new nodes. Additionally, Ehrig et al. [15] elaborated in their paper that Eclipse is a useful platform for the implementation of a visual programming editor.

As a consequence, the ROS-VP toolkit will be implemented into Eclipse as a plug-in. Therefore, Eclipse provides standard components (a selection is shown in Figure 5.1) to integrate it into the existing user interface. I chose the following four user interface components to retain the Eclipse user experience for the ROS-VP toolkit:

Wizard: The wizard is a pop-up window that can be used to configure and create a new resource like a project or a file in a multi-step process [3]. As a consequence, this component is considered to be familiar to Eclipse users. I use the wizard for the ROS-VP toolkit to ask from the user arguments like the environmental parameters to access the ROS tools or which packages the user needs (see Section 3.1).

Editor: The editor component is placed in the center of Eclipse and is opened by a user with a double-click on a file in the package explorer [49]. It is usually used to display the source code of a file. For the ROS-VP toolkit, this component is used to implement the visual programming language.

Action bar: The action bar is located at the top of Eclipse and contains buttons for example to create a new project or undo the last step [49]. The same options can also be added to a context menu that is available in the editor. For the ROS-VP toolkit, I use the action bar and the context menu to allow the user to access options that are not possible inside of the visual programming editor. For example, the action bar could offer the option to configure or connect nodes.

¹For details: <http://pydev.org/>, accessed July 10th 2011

²For details: <http://texlipse.sourceforge.net/>, accessed July 10th 2011

³For details: <http://subclipse.tigris.org/>, accessed July 10th 2011

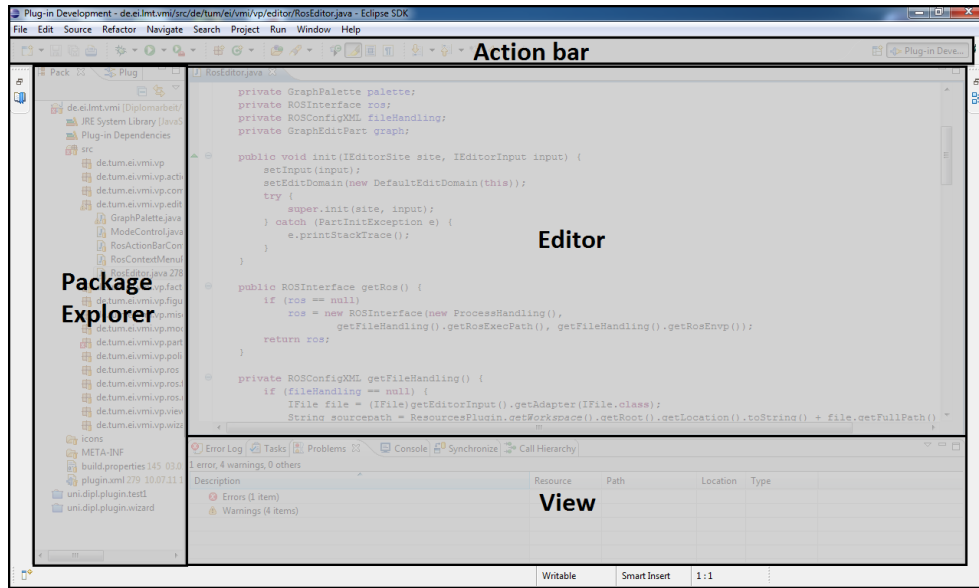


Figure 5.1: A screenshot of the user interface of Eclipse shows the package explorer and three extension points the ROS-VP toolkit uses: editor, actionbar and view

View: A view is applied to navigate the hierarchy of a file, open an editor (like the Package Explorer view in Figure 5.1), display or modify properties of the active editor [49]. Views may contain widgets like buttons, lists or text boxes. The ROS-VP toolkit will for example use a view to configure or connect nodes.

5.1.2 Visual programming language

The analysis of the state of the art (for the results see Section 4.4) showed that the dataflow based visual programming language employed by Microsoft Robotics Developer Studio and Simulink is most suitable for the ROS-VP toolkit. Consequently, nodes are connected to each other by directed graphs. The editor itself should consist out of a palette that contains the node libraries and the active area where nodes can be connected.



Figure 5.2: Two ROS nodes are connected inside of the ROS editor over the sum topic

The user creates a program with the toolkit by dragging nodes onto the active area. A ROS node is represented by a rectangle, labeled with a name. The node's background color is based on its category (the categories are explained in Section 5.2.1), that supports users to see the purpose of a node. Each node should initially be named based on its type

as well as a running number, e.g. the first node of the type talker is named talker1. The connection between two nodes is visualized by a directed graph and is labeled with the name of the ROS topic, as visualized in Figure 5.2.

5.2 Edit mode

The edit mode allows the user to create his own ROS program and therefore has to fulfill the majority of the functional requirements. During the edit mode, the palette on the left side of the editor is filled with, as required, existing ROS packages and their nodes. These packages are analyzed and information about the containing messages and nodes are stored into the ROS-VP package file. In addition, the user must be able to configure and connect placed nodes by creating topics. Another requirement is the possibility to open and modify the source code of a node. The last two requirements for this mode are that created ROS programs can be exported to launch files and re-imported into the ROS-VP toolkit.

5.2.1 Integration of ROS packages

To reuse existing ROS packages, the ROS-VP toolkit retrieves their directories and searches for executable files. These files are considered to be nodes that are automatically matched with available source files. Additionally, message definition files are searched and their content is analyzed. As a result of this processing the following information are available:

Messages: ROS Messages are defined by simple text files that have the .msg file ending and use Python like syntax⁴. These files contain the descriptions about the ROS specific variables that are inside of the message. The ROS-VP toolkit, retrieves for each message its name, the ROS package name it is part of and all variables of the message. As an example, the "HeaderString" message is part of the rospy_tutorial package and contains two variables: "Header" and "string".

Nodes: ROS nodes are executable files, that are usually combined with a source code. The source is used by the ROS-VP toolkit to retrieve information about which messages are subscribed or published. As I see Python and C++ as the two primary languages, only source files created in these languages are supported.

The information about messages and nodes of each package are automatically stored within the ROS-VP package file, called vp_package.xml, that is placed inside of the package directory. The automatic analysis of publishers and subscribers of the toolkit is limited to the standard syntax as presented in the tutorial⁵ and is not able to retrieve all data from complex syntax that for example some image processing nodes apply. Experienced users

⁴For details: <http://www.ros.org/wiki/msg>, accessed July 15th 2011

⁵For details: <http://www.ros.org/wiki/ROS/Tutorials>, accessed June 30th 2011

can revise the ROS-VP package file and add missing information about subscribers and publishers. The toolkit always searches the package directory for a `vp_package.xml` file. In case one is found, all information are extracted from this file and further processing of the directory is skipped.

In addition, the ROS-VP package file enables the user to manually classify the identified nodes. I defined the following seven categories that cover processing of numbers, strings and images, sources, outputs, control flow as well as uncategorized nodes. These categories implement the coloring scheme that is based on AIA, the App Inventor for Android (see Section 4.3.1), as neither Simulink nor RDS have a properly defined one:

Source: This node retrieves data from sensors or generates them by itself and publishes these information. It publishes then these messages on output topics. Possible input topics are only used to configure or start the node. For example, a node accesses a camera and distributes its video stream. Nodes of this category are colored blue as it is similar to the definition of a variable in AIA.

Math: Any node that receives numbers, processes and then publishes the result are classified as math nodes. As an example, a node adds two integers and publishes the sum. According to the math blocks of the App Inventor, these nodes are green.

String: Nodes that publish processed strings, like concatenation of two string are part of this category. AIA colors these nodes, like control nodes in orange. As LEGO Mindstorms NXT also uses the color orange for control nodes, string nodes are colored yellow and conditional will be colored orange.

Image: This category is used for nodes that modify or process images. For example a node may convert a color image into a grayscale one. Due to the fact, that AIA does not offer these kind of nodes, they are colored red like the AIA logic blocks.

Control: Nodes of this category check if a condition is matched, for example by comparing two messages. Based on this assessment, the node processes the message differently, like publishing it on another topic. The control blocks are colored orange based on AIA and LEGO Mindstorms NXT.

Output: An output node logs or displays messages. This might be a window to show images or just a node which saves messages in a file. As AIA offers no blocks specifically for output, I chose cyan to make it distinguishable from the other nodes.

None: Nodes that are not yet categorized or can not properly be classified are part of this default group. These blocks are colored white.

In order to store these information in a ROS-VP package file, I defined the XML schema as visualized in Figure 5.3. In the following, a sample⁶ ROS-VP package file is presented

⁶The source path is shortened for improved visibility

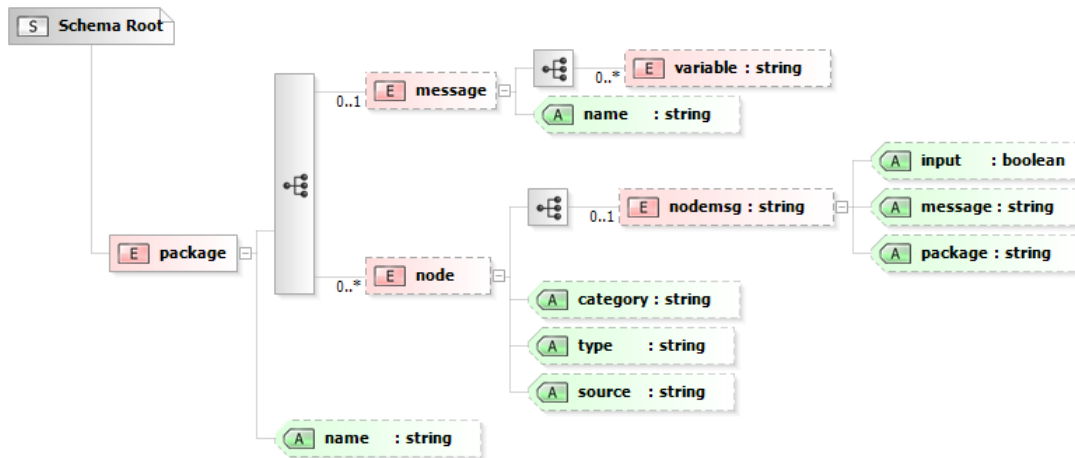


Figure 5.3: Visualization of the XML schema used for the ROS-VP package file. Red objects with an "E" represent XML elements like nodes or messages. Green objects with an "A" are attributes like the category and type of a node

that contains the "HeaderString" message and a "listener.py" node:

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="no"?>
<package name="rospy_tutorials">
  <message name="HeaderString">
    <variable>Header header</variable>
    <variable>string data</variable>
  </message>
  <node category="output" type="listener.py"
    source="../../rospy_tutorials/001_talker_listener/listener.py">
    <nodemsg input="true" message="String" package="std_msgs">
      chatter
    </nodemsg>
  </node>
  <node category="source" type="talker.py"
    source="../../rospy_tutorials/001_talker_listener/talker.py" >
    <nodemsg input="false" message="String" package="std_msgs">
      chatter
    </nodemsg>
  </node>
</package>
```

5.2.2 Connection and configuration of nodes

The user can connect nodes on custom topics by the usage of a dedicated view. Therefore, the ROS-VP tutorial analyzes the messages of both nodes and matches a subscriber and a

publisher as a pair. The user can select matched pairs in the list, define a name for them and create a topic, that lets both nodes communicate to each other. In case a pair with one custom topic is picked, the toolkit suggests the existing name for a new creation to connect several on one topic.

The user can configure every node using a dedicated view, like an image node with camera parameters. By analyzing the online documentation as well as various launch files, I identify the following parameters as the most important ones for the ROS-VP toolkit:

Name: The name of the node, which is used to address it during runtime. Consequently, ROS requires that all nodes have unique names.

Param: The param argument are private parameters for the node. For example, this can be used to define the value of a number generating node.

Remap: One of the key tasks of the toolkit is to allow the user to connect nodes to each other. The remap parameter defines to which topic a message is redirected. This list is synchronized with the custom topics created in the connect nodes view.

Output: The output attribute defines if the standard output is directed to the screen or to a log file.

Respawn: To let a node respawn in case it quits, this parameter must be true.

Namespace: This attributes allows the user to group nodes by setting a namespace.

For the proper configuration of a complex node, the user requires more knowledge than just the used topics. Consequently, the ROS-VP toolkit offers the option to open the source code of a node. This allows the user to see for example the required parameters.

5.2.3 Integration of ROS launch files

The ROS-VP toolkit allows users to export a created ROS program to a launch file and to import an existing program. Launch files are directly supported by ROS and therefore do not require any modifications by the user. By analyzing launch files, I selected the following three top level elements to be supported by the toolkit:

Node: A node element represents a ROS node and contains information about the type, name and name of its containing package. In addition, the attributes to configure a node are part of this element (see Section 5.2.2).

Group: A group is used to define a namespace for multiple nodes. Its child elements are the above mentioned node elements.

Param: The parameter server allows to define global parameters. This element only has two attributes: name and value.

5.3 Active mode

According to the requirements, the ROS-VP toolkit should be able to show the ROS program at run-time. This allows the user to control whether the created ROS program has properly connected nodes. The toolkit offers the active mode to display a running ROS program. It retrieves information about active nodes and their topics from the ROS master node. Since the active mode uses the same graphical elements as the edit mode does, it offers a consistent environment. To show that the nodes are active, the background is colored lightgrey

In contrast to the edit mode, the user's options are reduced during active mode. In particular, nodes can not be configured or connected and the source code is not accessible. Only the options to show the messages of a node or a topic are available.

Retrieving the required information can be achieved in two ways. The first one is to directly communicate with the ROS master node using XMLRPC. The second option is to use the integrated ROS command-line tools. This would allow to reuse the existing ROS tools and to limit the effort to implement a dedicated solution. As the ROS tools are part of the core ROS distribution, they will stay part of future releases. As the active mode's demand for information is limited and the requirements state that existing software artifacts should be integrated, the ROS tools are selected and integrated into the ROS-VP toolkit for information retrieval.

5.4 Provision of basic nodes

The requirements state, that the ROS-VP toolkit has to provide basic operations (see Section 3.3). In addition, the state of the art analysis showed (see Section 4.4), that although custom libraries might be supported, the provision of nodes with basic functionalities is always part of the application. ROS has a huge collection of community-contributed nodes that are primarily customized to the developers requirements. Abstract nodes like basic operations for numbers or strings are therefore not available. End-users who do not want to create their own packages would benefit from these nodes especially in the beginning. Consequently, the ROS-VP toolkit is bundled with a ROS stack that contains two packages with custom ROS-VP package files, one for processing numbers and the other one to modify strings. The third basic category, images, is already covered by various packages.

Those allow retrieval of images from cameras and conversion of those for example to a monochrome version.

5.4.1 Numbers package

The numbers package provides basic operations for integers and floats that are part of the `std_msgs` package⁷ of the core ROS distribution. Consequently, each node has a version for both data types. The following four elementary arithmetic operations and two filters are part of the package:

Addition: The addition node offers two input channels for the summands. These two are added and the sum is published on the output topic.

Subtraction: The subtraction node has one topic for the minuend and one for the subtrahend. The difference is then published to the output topic.

Multiplication: The calculated product of two messages is distributed by this node.

Division: Similar to the other nodes, the division node has two input topics. One is for the dividend and another one for the divisor. On the output topic, the quotient is published.

Average: To smoothen a signal, this node averages a definable number of messages and publishes the result. It can be used for discrete systems, for example a sensor to measure the angular movement of a lever. In case of a slow movement and a low resolution, the signal has "steps" in its signal. This node eliminates these steps.

Median: This node can be used to filter a continuous measurement and eliminate spikes. The user can define the size of the list, from which the median should be picked. As an example, it can be used to filter an analog signal that has noise because of interferences.

The above mentioned calculations are executed, each time a new message is received. In addition to these six nodes, the package also provides two sources to generate numbers, a converter between the two data types and the two control flow nodes:

Number: This node acts as a simple source and publishes a fixed number on its output topic. The user can define the number as well as the interval between two messages.

Random: A random number generator, generates for each message a new number in a given range. The range as well as the interval between two messages is defined by the user.

If-Else: This node has two input topics, the primary one with messages that should be redirected and the secondary receives messages to compare the primary to. Based

⁷For details: http://www.ros.org/wiki/std_msgs, accessed July 13th 2011

on the result, the primary message is either published on the true or false output topic. The user has the option to define the comparison method out of three different possibilities: lesser than and equal, equal or greater than and equal.

For-loop: A for-loop node is suitable to publish a message only for a fixed amount of cycles. It has two input topics, the primary one receives messages that are republished in case the node is active. The second subscriber receives messages that starts the node in case it is a non-zero value. The user parametrizes the interval between two messages in seconds as well as the amount of cycles.

Conversion: To allow the combination of floats and integers, this node converts integers to floats or vice versa.

These nodes allow the user to cover almost all situations which can be solved by generic nodes using integers and floats. A ROS program using a large amount of these nodes is possible but may become too complex for the user, favoring custom nodes.

5.4.2 Strings package

The strings package provides three basic operations to alter strings that are part of the `std_msgs` package⁸ of the core ROS distribution. These are explained in the following section.

Concatenation: Two strings are combined to a single message that is published. The user can define, if a string should be used to connect them. As an example, the two messages "position" and "33,5" are combined with the connection string ":" to the message "position: 33,5".

Split: The contrary of concatenation is the split command. A string message is separated into two substrings. Therefore, the user configures the node with the split string. As an example, the message "position: 33,5" is split using ":" into the two messages "position" and "33,5". The node is limited to create two substrings and consequently splits the incoming message at the first appearance of the split string.

Substitution: Substitution allows to replace parts of a message with another string. For example, it could be used to eliminate colons or spaces. The user parametrizes the node by defining two strings: the first replaces the second string in every received message. As an example, the string "test" should be replaced by "t". Consequently, the input message "test123" will be processed and the result "t123" will be published.

Besides the above mentioned three operations, the string package also provides five nodes, one for the creation of strings, two for control flow and two for the transformation of strings into numbers and vice versa.

⁸For details: http://www.ros.org/wiki/std_msgs, accessed July 13th 2011

String: This node is a source that publishes a user defined string on its output topic. In addition, the interval between two messages can be defined in seconds.

Compare: Two input messages are compared in this node. If the strings match, the incoming message is published on the true topic.

For loop: This node has two input topics, one to start the node in case a non-zero number is received and the other and primary one to receive strings. In case the node is active, the latest message of the primary topic is published for a fixed amount of cycles. The user defines the amount of cycles as well as the interval between two messages.

String to number: To convert a string to an integer or float, this node is provided. The string message must be prepared, meaning that it may only contain numbers or a decimal point to convert it into a float.

Number to string: This node is the contrary to the above mentioned node, as it converts a number into a string. No preparation of the number message is required.

Chapter 6

Prototype

This chapter presents the prototype of the ROS-VP toolkit that implements the in Chapter 5 developed concept. Therefore I outline the prototype's software architecture. In this section I elaborate how the used Eclipse components are linked to each other, the architecture of the graphical objects as well as the internal structure that represents ROS. Next, I describe the wizard that creates the ROS-VP configuration file, the usage of the visual programming editor, the actions that are part of the action bar and context menu as well as the views that offer additional options. The final aspect of the implementation explains the two ROS packages for numbers and strings. This chapter is concluded with the evaluation of two use cases from Chapter 3: number processing for end-users and rapid prototyping of an image processing program for expert-users.

6.1 Software architecture

According to the requirements, the toolkit should be implemented using software engineering frameworks to achieve a high degree of modularity. Eclipse¹ separates its components like wizard, editor or view (see Section 5.1.1 for their explanation) and offers interfaces to connect them. I explain in this section how the components are connected to each other. The next step focuses on the graphical elements that are used inside of the visual programming editor. They are based on the Graphical Editing Framework (GEF) and apply the Model View Controller (MVC) software architecture [38]. At last, the section explains the structure I use to represent ROS in the toolkit.

6.1.1 Information exchange of user interface components

The ROS-VP toolkit uses four components (see Section 5.1.1 for their explanation) that communicate to each other: wizard, editor, action bar and view. Eclipse offers services and interfaces to link components and to allow them to exchange information [49]. The

¹Eclipse as of version 3.6.1 is supported

ROS-VP toolkit requires that information can be transferred from the wizard to the editor. In addition, the action bar and context menu must be aware of the selection inside of the editor to enable or disable custom options and to open a file dialog or a view. The views, used to show more information or to configure nodes, require access to elements inside the editor to retrieve data and to save them after modification. Figure 6.1 shows how these information are exchanged in the toolkit. A more detailed description can be found in the following paragraph.

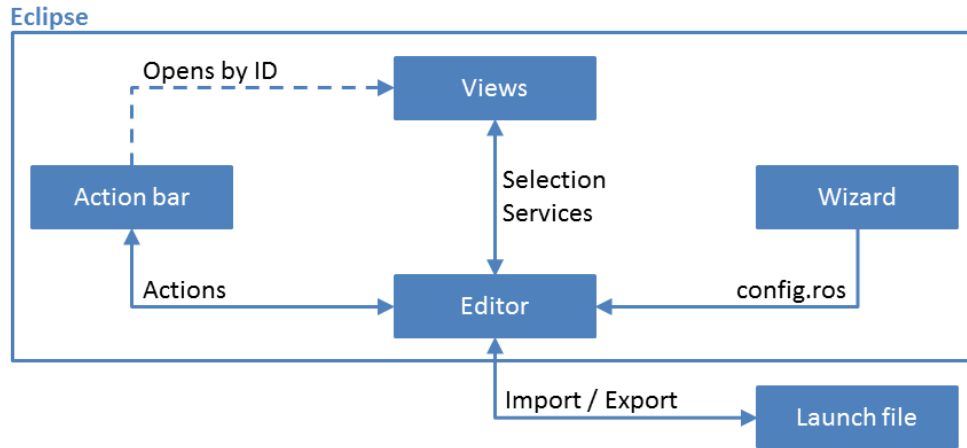


Figure 6.1: Diagram that visualize how the Eclipse components of the ROS-VP toolkit exchange information and store or retrieve the ROS program

Wizard: The user provides the wizard with the required environmental parameters for the ROS tools as well as the required packages. As Eclipse defines that an editor must be opened with a double-click on a file [49], I decided that these information are stored into a ROS-VP configuration file with the ".ros" ending. Consequently, if no parameters change, the user can reuse the already created ROS-VP configuration file. Users may create several files to have for example a configured editor for image processing and one for number processing. This file is used to transfer the information from the wizard, represented by the `ROSWizard`² class, to the editor.

Editor: In the ROS-VP toolkit, the editor allows the creation of a ROS program. Eclipse requires that an editor is associated with a file ending. This means that this editor is always opened in case this file ending is detected [49]. I selected the file ending ".ros" that is used to open the visual programming editor. Consequently, the ROS-VP toolkit editor analyzes this file and extracts the stored information like the parameters for the ROS tools or the selected packages. As the output artifact of the editor should be a launch file, the ROS program is not saved but must be exported. Saving the editor has no effect as the ROS-VP configuration file is not modified by the editor. The editor offers an interface to allow the user to import existing launch files and

²This font denotes classes of the prototype

to export the created ROS program. That way only the position of the nodes is not saved which I consider as acceptable. To allow the action bar and context menu to display which options are active, the editor registers all actions. Actions whose status depends on the selection of graphical objects, are updated by the editor each time the selection changes. The other ones are only informed when the editor mode switches. The editor is implemented by the `ROSEditor` class.

Action Bar: The action bar and the context menu allows a plug-in to offer additional functions for an editor or a view [49]. Within the ROS-VP toolkit, both components are enabled while the `ROSEditor` is active. I added all actions that are registered in the editor to the action bar and the context menu. If the parameters of an action is matched and it is active, its text becomes black, otherwise the option is grayed out. These actions either trigger a file dialog to import or export a launch file, open a source file in a new editor or call and open an Eclipse view by its ID. The action bar is implemented by the `ROSActionBarContributor` class, the context menu by the `ROSContextMenuProvider` class.

Views: Views can be used to display information in tables or trees and allow the user to modify preferences [49]. In the ROS-VP toolkit, a view is opened if its ID is called by an action or the user selects it in the Eclipse settings. As no informations can be handed over, the view uses the selection services that are provided by Eclipse. These allow the view to retrieve the selection of the active editor window. If a usable graphical object is selected, the view fetches the required information. Altered information are stored inside of the selected object in the editor. I implemented a selection listener in each view, that allows them to update themselves if the selection is changed. Views are implemented with one of the following four classes: `ConfigureNodeView`, `ConnectNodesView`, `ShowMsgsView` and `ShowTopicView` that are explained in Section 6.2.4.

6.1.2 Graphical object architecture)

Eclipse provides the Graphical Editing Framework³ that allows the creation of a visual programming editor inside of a plug-in. This framework provides abstract classes of graphical objects that I extended for the ROS-VP toolkit. The toolkit uses three graphical objects inside of the `ROSEditor`: the graph which is the primary graphical object and its two children, node and connection.

Moore et al. [38] explain that to use GEF, all graphical objects must implement a Model View Controller architecture. Each graphical object ROS-VP toolkit consists of up to three classes: model, view and controller. Gamma et al. [19] elaborate, that the model class holds all important information of the object but has no reference of the view. The

³For details: <http://www.eclipse.org/gef/>, accessed June 12th 2011

view is a "dumb" class that has no knowledge of any other classes and is responsible for the objects graphical representation. The controller synchronizes the model and the view of the object. Additionally, it handles the user's input and is called in GEF `editPart`. This means, that the Eclipse selection services always provide the `editPart` class of an selected object and not its model. To change for example the name of a node, the controller offers a method to store it inside of the model and then updates the view. Moore et al. [38] argue that the creation of a model or a controller class requires the usage of factory classes. Additionally, to create, delete or modify graphical objects, GEF offers so-called policies which are implemented into the controller and can call the respective commands that executes the task. This clear separation enables the classes to be reusable and allows an improved handling of the overall system. Figure 6.2 shows how my implemented architecture of the three graphical elements: graph, node and connection. Figure 6.3 visualizes a simplified version of the relation of graphical elements, in particular the model classes `Node` and `Connection`, with the classes representing ROS (see Section 6.1.3 for their explanation). In the following paragraph, I explain the software architecture, implemented for the three graphical objects:

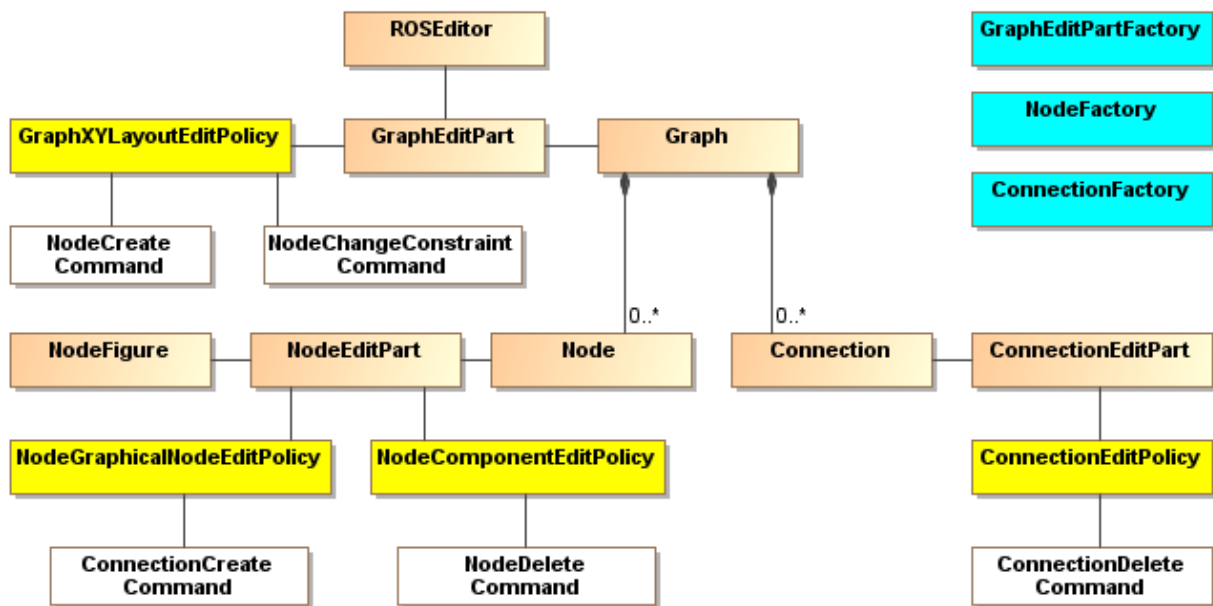


Figure 6.2: UML diagram shows a simplified version of the implemented architecture of the graphical elements. The factories that create these elements are colored cyan, the policies to edit the objects are in yellow and the commands the policies call are in white

Graph: The graph represents the active area inside the editor, in which the nodes are placed and connected. The graph is required for all nodes and connections and is the first graphical element to be instantiated. Its model holds references of all nodes and connections. The tasks of the graph are to update and create the connections between

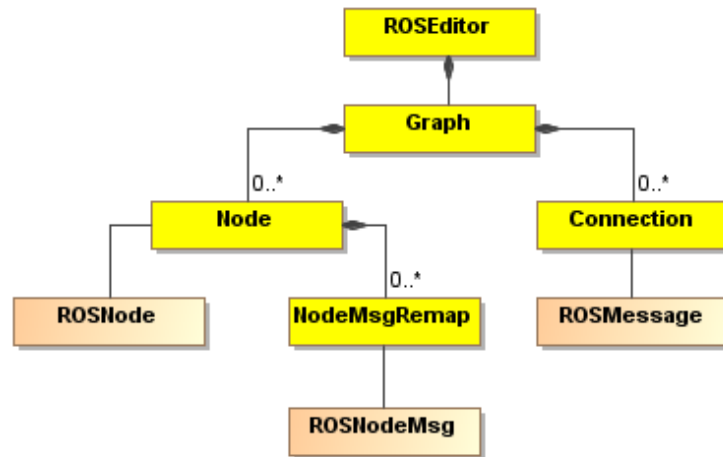


Figure 6.3: UML visualization of the linkage between the model classes of the graphical objects and the ROS classes used in the ROS-VP toolkit

the node objects, to change the mode of the editor or to create the initial unique name of a node object. The graph consists of the model (**Graph**) and the controller (**GraphEditPart**) class. The graph has no view class but implements the active area in its controller. The **GraphEditPart** implements a **GraphXYLayoutEditPolicy** that calls a command to create or change a node. There is no view class as the graph is not visible to the user.

Node: A node object is able to represent any possible ROS node and therefore requires the reference of a **ROSNode**. In addition, the node has a list of **NodeMsgRemap** objects, that reference a **ROSNodeMsgs** and contain information about the node's publishers and subscribers (see Figure 6.3). The references of the **ROSNode** or the **ROSNodeMsg** must not be modified by the node object and may only be accessed to retrieve information. All nodes of one type share a single **ROSNode** object. The view is represented by the **NodeFigure** that creates the labeled rectangle with a coloring based on the object's category. The **NodeEditPart** implements two edit-Policies: **NodeGraphicalNodeEditPolicy** calls the **ConnectionCreateCommand** while the **NodeComponentEditPolicy** allows the user to delete the selected node using the **NodeDeleteCommand**.

Connection: A connection is a directed graph between two nodes and represents a topic that both nodes use to communicate with each other. Therefore, the connection references a **ROSMessage** as well as its source and target node. The controller defines the figure as a simple directed graph that is labeled with the name of the represented topic. Additionally, the controller implements the **ConnectionEditPolicy**, which executes the **ConnectionDeleteCommand**.

6.1.3 Internal ROS representation

The internal structure to represent ROS in the ROS-VP toolkit is derived from the organization of nodes and messages into packages [41]. As a consequence, for each of these ROS objects a corresponding class is created. Moreover, I added one class to access ROS tools, one to display the ROS program at run-time, one to export and import launch files as well as one to extract the information out of the ROS-VP configuration file (visualized in Figure 6.4). The next paragraph offers a more detailed description of these classes.

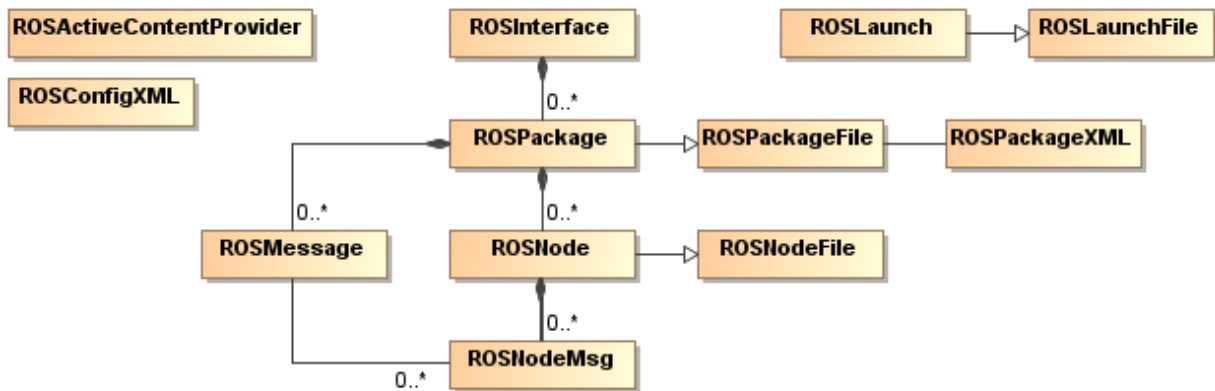


Figure 6.4: The UML diagram shows a simplified structure of the classes to represent ROS in the ROS-VP toolkit

ROSInterface: The primary function of the `ROSInterface` class is, to integrate the ROS tools (e.g. `rospack` or `roscpp`). It uses therefore a Java Runtime which is configured with environmental parameters. These information are retrieved from the ROS-VP configuration file using the `ROSConfigXML` class. Additionally, the `ROSInterface` contains a list of all `ROSPackage` objects. This class offers an interface to retrieve `ROSPackages`, either fetched from the list or by finding and analyzing a not yet indexed package on the computer.

ROSPackage: This class represents one ROS package and is created using the path to the directory. To allow the toolkit to analyze the package on the file system, this class inherits the required functions from the `ROSPackageFile`. This facilitates the retrieval of all containing nodes and messages. The `ROSPackageXML` class allows to store the information from the directory into the ROS-VP package file and to retrieve the data in case the file is found. I implemented this separation into three classes, to show which one accesses files outside of the ROS-VP toolkit and which provides an interface to other functions inside of the toolkit. The `ROSPackage` can be used to access the containing `ROSNodes` and `ROSMessages`.

ROSNode: A `ROSNode` is a child element of a `ROSPackage` and represents a single ROS node. It extends a `ROSNodeFile`, which analyzes the node's source code for publishers

and subscribers. Each `ROSNode` has a list of `ROSNodeMsg` objects, that contain the information which messages are published or subscribed. In case a ROS-VP package file is found in the package directory, these information are used to create the `ROSNode` and no source code is analyzed.

ROSMesssage: A `ROSMesssage` represents a message that is stored inside of a package and is therefore a child item of a `ROSPackage`. Required information, like the used variables, are extracted by the `ROSPackageFile` class from the message definition or ROS-VP package file.

ROSLaunch: This class allows users to export and re-import created ROS programs. Therefore, `ROSLaunch` extends the `ROSLaunchFile` class, which is utilized to handle the XML tree. For the creation of a launch file, the `ROSLaunch` receives a list of all nodes which is then analyzed and added to the XML tree. To import an existing launch file, this class generates a list of nodes that is used by the graph to create the visual representation of the ROS program.

ROSActiveContentProvide: The active mode uses this class to retrieve the running nodes and connections and then visualizes them inside the `ROSEditor`. Therefore, the `ROSInterface` retrieves these information using the `roscpp` tool.

6.2 Implementation

This section presents the implementation of the four different Eclipse components: wizard, editor, action bar and view. First, the wizard and its two-step process to create the ROS-VP configuration file is presented. Next, I outline how the visual programming is implemented concerning the user interaction into the editor. All implemented actions that are used by the action bar and context menu are explained, followed by the implemented views. At the end of the section, the two ROS packages for numbers and strings are described.

6.2.1 Configuration wizard

Before the ROS-VP toolkit editor can be utilized, the ROS-VP configuration wizard is used to generate the configuration file. To cover the required environmental parameters as well as the selection of the packages, I implemented the wizard in a two-step process (see Figure 6.5 for screenshots of both steps):

ParametersPage: The purpose of this page is to define the parameters that are applied by the ROS-VP toolkit to use the ROS tools as well as to define the location and name of the new ROS-VP configuration file. Therefore, three browse buttons allow the user to search for the matching directories. The definition of the ROS variables usually only

requires to find the ROS directory. All other parameters are automatically updated if the path is changed using the browse button. In case the user does not use the standard ROS configuration, the user can adjust each of the parameter using the text boxes.

StacksPage: The second page of the wizard allows the selection of ROS packages, which are offered in the editor's palette. Therefore, the page consists of two list. The one on the left contains all available stacks on the computer. In case the user selects one or multiple, the list on the right displays the containing packages. After the user selected his packages, the wizard can be finished. The `internal_vp_nodes` package is added automaticall and allows the user to add the Parameter Server⁴. As this package is handled internally, a possible ROS package with the same name would be ignored.

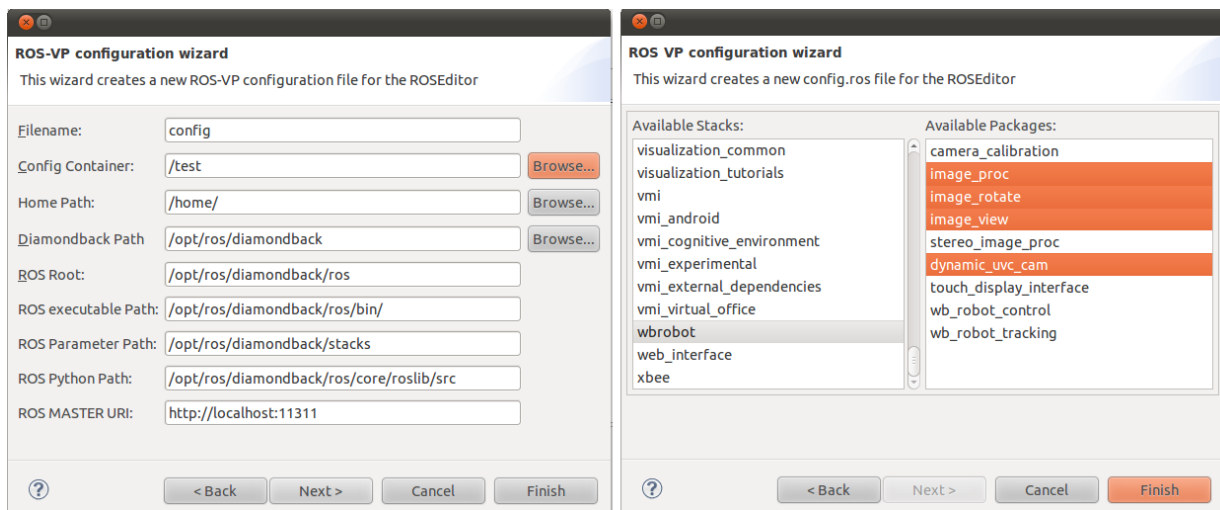


Figure 6.5: Screenshots of the implemented wizard, on the left is the `ParametersPage` and on the right is the `StacksPage`

After the finish button is pressed, the informations are stored inside the ROS-VP configuration file and the visual programming editor can be used. The ROS-VP configuration file uses the XML schema visualized in Figure 6.6. A sample configuration file is shown as follows:

```
<configuration description="ROS-VP configuration file">
  <execution>/opt/ros/diamondback/ros/bin/</execution>
  <envp>PYTHONPATH=/opt/ros/diamondback/ros/core/roslib/src</envp>
  <envp>ROS_ROOT=/opt/ros/diamondback/ros/</envp>
  <envp>ROS_MASTER_URI=http://localhost:11311</envp>
  <envp>ROS_PACKAGE_PATH=/opt/ros/diamondback/stacks</envp>
  <envp>HOME=/home/chakka</envp>
```

⁴For details: <http://www.ros.org/wiki/Parameter%20Server>, accessed July 11th 2011

```

<package>internal_vp_nodes</package>
<package>vp_numbers</package>
</configuration>

```

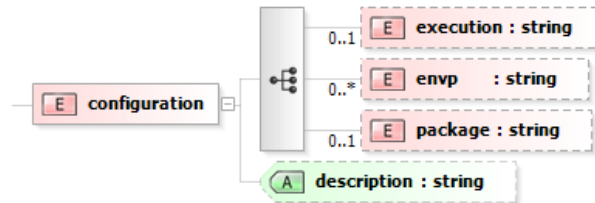


Figure 6.6: Visualization of the XML schema of the ROS-VP configuration file, the red rectangles symbolize the three used XML elements to store the required parameters

6.2.2 Visual programming editor

The ROSEditor (see Figure 6.7) implements a graphical editor with a palette on its left side. The palette is filled at the beginning of the edit mode with a selection tool and a varying amount of so-called drawers that contain nodes. A drawer is named according to the package and contains all its nodes. To gain a better overview if the user has multiple packages, the drawer can be opened or closed. Each node has its own color icon that indicates its category, defined in the ROS-VP package file (see Section 5.2.1).

The active area contains nodes and directed graphs to connect them. The connections are not drawn by the user himself, but are created by the editor. Therefore, the editor updates all connections in case a node is created or deleted. If two nodes are connected using the connect nodes view, the connections for both nodes are updated. I granted the user the opportunity to delete nodes (e.g. the ROS master node) and connections during active mode to show only the interesting parts of the running ROS program.

6.2.3 Action bar and context menu

Besides the possibility to drag and drop nodes, the action bar and the context menu of the ROS-VP toolkit offer further features. As explained in Section 6.1.1, each of these options require a separate action. For the ROS-VP toolkit, I created eight actions that are able to match the functions defined in the concept (Figure 6.8 shows the action bar with all implemented options).

ChangeModeAction: This action allows the user to change the mode of the visual programming editor from active to edit mode and vice versa. If this option is selected, the editor is cleared of its content and either the palette is filled with selected packages or the active ROS content is loaded. This option is always available to the user.

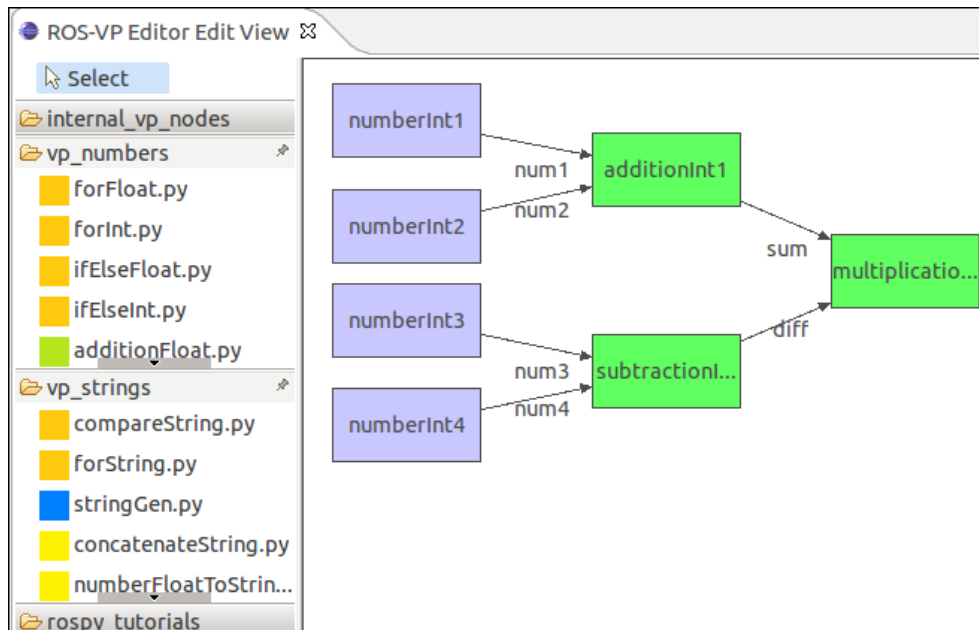


Figure 6.7: Screenshot cutout of the editor in edit mode, the vp_numbers and vp_strings package drawers are opened, all others are closed

ConnectNodesAction: In case the user wants to connect two selected nodes, this option opens the **ConnectNodeView**. This action is only available in edit mode while exactly two nodes are selected.

ConfigureNodeAction: To configure nodes and set attributes like the name, param or remap attributes, this action calls the **ConfigureNodeView**. It is only available for nodes during edit mode.

ShowMsgsAction: The show messages option allows the user to see in the **ShowMsgsView** which messages of a node are published or subscribed. It can be used to get a better overview of the whole system and is available for nodes in both modes.

ShowCodeAction: If the user wants to see or modify the source code of a node, this option opens a new editor window containing the content of the source code file. The user can simply display or modify the code and store it. It is available for any node during edit mode.

ShowTopicAction: By clicking on a connection, the user gets in the **ShowTopicView** information about the ROS message type on this topic. This action can be used in edit and active mode as long as a connection is selected.

ImportLaunchAction: This action opens a file dialog that allows the user to select a launch file during edit mode.

ExportLaunchAction: After the user finished his ROS program in edit mode, he can export

it by using this action. A file dialog is opened that requires the user to enter a filename. This action is active during edit mode.

Each element of the context menu that requires knowledge of the selection of a user is derived from a `SelectionAction`. All other actions, like the switch mode option, extend a `WorkbenchPartAction`. Each class has a `RetargetAction` class that is used to visualize the option for the action bar. In addition, action bar and context menu implement the standard options undo, redo and deletion of objects.

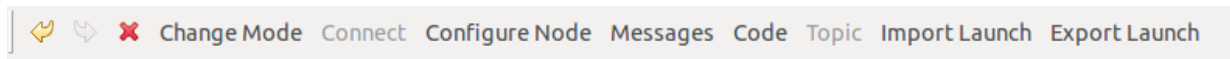


Figure 6.8: Screenshot of the action bar that implements the ROS-VP toolkit's options

6.2.4 Views of the ROS-VP toolkit

The options to configure or connect nodes and to show the used message of nodes and topics, the ROS-VP toolkit implements views as these options can not be accessed inside of the editor. I implemented the following four views that are opened by actions as described in Section 6.2.3:

ConnectNodesView: The view contains, as visualized in Figure 6.9, a list to select a pair of topics, a text box to name the new topic as well as a button to create the custom topic. The view analyzes the topics of the two selected nodes and ensures that a matching pair has the same message and consists of a publisher and a subscriber. The user can select a pair of the list to connect the two nodes. In case one of the two nodes already uses a custom topic for this message, the view suggests the same name to connect all these messages on one topic. A namespace of a node is visualized in a text box that is not modifiable by the user and is part of the topics name. It is not possible to connect two nodes that have different namespaces.

Node(1)	In/Out	Topic(1)	Node(2)	In/Out	Topic(2)	Message	Package
camera/image_proc1	Out	camera/image_mono	image_view1	In	image	Image	sensor_msgs
camera/image_proc1	Out	camera/image_color	image_view1	In	image	Image	sensor_msgs
camera/image_proc1	Out	camera/image_rect	image_view1	In	image	Image	sensor_msgs
camera/image_proc1	Out	camera/image_rect_color	image_view1	In	image	Image	sensor_msgs

Connecting both node messages on topic:

Figure 6.9: Screenshot of the connect nodes view, connecting two nodes of which one is in the namespace "camera"

ConfigureNodeView: This view has in the first row two text boxes for the name and namespace as well as two combo boxes for the respawn and output option of the selected node. In the second row, the user can add new param or remap arguments,

using the combo box and the two text boxes. All already existing remap and param arguments are part of the list (see Figure 6.10). By selecting one of the arguments the delete button is enabled and it can be removed. Remap arguments are automatically generated in case two nodes are manually connected. Additionally, the user can delete or add a remap argument to change the connections of the selected node.

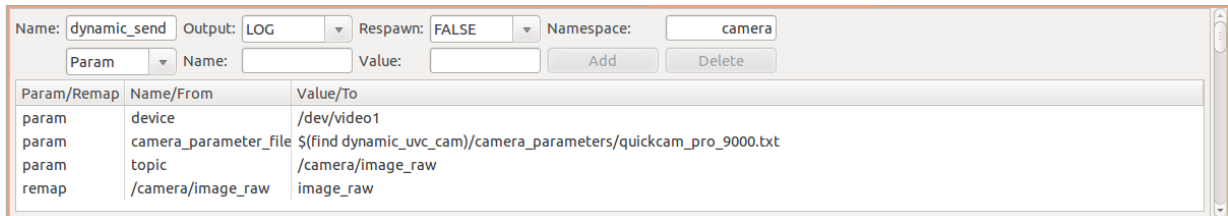


Figure 6.10: Screenshot of the configuration view, which allows to parametrize a node

ShowMsgsView: This view consists of a simple list, which contains detailed information about all messages of a node. It shows for each message whether it is published or subscribed, the name of the topic, of the message and the message's package as well as the variables that are used (see Figure 6.11).

In/out	Topic Name	Message Type	Package	Message variables
In	camera/camera/set_videomode	videomode	dynamic_uvc_cam	Header header; Header header; bool immediately
In	camera/camera/set_state	state	dynamic_uvc_cam	Header header; Header header; string state
Out	camera/camera/image_raw/compressed	CompressedImage	sensor_msgs	Header header ; Header header ; string format
Out	camera/camera/image_raw/theora	Packet	theora_image_transport	Header header ; Header header ; uint8[] data ;
Out	camera/camera/image_raw/theora/parameter_descriptions	ConfigDescription	dynamic_reconfigure	ParamDescription[] parameters; ParamDescription
Out	camera/camera/image_raw/compressed/parameter_descriptions	ConfigDescription	dynamic_reconfigure	ParamDescription[] parameters; ParamDescription
Out	camera/camera/camera_info	CameraInfo	sensor_msgs	Header header ; Header header ; uint32 height; u

Figure 6.11: Screenshot of the show messages view, showing all messages of a multiplication node

ShowTopicView: Similar to the **ShowMsgsView**, this view shows which message is used on the selected topic. It has a list with a single entry, of the message name, the package of the message and the message variables.

6.2.5 Package for numbers and strings

Based on the in Chapter 5 presented concept, I implemented the `vp_common` ROS stack that contains two packages: `vp_numbers` and `vp_strings`. These nodes are intended to cover basic operations for numbers and strings as explained in Section 5.4. Both packages are usual ROS nodes and are therefore not limited to the ROS-VP toolkit, but can be used in any possible ROS project. All nodes are created in Python and only depend on the

`std_msgs` package⁵. As there is a 32 and 64 bit version of integers and floats, the following four messages are used in the `vp_numbers` package:

Int32: Integer with a data width of 32 bit.

Int64: Integer with a data width of 64 bit.

Float32: Floating point number with a data width of 32 bit.

Float64: Floating point number with a data width of 64 bit.

Each number node has for each topic a 32 bit and 64 bit version. Internally, the 32 bit variables have no knowledge of their 64 bit versions and vice versa. For example, the `additionInt` node adds the two `Int32` messages and publishes its result only on the `Int32` output topic. The `vp_numbers` package contain nodes to convert a 32 bit value into a 64 bit value and the other way round. Consequently, the nodes from the `vp_strings` package that convert strings to numbers and vice versa apply the same logic and implement a 32 and a 64 bit version. Both packages are equipped with a ROS-VP package file that classifies all nodes.

6.3 Evaluation of two use-cases

For an evaluation of the ROS-VP toolkit prototype, two use cases from Section 3.1 are realized. The first one implements the number processing for end-users use case (see Section 3.1.2). The second scenario is derived from the rapid-prototyping for expert-users use case (see Section 3.1.4) and an image processing program is created. Consequently, to make them more recognizable, I use the same actors again.

6.3.1 Simple math program

The task of Stephanie is to create a ROS program as visualized in Figure 6.12. Therefore, the toolkit is configured to use the `vp_numbers` and `vp_strings` package out of the `vp_`-common stack as well as the `rospy_tutorial` package from the `ros_tutorial` stack. She picks integers with a 32 bit width as those are sufficient for her purpose.

For the generation of fixed numbers, she selects four times the `numberInt` node. Stephanie configures the nodes and sets the attribute "value" for `numberInt1` to 2, `numberInt2` to 4, `numberInt3` to 10 and `numberInt4` to 3. In a next step, she places a `additionInt` and a `subtractionInt` node onto the screen connecting `numberInt1` and `numberInt2` to `addition` node and the other two to `subtraction` node as show in Figure 6.13. The outputs of both nodes are then connected to the newly dragged `multiplication` node. As the user wants to log the result in form of a string, she connects the `numberIntToString` node to

⁵For details: http://www.ros.org/wiki/std_msgs, accessed July 13th 2011

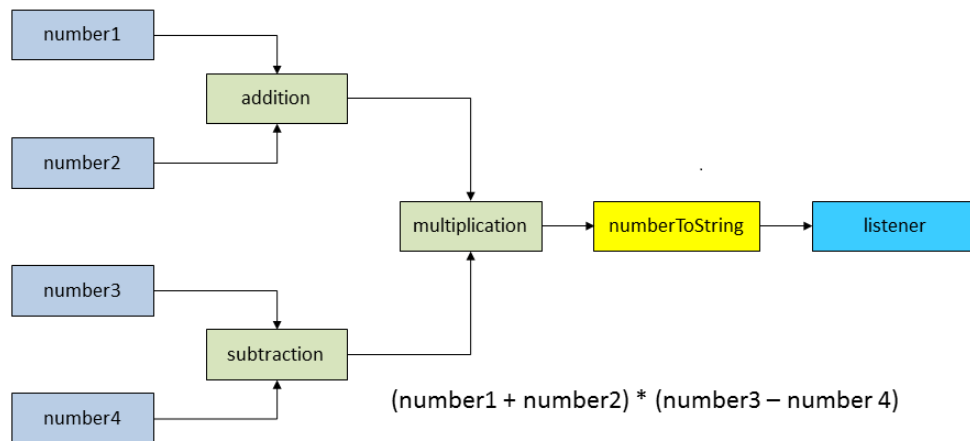


Figure 6.12: Showing a basic dataflow to solve the equation and converting the result into a string which is then logged

multiplication node. Finally, to log the result, a listener node is placed and included. The finished ROS program (see Figure 6.13) is then exported to a launch file. Stephanie starts the launch file in a command-line to check if it is properly running. As an output, the listener node logs "42" which proves that the ROS program is working.

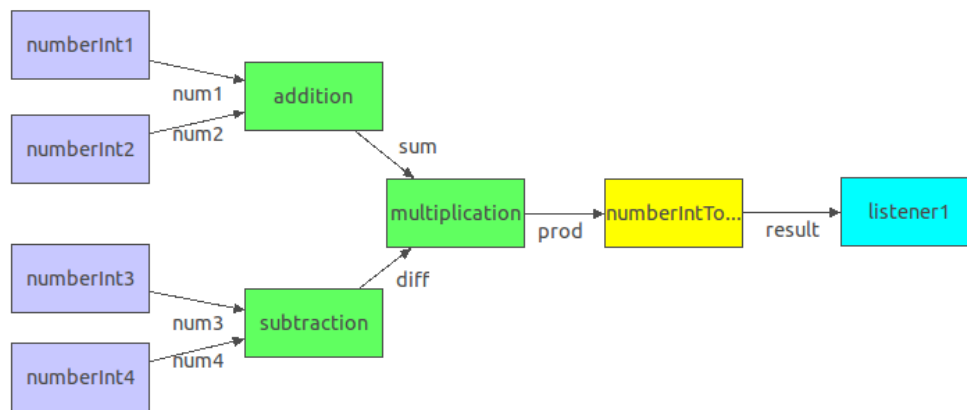


Figure 6.13: Screenshot of the created and configured ROS program that processes numbers and generates the number 42 which is then converted to a string

To extend the program, Stephanie replaces the numberInt4 node with a randomInt node. In addition, she adds an ifElseInt node after the multiplication block. In combination with a fixed numberInt node with the value 42, all messages equal to 42 are published on the true topic. Other messages are published on the false topic. Stephanie adds a node to convert the true message to a string and connect it, using a stringGen and a concatenate node, to the resulting string of "Value: 42" as seen in Figure 6.14.

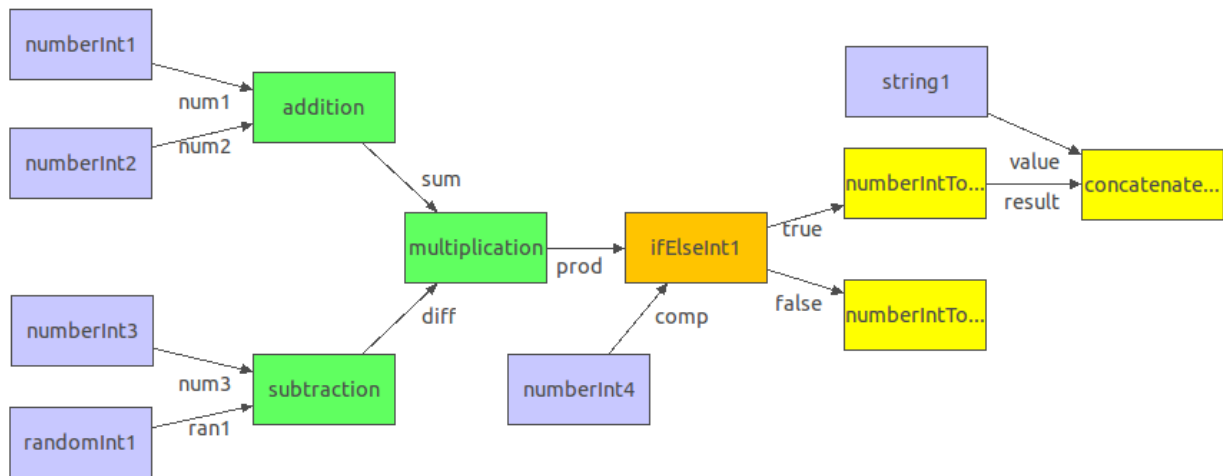


Figure 6.14: Screenshot of the created and configured ROS program that processes a random numbers and controls the dataflow based on the result of the multiplication

6.3.2 Image processing program

To visualize a more complex use case, Florian (see Section 3.1.4) creates an image processing program. To get accustomed to the available nodes, he wants to capture the video of a webcam, process it and view a monochrome, a normal and a zoomed image stream. To accomplish this, the user selects the `dynamic_uvc_sender` package from the `wbrobot` stack for capturing the video signal as well as the `image_proc`, `image_rotate` and `image_view` packages from the `image_pipeline` stack. As the toolkit is not able to retrieve the information about the subscribers and publishers, Florian already modified the ROS-VP package file of all four packages and added the missing data. He drags the `dynamic_sender` and configures it with the required parameters: the device value, camera parameters, used topic, width and height of the image as well as the count of frames per second. This node is then linked to an `image_proc` node which generates, a colored and a monochrome image. The `image_rotate` node uses the colored image and zooms it. In the end, an `image_view` shows the monochrome, another one the colored and a third one the zoomed colored image. Florian exports his ROS program (see Figure 6.15) and starts it. Immediately 3 windows are opened and he only have to send the start signal to the `dynamic_sender` node and the three videos streams appear.

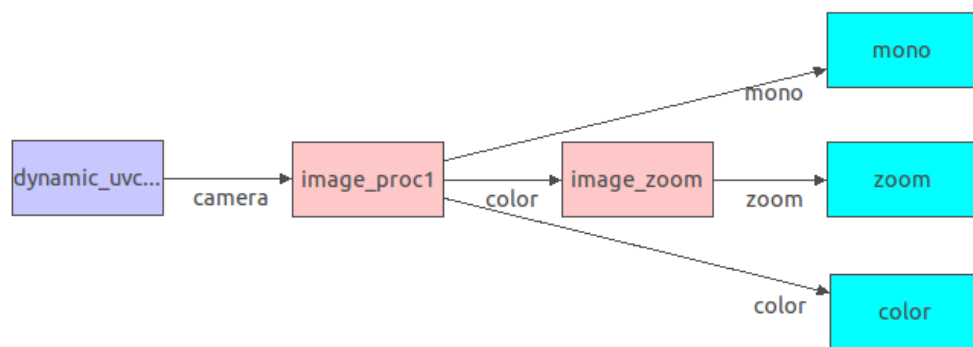


Figure 6.15: Screenshot of the created and configured ROS image processing program

Chapter 7

Conclusion

This thesis assumes that ROS, a middleware for distributed sensor-actuator systems, is currently not attractive for end-users. In particular, it provides no end-user tool support to create and configure a ROS program. This issue is tackled with the development and implementation of the ROS-VP toolkit that enables End-User Programming. The scope of the ROS-VP toolkit is defined by the requirements. They are derived in Chapter 3 from four use-cases as well as general user needs for user innovation and user needs for the integration into existing development processes. The state of the art (see Chapter 4) analysis shows that Simulink and Microsoft's Robotics Developer Studio are based, like ROS, on dataflow programming and both target professionals. Consequently, the visual programming concepts of both applications are applied as a basis for the ROS-VP toolkit concept. The concept integrates the toolkit into Eclipse and uses visual programming to show ROS nodes and topics. An edit mode is offered that allows users to create their ROS program by configuring and connecting nodes. The active mode visualizes the ROS program at run-time in the editor. In addition, the concept provides basic operations and control flow for strings and numbers. In Chapter 6, the ROS-VP toolkit prototype is described. At first, the developed software architecture is outlined, which explains how the ROS-VP toolkit links the different Eclipse components, how the visual programming objects are structured and how the toolkit represents ROS internally. The implementation of the prototype shows the two-step process of the wizard and the visual programming editor. In addition, the actions of the action bar and context menu are listed and the used Eclipse views are described. As a next step, the thesis presents the created ROS stack, which contains one package to process numbers and one to modify strings. Finally, the toolkit is evaluated by testing two of the four use-cases from Section 3.1. It is shown, that the number processing program as well as the image processing program can be implemented using the prototype, proving that it is suitable for end-users as well as for expert-users.

To conclude this thesis, I discuss the results of the implemented prototype and its concept as well as reflect to which extent the ROS-VP toolkit is able to cover the described requirements from Section 3.3. The limitations of the toolkit will be used to show future

steps the toolkit may pursue.

7.1 Discussion

The discussion of the developed ROS-VP toolkit is solely based on my own experiences gathered during the development and testing of the prototype as well as the evaluation of two defined use cases. As a first step, the advantages of the toolkit are shown comparing it with the existing process before the issues are described:

Proof of concept: First of all, the deployed prototype proves that it is in fact possible to implement a visual programming editor to create a ROS program. Two sample use-cases were realized, with the limitation that the image processing program required the modification of the ROS-VP package files. This means, that end-users can build and extend their ROS programs by dragging nodes and connecting them to each other without being exposed to the XML syntax.

Limited knowledge: End-users usually have no or only limited knowledge about the syntax of the launch file or how to program in general. The toolkit is able to solve this issue, by supporting the configuration of nodes. A complex task like connecting nodes does not require an in-depth understanding of the source code. The user can simply select and connect two nodes. As a consequence, users apply the core principles of ROS without the drawback of learning the syntax. The generated launch file reduces the chances of errors as typos and syntax errors in the XML tree are not likely.

Automatic analysis: As existing ROS packages can be included, the prototype's automatic analysis of the source code is able to extract the information about publishers and subscribers. In my opinion, this is an important feature as the user no longer has to investigate the source code on his own. Especially the extraction of ROS messages works smoothly and does not require any modifications by the user.

Support for developers: The developer may use complex message handling, like a custom camera publisher that the ROS-VP toolkit is not able to recognize it. As the effort to cover any possible variation of subscriber and publishers is high, the ROS-VP package file allows the manual entering of these information. Consequently, the toolkit is still able to remap the topics if the proper information are provided. The ROS-VP package file can also be used to include files that are not created in C++ or Python as these nodes must no longer be analyzed. If it could be achieved that developers of packages add and share these files, the user could have the full support of the ROS-VP toolkit for all nodes, without having any effort on his own.

Unification of applications: ROS users are, by using the ROS-VP toolkit, able to do all their required steps in a single application and a terminal. Eclipse supports the creation of ROS nodes in programming languages like Java, Python or C++.

The ROS-VP toolkit allows users to code their program inside of Eclipse and to have a look onto the running system. They only have to use the command-line to create and compile packages or to start a launch file. This reduces the need for additional applications or tools and enables Eclipse to become a universal ROS developer environment.

Besides these advantages, the implemented prototype has still shortcomings that should be addressed to allow end-users to build complex robotic systems or Intelligent Environments.

Limited functionalities: The prototype simplifies the creation of launch files and does not offer all potentially feasible options. This is necessary due to the time constraints of this thesis of six months and to keep the user interface simple. The integration of all possible options that the launch offers will create a very complex user interface that is not suitable for end-users.

No automatic layout: The active mode automatically detects the running nodes, places them in rows and connects them to each other. More complex ROS programs may confuse the user as the nodes are not placed in a way the dataflow would propose. The user can place nodes by himself, but an automatic layout algorithm would make the active mode more useful.

Limited run-time support: During run-time, the toolkit only shows the running nodes and the connecting topics. ROS includes a tool¹ to inspect an active ROS program that offers more functionalities and is therefore at the moment the tool of choice to analyze a running ROS program for experienced users.

Limited visualization: The toolkit offers limited support for a convenient visualization of the nodes or the options in the context menu. To let the toolkit's action bar blend into the existing tool bar, icons are required. In addition, it might be useful to add icons or small graphics to the nodes, as the coloring is not self-explanatory.

To put it in a nutshell, the ROS-VP toolkit was able to match all requirements that were prioritized with a Must, a Should or a Could (see Section 3.3) and is therefore considered as a success. Regarding the functionalities, a trade-off remains between the simplification of the creation process and supporting all possible functions.

7.2 Future work

The thesis is embedded into the research on Intelligent Environments and utilizing ROS as a middleware for distributed sensor-actuator systems at the Distributed Multimodal Information Processing Group of the Technische Universität München. This thesis should be seen as a first attempt to make ROS more attractive to end-users and to support existing

¹For details: <http://www.ros.org/wiki/rxgraph>, accessed June 17th 2011

users in their development. As the discussion in the previous paragraph showed various limitations, this section provides ideas to improve and advance the toolkit.

Improved visualization: The visualization of the options in the action bar, of the nodes or the icons is rudimentary. To make it more appealing for new users, a designer should be included in the project.

Advanced code analysis: The current analysis of the source code is limited to two programming languages and messages. In the next iteration, the toolkit should be able to analyze the source code for required parameters and the services the nodes utilize. These arguments would then be used for a more simplistic configuration of nodes and could be integrated into the ROS-VP package file.

Field-study: As the toolkit was not yet tested by end-users, a field study should be executed to integrate their feedback. A possible implementation could be to include the prototype into a university course about Intelligent Environments and the students are asked to create ROS programs as assignment. They should be asked to create the launch file by themselves, once with a simple XML editor and once using the ROS-VP toolkit. A survey and user interviews would reveal how suitable visual programming is and what features users miss and should be implemented.

Automatic node suggestion: As end-users have no knowledge of different ROS packages and what each node can accomplish, the toolkit should have a library of all possible nodes. This library should contain information about the purpose of each node, its messages and its parameters. As some packages provide very detailed description in the wiki², these information could also be integrated. As some of these details are already stored in the ROS-VP package file, additional arguments could be added. A help view, integrated into the toolkit would allow to search for required nodes. For example, a user needs a node that accepts image messages and can rotate them. The help view would show him that the `image_pipeline` package contains the `image_rotate` node and lets him add this package to the palette.

Run-time support: Debugging the whole ROS system is at the moment not very comfortable. Detailed analysis of a ROS program, like listening to topics, requires the usage of command-line tools. The active mode could be extended to a run-time analysis suite, offering the user options like listening to topics and displaying the results. As a first step, the user should have the opportunity to run existing ROS programs from the toolkit. Next, the toolkit should offer scopes for the standard messages like numbers and string that allow users to see the activities on the topic. In a future revision, features like dynamic configuration of node parameters and replacing running nodes would transform the toolkit to a full-fledged ROS run-time suite.

²For example: http://www.ros.org/wiki/image_rotate, accessed July 14th 2011

List of Figures

2.1	A collection of tangible programming parts Horn and Jacob [24] used in their concept to control a robot. These objects can be connected, photographed and converted into a program	14
3.1	Showing a basic dataflow (for details on the coloring scheme see Section 5.2.1) to solve the equation and logging the converted results	22
4.1	The user interface of the NXT-G programming environment that is part of LEGO Mindstorms NXT. The visualized sequence shows a program, that moves forward until an obstacle appears, breaks and then activates the second actuator	31
4.2	A simple program created in Microsoft Robotics Developer Studio that checks the value of a number. If the value of the variable "test" is equal to 1, the word "Finished" is spoken by the computer. In any other case, the number is decremented and its value is said by the computer	32
4.3	The visual editor of Simulink shows a program to convert the measured angular value to corrected one. Therefore the program adds offsets and scales the raw value	33
4.4	App Inventor for Android block that enables or disables a ball and changes the color and text of the pressed button	35
4.5	The Starlogo TNG visual editor shows how pressing the space bar or the "a"-key is handled. Using the space bar moves the agent and let it say "abc", a keystroke on the "a"-key changes the shape of the 3D agent to a fish	36
4.6	A block from Scratch that is executed when the green flag button is pressed. The loop runs forever and changes the position of an object until the y-position is too low and the game is stopped [43]	37
5.1	A screenshot of the user interface of Eclipse shows the package explorer and three extension points the ROS-VP toolkit uses: editor, actionbar and view	43
5.2	Two ROS nodes are connected inside of the ROS editor over the sum topic	43
5.3	Visualization of the XML schema used for the ROS-VP package file. Red objects with an "E" represent XML elements like nodes or messages. Green objects with an "A" are attributes like the category and type of a node	46
6.1	Diagram that visualize how the Eclipse components of the ROS-VP toolkit exchange information and store or retrieve the ROS program	53

6.2	UML diagram shows a simplified version of the implemented architecture of the graphical elements. The factories that create these elements are colored cyan, the policies to edit the objects are in yellow and the commands the policies call are in white	55
6.3	UML visualization of the linkage between the model classes of the graphical objects and the ROS classes used in the ROS-VP toolkit	56
6.4	The UML diagram shows a simplified structure of the classes to represent ROS in the ROS-VP toolkit	57
6.5	Screenshots of the implemented wizard, on the left is the ParametersPage and on the right is the StacksPage	59
6.6	Visualization of the XML schema of the ROS-VP configuration file, the red rectangles symbolize the three used XML elements to store the required parameters	60
6.7	Screenshot cutout of the editor in edit mode, the <code>vp_numbers</code> and <code>vp_strings</code> package drawers are opened, all others are closed	61
6.8	Screenshot of the action bar that implements the ROS-VP toolkit's options	62
6.9	Screenshot of the connect nodes view, connecting two nodes of which one is in the namespace "camera"	62
6.10	Screenshot of the configuration view, which allows to parametrize a node .	63
6.11	Screenshot of the show messages view, showing all messages of a multiplication node	63
6.12	Showing a basic dataflow to solve the equation and converting the result into a string which is then logged	65
6.13	Screenshot of the created and configured ROS program that processes numbers and generates the number 42 which is then converted to a string . . .	65
6.14	Screenshot of the created and configured ROS program that processes a random numbers and controls the dataflow based on the result of the multiplication	66
6.15	Screenshot of the created and configured ROS image processing program .	67

List of Tables

3.1	Functional requirements for the ROS-VP toolkit	28
3.2	Nonfunctional requirements for the ROS-VP toolkit	29
4.1	Summary of visual programming applications	39

Nomenclature

AIA	App Inventor for Android
API	Application Programming Interface
CAD	Computer Aided Design
EUP	End-User Programming
GEF	Graphical Editing Framework
IDE	Integrated Development Environment
MoSCoW	Must, Should, Could, Won't
MVC	Model View Controller
NXT	LEGO Mindstorms NXT
OpenCV	Open Computer Vision
PDF	Portable Document Format
RDS	Microsoft Robotics Developer Studio
RFID	Radio Frequency Identification
ROS	Robot Operating System
UML	Unified Modeling Language
VP	Visual Programming
WLAN	Wireless Local Area Network
XML	Extensible Markup Language
XMLRPC	Extensible Markup Language Remote Procedure Call

Appendix A

Source code

Bibliography

- [1] E. Aitenbichler, J. Kangasharju, and M. Muhlhauser. Mundocore: A light-weight infrastructure for pervasive computing. *Pervasive and Mobile Computing*, 3(4):332–361, 2007.
- [2] A. Angermann. *Matlab-Simulink-Stateflow: Grundlagen, Toolboxen, Beispiele*. Oldenbourg Wissenschaftsverlag, 2007.
- [3] R. Assisi. *Eclipse 3 Einführung und Referenz*. Carl Hanser Verlag, München Wien, 2005.
- [4] N. Baker, M. Zafar, B. Moltchanov, and M. Knappmeyer. Context-aware systems and implications for future internet. *Towards the Future Internet*, page 335, 2009.
- [5] A. Begel and E. Klopfer. Starlogo tng: An introduction to game development. *Journal of E-Learning*, 2007.
- [6] N. Bencomo, P. Sawyer, P. Grace, and G. Blair. Ubiquitous computing: Adaptability requirements supported by middleware platforms. In *Workshop on Software Engineering Challenges for Ubiquitous Computing*. Citeseer, 2006.
- [7] G. Borriello. Invisible computing: automatically using the many bits of data we create. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 366(1881):3669, 2008.
- [8] G. Bradski and A. Kaehler. *Learning OpenCV: Computer vision with the OpenCV library*. O’Reilly Media, 2008.
- [9] M. Branscombe. Android@home: what you need to know, May 2011. URL <http://www.techradar.com/news/digital-home/android-home-what-you-need-to-know-955045>.
- [10] D. Clegg and R. Barker. *Case Method Fast-Track: A Rad Approach*. Addison-Wesley, 1994.
- [11] A. Cockburn. *Writing effective use cases*, volume 1. Addison-Wesley, 2001.
- [12] M.H. Coen et al. Design principles for intelligent environments. In *Proceedings of the National Conference on Artificial Intelligence*, pages 547–554. JOHN WILEY & SONS LTD, 1998.

- [13] F. Dabek, N. Zeldovich, F. Kaashoek, D. Mazieres, and R. Morris. Event-driven programming for robust software. In *Proceedings of the 10th workshop on ACM SIGOPS European workshop*, pages 186–189. ACM, 2002.
- [14] J. des Rivieres and J. Wiegand. Eclipse: A platform for integrating development tools. *IBM Systems Journal*, 43(2):371–383, 2004.
- [15] K. Ehrig, C. Ermel, S. Hänsen, and G. Taentzer. Generation of visual editors as eclipse plug-ins. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 134–143. ACM, 2005.
- [16] Michael Eisenberg. End user programming. *Handbook of Human Computer Interaction, second, completely revised edition*. North-Holland, pages 1127–1146, 1997.
- [17] P. Elmer-DeWitt. Apple users buying 61% more apps, paying 14% more per app, July 2011. URL <http://tech.fortune.cnn.com/2011/07/11/apple-users-buying-61-more-apps-paying-14-more-per-app/>.
- [18] A. Fox, R. Griffith, AD Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. Above the clouds: A berkeley view of cloud computing. *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS*, 28, 2009.
- [19] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*, volume 206. Addison-Wesley, 1995.
- [20] W. Grega and A. Pilat. Real-time control teaching using lego® mindstorms® nxt robot. In *Computer Science and Information Technology, 2008. IMCSIT 2008. International Multiconference on*, pages 625–628. IEEE, 2008.
- [21] D.C. Halbert. Programming by example. *University of California, Berkeley*, page 121, 1984.
- [22] C. Hartmann. Microsoft visual programming language: End-user perspective. 2008.
- [23] D. D. Hils. Visual languages and computing survey: Data flow visual programming languages. *Journal of Visual Languages & Computing*, 3(1):69 – 101, 1992. ISSN 1045-926X.
- [24] M. S. Horn and R. J. K. Jacob. Designing tangible programming languages for classroom use. *Proceedings of the 1st international conference on Tangible and embedded interaction - TEI '07*, page 159, 2007.
- [25] H. Ishii. Tangible bits: beyond pixels. In *Proceedings of the 2nd international conference on Tangible and embedded interaction*, pages xv–xxv. ACM, 2008.
- [26] H. Ishii and B. Ullmer. Tangible bits: towards seamless interfaces between people, bits and atoms. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 234–241. ACM, 1997.

- [27] J. Jackson. Microsoft robotics studio: A technical introduction. *Robotics & Automation Magazine, IEEE*, 14(4):82–87, 2007.
- [28] G. Juve, E. Deelman, K. Vahi, G. Mehta, B. Berriman, B.P. Berman, and P. Maechling. Scientific workflow applications on amazon ec2. In *E-Science Workshops, 2009 5th IEEE International Conference on*, pages 59–66. Ieee, 2009.
- [29] C. Kidd, R. Orr, G. Abowd, C. Atkeson, I. Essa, B. MacIntyre, E. Mynatt, T. Starner, and W. Newstetter. The aware home: A living laboratory for ubiquitous computing research. *Cooperative Buildings. Integrating Information, Organizations and Architecture*, pages 191–198, 1999.
- [30] S.H. Kim and J.W. Jeon. Programming lego mindstorms nxt with visual programming. In *Control, Automation and Systems, 2007. ICCAS'07. International Conference on*, pages 2468–2472. Ieee, 2007.
- [31] E. Knorr and G. Gruman. What cloud computing really means. *InfoWorld*, 7, 2008.
- [32] M. Kranz and A. Schmidt. Prototyping smart objects for ubiquitous computing. In *Workshop on Smart Object Systems, 7th International Conference on Ubiquitous Computing (Ubicomp)*. Citeseer, 2005.
- [33] M. Kranz, A. Schmidt, and P. Holleis. Embedded interaction: Interacting with the internet of things. *IEEE Internet Computing*, 14(2):46 – 53, March-April 2010. ISSN 1089-7801. doi: 10.1109/MIC.2009.141.
- [34] H. Lieberman, F. Paterno, M. Klann, and V. Wulf. End-user development: An emerging paradigm. *End User Development*, pages 1–8, 2006.
- [35] T. Linner, M. Kranz, L. Roalter, and T. Bock. Compacted and industrially customizable ambient intelligent service units: Typology, examples and performance. In *Proceedings of the 2010 Sixth International Conference on Intelligent Environments*, IE '10, pages 295–300, Washington, DC, USA, July 2010. IEEE Computer Society. ISBN 978-0-7695-4149-5.
- [36] T. Linner, M. Kranz, L. Roalter, and T. Bock. Robotic and ubiquitous technologies for welfare habitat. In *Journal of Habitat Engineering*, volume 3, pages 101–110, March 2011.
- [37] B. Magnuson. *Building Blocks for Mobile Games: A Multiplayer Framework for App Inventor for Android*. PhD thesis, Massachusetts Institute of Technology, 2010.
- [38] B. Moore, International Business Machines Corporation. International Technical Support Organization, and Inc ebrary. *Eclipse development using the graphical editing framework and the eclipse modeling framework*. IBM, International Technical Support Organization, 2004.
- [39] B. A. Myers, A. J. Ko, and M. M. Burnett. Invited research overview: end-user

- programming, 2006. URL <http://portal.acm.org/citation.cfm?id=1125451.1125472>.
- [40] B.A. Myers. Taxonomies of visual programming and program visualization*. *Journal of Visual Languages & Computing*, 1(1):97–123, 1990.
 - [41] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng. ROS : an open-source Robot Operating System. In *ICRA Workshop on Open Source Software*, 2009.
 - [42] B. Z. Research. 5th annual eclipse adoption study. November 2008.
 - [43] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, et al. Scratch: programming for all. *Communications of the ACM*, 52(11):60–67, 2009.
 - [44] L. Roalter, M. Kranz, and A. Möller. A middleware for intelligent environments and the internet of things. In Zhiwen Yu, Ramiro Liscano, Guanling Chen, Daqing Zhang, and Xingshe Zhou, editors, *Ubiquitous Intelligence and Computing*, volume 6406 of *Lecture Notes in Computer Science*, pages 267–281. Springer Berlin / Heidelberg, 2010.
 - [45] R.V. Roque. Openblocks: an extendable framework for graphical block programming systems. Master’s thesis, Massachusetts Institute of Technology, 2007.
 - [46] C. Scaffidi, M. Shaw, and B. Myers. Estimating the numbers of end users and end user programmers. In *Visual Languages and Human-Centric Computing, 2005 IEEE Symposium on*, pages 207–214. IEEE, 2005.
 - [47] A. Schmid. A concept for a community-based user interface design process. Master’s thesis, Technische Universität München, 2008.
 - [48] R.W. Sebesta. *Concepts of programming languages*. Addison-Wesley New York, 1996.
 - [49] S. Shavor. *Eclipse: Anwendungen und Plug-Ins mit Java entwickeln*. Addison-Wesley, 2004.
 - [50] B. Shneiderman. Direct manipulation: a step beyond programming languages. *Sparks of Innovation in Human-Computer Interaction*, 1993.
 - [51] A.C. Smith. Using magnets in physical blocks that behave as programming objects. In *Proceedings of the 1st international conference on Tangible and embedded interaction*, pages 147–150. ACM, 2007.
 - [52] A.C. Smith. Tangible interfaces for tangible robots. 2010.
 - [53] I. Sommerville. *Software Engineering (7th Edition)*. Pearson Addison Wesley, 2004. ISBN 0321210263.

- [54] E. Spertus, M. L. Chang, and D. Wolber. Novel Approaches to CS 0 with App Inventor for Android. *Learning*, pages 325–326, 2010.
- [55] P. Szekely. User interface prototyping: Tools and techniques. In *Software Engineering and Human-Computer Interaction*, pages 76–92. Springer, 1995.
- [56] Y.P. Tsou, J.W. Hsieh, C.T. Lin, and C.Y. Chen. Building a remote supervisory control network system for smart home applications. In *Systems, Man and Cybernetics, 2006. SMC'06. IEEE International Conference on*, volume 3, pages 1826–1830. IEEE, 2006.
- [57] I. Utting, S. Cooper, M. Kölling, J. Maloney, and M. Resnick. Alice, greenfoot, and scratch—a discussion. *ACM Transactions on Computing Education (TOCE)*, 10(4):17, 2010.
- [58] E. Von Hippel. Learning from open-source software. *MIT Sloan management review*, 42(4):82–86, 2001.
- [59] E. Von Hippel. Perspective: User toolkits for innovation. *Journal of Product Innovation Management*, 18(4):247–257, 2001.
- [60] E. Von Hippel. Democratizing innovation: The evolving phenomenon of user innovation. *Journal für Betriebswirtschaft*, 55(1):63–78, 2005.
- [61] E. Von Hippel and R. Katz. Shifting innovation to users via toolkits. *Management science*, 48(7):821–833, 2002. ISSN 0025-1909.
- [62] K. Wang, C. McCaffrey, D. Wendel, and E. Klopfer. 3d game design with programming blocks in starlogo tng. In *Proceedings of the 7th international conference on Learning sciences*, pages 1008–1009. International Society of the Learning Sciences, 2006.
- [63] M. Weiser. The computer for the 21st century. *Scientific American*, 265(3):94–104, 1991.
- [64] M. Weiser. Ubiquitous computing. *Computer*, 26(10):71–72, 1993.
- [65] M. Weiser, R. Gold, and J.S. Brown. The origins of ubiquitous computing research at parc in the late 1980s. *IBM systems journal*, 38(4):693–696, 1999.
- [66] P. Wyeth and H.C. Purchase. Tangible programming elements for young children. In *CHI'02 extended abstracts on Human factors in computing systems*, pages 774–775. ACM, 2002.