USING ROS FOR AGRICULTURAL ROBOTICS - DESIGN CONSIDERATIONS AND EXPERIENCES

Ruud BARTH[1], Jörg BAUR[2], Thomas BUSCHMANN[2], Yael EDAN[3], Thomas HELLSTRÖM[4], Thanh NGUYEN[6], Ola RINGDAHL[4], Wouter SAEYS[6], Carlota SALINAS[5], Efi VITZRABIN[3]

[1]Greenhouse Horticulture, Wageningen University & Research Center, Droevendaalsesteeg 107 Wageningen, 6708 PB, the Netherlands

[2]Institute of Applied Mechanics, Technische Universität München, Boltzmannstr. 15, 85748 Garching, Germany

[3]Department of Industrial Engineering and Management, Ben-Gurion University of the Negev, Beer Sheva, Israel

[4]Department of Computing Science, Umeå University, SE-901 87 Umeå, Sweden

[5]Centre for Automation and Robotics UPM-CSIC, Crta. Campo Real 0,2 Km, 28500 Arganda del Rey, Spain

[6]KU Leuven, Department of Biosystems, Kasteelpark Arenberg 30, Leuven, Belgium

Abstract. We report on experiences of using the ROS middleware for development of agricultural robots. We describe software related design considerations for all main components in developed subsystems as well as drawbacks and advantages with the chosen approaches. This work was partly funded by the European Commission (CROPS GA no 246252).

Keywords: Agriculture, Robotics, ROS, Software development

1 Introduction

Modern robotics emphasizes how physical appearance, or morphology, is tightly connected to truly intelligent behaviour (Pfeifer, Scheier 2001). Behaviour is viewed as the result of sensors controlling actuators in an inseparable sensory-motor loop mediated by the environment, under the control of an appropriate software system (Billing et al. 2011). While this view emphasizes the importance of hardware, it also indicates that flexibility and goal-directed intelligence   primarily depends on the piece of software that connects sensing with acting. Software is indeed what "blows life" into otherwise dead robot hardware. Compared to hardware, software has the great advantages of being small, lightweight, adaptable, and multipurpose. Furthermore, software components often have complexity, degrees of freedom, and dimensionality that by far exceed the sensors and actuators they are interfaced to.

Agricultural robots typically require software in several parts of the robots, and for different purposes. Many sensors, such as laser scanners and cameras, often contain embedded computers for data acquisition, data processing and communication. Further data processing is typically performed in a separate computer running a dedicated analysis program. Robot arms and grippers often contain low-level controllers for control of joint angles. A dedicated top-level computer typically

contains driver routines for communication with sensors and actuators, and algorithms for sensing, motion planning, motion control, gripping, and mission control. Several generic tools have been proposed to support the development of robot software. So called Robotic Development Environments (RDE), such as ROS (Quigley et al. 2009), MRDS (Jackson 2008), NAV2000 (Hellström et al. 2008), and Orca (Makarenko et al. 2006) aim at simplifying interfacing to hardware and communication between software components in a system. The ROS system provides support for interfacing with sensors and actuators, communication between software components, and also high-level modules for navigation and path planning. ROS has become very popular in the last years, not least for academic research and development. The main contribution of this paper is an analysis of design considerations related to using ROS for development of, in particular agricultural, robots. The work is based on experiences from software development within the CROPS project (www.crops-robots.eu/), which aims at developing a number of robots for agricultural and forestry use. The presented analysis and experiences may serve as useful guidelines in similar development projects, both for deciding on appropriate software tools, and for the actual software design work. Section 2 gives some background to the CROPS project and the ROS system. In Section 3, software related design considerations for all main subsystems in the developed CROPS robots are described. Drawbacks and advantages with chosen approaches are identified and compared to alternative approaches. In Section 4, experiences from the collaborative development work are reported and discussed. Section 5 concludes the paper with a summary and final discussion.

2 Background

2.1 The CROPS robots

Currently there is a high demand to automate labour in modern greenhouses, orchards, plantations and forests. The availability of a skilled workforce that accepts repetitive tasks is decreasing rapidly. Furthermore, the climate conditions of the working environment in greenhouses are harsh. The resulting increase in labour costs and reduced capacity puts pressure on the economic viability of the greenhouse sector. A major objective in the CROPS project is to develop scientific know-how for a highly configurable, modular and clever carrier platform that includes a modular manipulator and intelligent tools (sensors, algorithms, sprayers, grippers) that can be easily installed onto the carrier and are capable of adapting to new tasks and conditions. Several technological demonstrators are developed for high value crops like greenhouse vegetables, fruits in orchards, and grapes for premium wines. The robotic platform will be capable of site-specific spraying (targets spray only towards foliage and selective targets) and selective harvesting of fruit (detects the fruit, determines its ripeness, moves towards the fruit, grasps it and softly detaches it). Another objective in CROPS is to develop techniques for reliable detection and classification of obstacles and other objects to enable successful autonomous navigation and operation in plantations and forests. The agricultural and forestry applications share many research areas, primarily regarding sensing and learning capabilities. The description in this paper will focus on the development of the sweet-pepper harvesting robot and the apple-harvesting robot in CROPS.

2.2 ROS

In the CROPS project, most software development has been done using the ROS (Robot Operating System) environment (Quigley et al. 2009), with programming done in C++. ROS manages parallel execution of software modules, denoted *nodes*, and administrates Ethernet based communication between nodes (message passing). ROS also supports transparent physical relocation of nodes such that computer intense program modules easily can be moved to a separate computer. This is particularly useful during system development since several research groups can easily connect their respective computers to a working system.

The concept *software components* (Szyperski 2002) is based on the idea that software should be developed by gluing prefabricated components together. Each component should be stand-alone with a well-defined interface, and not depend on others to work. An example is two components for image analysis that can be interchanged without modifying any other code. Our implementation work has been guided by this design philosophy, which is also supported by the three major communication methods in ROS. The most common method is denoted *publish-subscribe* or *event-driven communication*. Nodes that are interested in data *subscribe* to the relevant *topic*, and nodes that generate data *publish* to the relevant topic. With this method, nodes are not aware of whom they are communicating with, and data generation and usage are synchronized in time. A second communication method, denoted *request-reply communication* is supported via so called *services,* and may be used if information is needed before the execution of a node should continue. A node sends a request to another node, which replies with the requested data. This method is similar to a regular function call. A third method, denoted *actions*, is used when a node needs to wait for or supervise another node to complete a task. A relevant example is manipulator motion. By using actions, the calling node gets feedback from the manipulator node during motion, and is thereby able to cancel the motion or take other actions if necessary.

3   Technical aspects on and experiences from using ROS

In CROPS, ROS is used to construct software for a number of subsystems for sensing, perception, manipulator control, mission control, and system framework. Design considerations, with a specific focus on the use of ROS, are described and discussed in separate subsections below for each subsystem.

3.1 Sensing

The sensory subsystem is composed of two devices, a high-resolution colour CCD camera and a time-of-flight (ToF) camera. Additionally, two *virtual sensors* and a sensory system controller are provided. A virtual sensor is a software unit that acts like a sensor and internally retrieves and combines data from other physical, or virtual, sensors. One virtual sensor is a multispectral system with data acquisition achieved by a combination of a controlled filter wheel and a CCD camera (RGB and monochrome format). Another virtual sensor is the RGB-D system, performing data registration between a CCD camera (RGB format) and a ToF device (amplitude data, confidence map and point cloud data). Use of the virtual sensor concept allows a

multi-sensor framework to be easily adapted to various hardware configurations. The control system for the filter wheel and the DAQ card for image acquisition are hosted in the real-time framework xPC Target*, which communicates with ROS via TCP messages. Low-level control of the system is implemented in the real-time platform with ad hoc libraries programmed in C language, and high-level control integrated in a ROS node. The TCP messages transmit parameters and commands required for controlling and monitoring motion and data acquisition tasks of the sensory subsystem.

Each camera is implemented as a ROS node that provides facilities to dynamically reconfigure camera parameters without having to restart the node, as well as services for some parameters such as acquisition mode, exposure time, pixel format and integration time. Data acquired by each camera, and status of the acquisition, is transferred by publishing messages, instead of using request-reply services, as several subscribed nodes might require the same information. The sensory system controller node is in charge of controlling image acquisition, configuration of parameters, and filter wheel motion. The services of the camera nodes allow the sensory system controller to set specific hardware configurations required to generate data for the two virtual sensors. Synchronous image acquisition is achieved when the controller publishes a specific trigger message. This trigger message is sent when the filter wheel reaches a requested position. The motion to the next target position is sent as soon as the cameras data are successfully acquired.

3.2 Perception

Perception is the process of transforming sensor data to a higher level of abstraction. The perception architecture is illustrated in Figure 1. Each virtual sensor provides feature data. For detection of both apples and peppers several virtual sensors (e.g. NDI space, texture, colour normalization) are used.
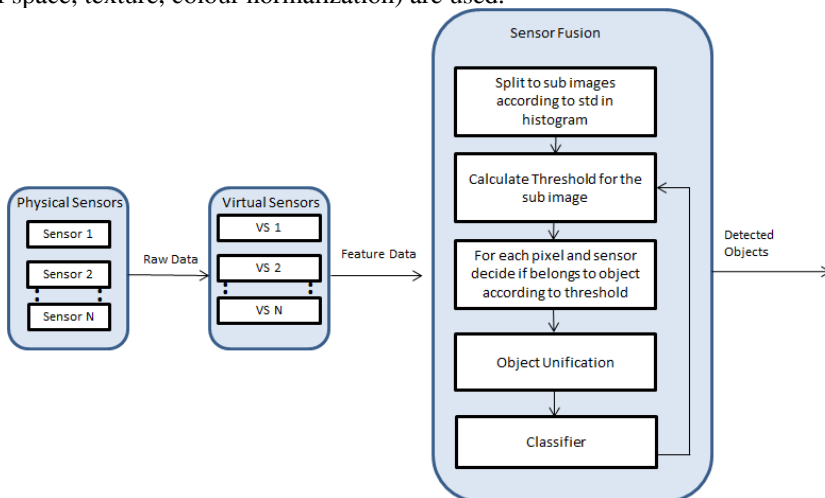

Figure 1. Perception framework

The sensor fusion block includes the following series of blocks that provide one output based on several feature data inputs. **Split sub images** – splits the image into several parts depending on variance in lighting conditions and information from other sensors. **Calculate threshold** – calculates the threshold for the sub image, and then decides on the existence of an object according to the threshold. **Feature classifier** – decides for each pixel, for each sensor if it belongs to an object (e.g., fruit) or not. **Object unification** – unifies close pixels marked as object into one object from different sensors. **Classifier** – classifies each object to its proper class (e.g., if it is a fruit or not). This task is performed with adaptive weighting fusion.

Each virtual sensor node sends feature data by a published message (2D matrix, with several dimensions according to the feature data) so that the sensor fusion node continuously receives updated data. The output of the sensor fusion node includes data on all detected fruit. Apples are represented by a sphere (X,Y,Z center coordinates in image coordinate system and radius in cm). Peppers are represented by a truncated cone (X,Y,Z and radius for upper and lower circles of the truncated cone). The output data is published continuously as a message since several clients need this information.

This modular framework was chosen to simplify integration of new virtual sensors, if and when they become available. Full implementation was conducted in ROS and in Matlab. The parallel implementation in Matlab enabled faster development and evaluation of algorithms. Since this project was characterized by algorithm development in parallel to development of the overall operational system it was important to have an independent tool which provides modularity and extensive toolboxes for rapid development. Once algorithms were finalized, transfer over to ROS was straightforward.

3.3 CROPS Manipulator

For robot actuation, low-level motor control and an interface to the joint sensors are required. Furthermore, for robot motion, the application of real time systems is very common. This allows for synchronized and predictable joint movement and gives the possibility to implement high-level control algorithms. Although ROS is not a real-time framework, there are several standard approaches to combine real time processes with ROS. First, one can use a standard PC with Ubuntu and ROS installed. With extension cards, low-level communication to the actuators can be provided. With a real time extension for Linux, it is possible to control the motors or communicate with the motor controllers. The advantage with this approach would be a very good integration of the software with ROS. However, even though real-time performance of Linux can be improved with the RT-Preempt patch, jitter and latency are still sub-optimal. A second approach is a real-time operating system with support for ROS. One can either implement a custom interface or integrate the ROS communication interface as an application, running on the real-time operating system (e.g. with the ROS package *rosserial*). For a more detailed discussion on this topic, refer to (Bouchier, 2013). Figure 2 shows the solution chosen for the CROPS manipulator. The hardware system consists of a real-time control unit running the commercial

operating system xPC Target[†]. The xPC is integrated with Matlab/Simulink and includes several hardware drivers enabling fast integration and testing. The realtime system handles control and low-level communication with manipulator joints and end-effectors. The hardware system communicates with ROS using custom implemented UDP messages, allowing monitoring and control of the manipulator. Since this is the only interface to the rest of the system, all ROS parameters and messages required for robot motion must be sent to the real time system by transforming them into UDP messages, and status messages from the real time system must be transformed back into ROS messages for monitoring. This requires additional programming effort. From our experience this is the main drawback of this approach. An advantage is the full support and good documentation from the company MathWorks, as well as independence from the middleware ROS. Safety features, like self-collision checks and inverse kinematic algorithms are implemented on the real time control unit. Custom interfaces to ROS high level functionality, like the motion planning library MoveIt (http://moveit.ros.org/), are provided.
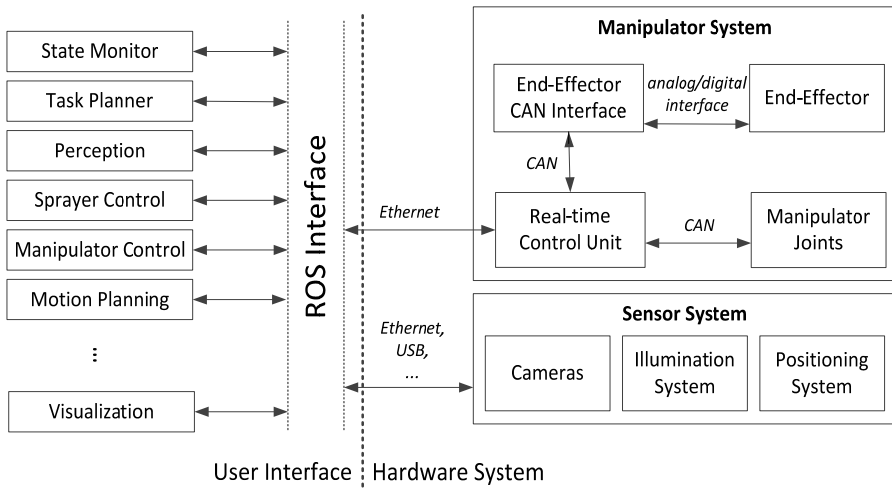

Figure 2. Hardware and software architecture of the CROPS manipulator

3.4 System framework

A generic software framework for development of agricultural and forestry robots was developed as part of the project (Hellström, Ringdahl 2013). The goal was to provide generic high-level functionality and to encourage distributed and structured programming, thus leading to faster and simplified development of robots. A framework developed with similar goals is presented by Jensen et al. (2012). Individual ROS nodes are able to respond to input, and can trigger other ROS nodes. In theory this allows a chained collection of ROS nodes to perform complex sequential tasks. However, building a program in such a way is not always appropriate since coordination and core functionality then is mixed. Our solution is to

---

[†] MathWorks (http://www.mathworks.com)

use *a general state machine* to organize and control the order of execution; the sequence of functionalities provided by the different ROS nodes. Each state represents an algorithmic functionality to be performed by one or several ROS nodes. The state machine manages and processes information returned from the ROS nodes, and passes it along as input to other nodes when needed. The state can use any of three available methods to communicate between nodes; publish-subscribe, request-reply, or action (see Section 2.3). When certain conditions are met, for example when computational results of the ROS nodes are reached, the state machine makes a transition to one of the possible next states. Centralizing this transition mechanism to a general system component simplifies implementation of user interfaces and error handling functionality. A *performance monitor* constantly checks the status of the software system. If an error is detected, an *error handler* decides on a suitable action, such as resetting a node or stopping the system and alerting the user via an error message in the *graphical user interface* (GUI). In addition to showing the status of the system, the GUI is also used to start, pause and stop the state machine, and hence the entire application.

3.5 Mission control

For a specific application, the algorithmic sequence to be executed is defined with a flowchart, which can be directly translated into the state machine as described in the previous section. In Figure 3, the flowchart for the CROPS sweet pepper robot is shown. It contains three main functionalities: initialization, sensing and harvesting. Note that there is a sub-state machine included for moving the manipulator. The state machine based framework has proven to be useful for coordinating and sequencing of operations. It logically arranges the computations to be performed by all ROS nodes and thereby separates the coordination task from the actual computations.

4 Experiences from working with ROS

In general, ROS has worked to satisfaction, and it has simplified integration of components implemented by different partners in the project. ROS provides useful tools and libraries that support introspection, debugging, visualizing the state of the robot system, and map based localization. ROS also provides seamless integration with other popular open-source libraries such as OpenCV, PCL, and MoveIt, which may add powerful capabilities to the robot.

However, as with any framework, it takes considerable time to learn how to use ROS, its tools and additional libraries. An experienced problem is that new ROS versions often lack full backwards compatibility and a lot of code normally has to be changed. This makes ROS particularly problematic to use in commercial applications. Therefore, we suggest switching to newer versions only after careful considerations and testing.

Although ROS has its own set of standard message formats that cover most common cases, customized message formats are sometimes required. They should follow the defined messages description language and should be agreed upon by all partners and approved by the responsible software architect.

ROS provides a possibility to maintain a stable publishing frequency of each message. Choosing the most efficient frequencies requires careful considerations by all involved partners.

Since ROS is a distributed system using messages to communicate between separated nodes, debugging can sometimes be difficult. It is not always obvious why things do not work, and in which node the error occurs. On the other hand, the distributed approach simplifies development of individual components since they often can be tested in isolation by simulating message communication with the rest of the system.

One initially overlooked decision was whether to use a 32 or 64-bit operating system. Whenever possible, the nodes in our system include both 32 and 64-bit libraries such that they can be used with both types of operating systems. However, some drivers are available only as 32-bit libraries, and can therefore not be compiled on 64-bit operating systems. Except for the mentioned problem, it is possible to have both 64 and 32-bit systems in a project, since ROS is able to communicate over the network regardless of operating system type.
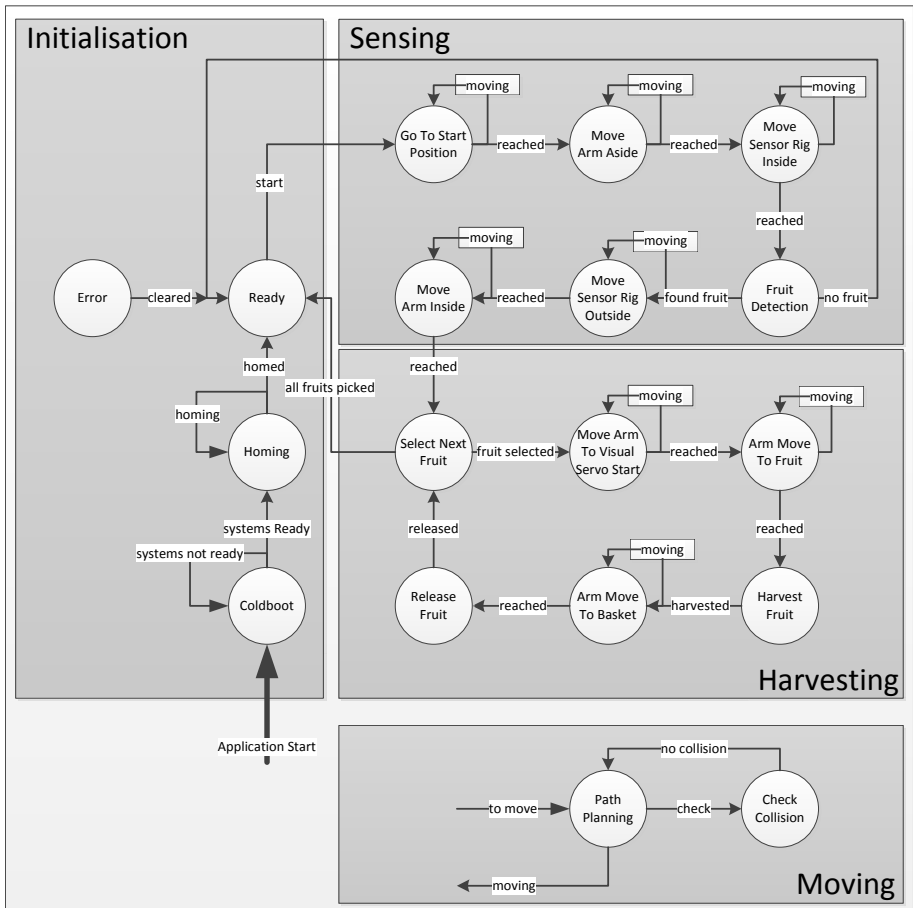

Figure 3. Flowchart for the finite state machine of the sweet pepper application

5 Conclusions and discussion

ROS is a very popular tool for development of software for robots. This fact alone is one of the biggest advantages with the system. Many manufacturers of hardware components supply ROS drivers. To avoid problems with missing drivers from third parties, it is advisable to use only 32-bit operating systems. The ROS user community is highly active. Solutions to many experienced problems can often be found on the ROS Wiki or elsewhere online.

Functionally, ROS normally does its job well. It does not support real-time response, which depending on the application can be a disadvantage or not. The modular approach makes is possible to develop time-critical components separately, with interfaces to the ROS system. The time to learn ROS should not be underestimated. To speed up the learning curve, using the tutorials on the ROS Wiki can be recommended. Prior knowledge of programming in C++ or Python is a clear advantage, if not a prerequisite.

References

Billing, E., Hellström, T. & Janlert, L. E. (2011). Robot learning from demonstration using predictive sequence learning. In: Ashish Dutta (Ed.), *Robotic systems: applications, control and programming* (pp. 235-250). Kanpur, India: IN-TECH.

Bouchier, P. (2013), Embedded ROS [ROS Topics], *Robotics & Automation Magazine, IEEE*, vol.20, no.2, pp. 17-19, June 2013, doi: 10.1109/MRA.2013.2255491.

Hellström, T., Johansson, T., and Ringdahl, O. (2008), A Java-based Middleware for Control and Sensing in Mobile Robotics. International Conference on Intelligent Automation and Robotics 2008 (ICIAR'08), San Francisco USA, pp. 649-654, 2008.

Hellström, T. and Ringdahl, O. (2013), A software framework for agricultural and forestry robots, *Industrial Robot: An International Journal*, Vol. 40, Issue 1, pp. 20-26.

Jackson, J. (2008), Microsoft robotics studio: A technical introduction, *Robotics & Automation Magazine, IEEE, 14*, 82-87.

Jensen K., Nielsen, S.H., Larsen, M., Bøgild, A., Green, O. and Jørgensen, R.N. (2012), FroboMind, proposing a conceptual architecture for field robots. Automation Technology for Off-Road Equipment (ATOE), International conference of agricultural engineering, CIGR-Ageng2012, pp. 8-12.

Makarenko, A., Brooks, A. and Kaupp, T. (2006), Orca: Components for Robotics, In International Conference on Intelligent Robots and Systems (IROS), pp. 163-168, Oct. 2006.

Pfeifer, R. and Scheier, C. (2001), Understanding Intelligence. MIT Press. Cambridge, Massachusetts.

Quigley, M., Gerkey, B., Conley, K., Faust, J., Foote, T., Leibs, J., Berger, E., Wheeler, R., and Ng, A. Y. (2009), ROS: an open-source Robot Operating

System, In *Proc. Open-Source Software workshop of the International Conference on Robotics and Automation (ICRA).*

Szyperski, C. (2002), Component Software: Beyond Object-Oriented Programming (2nd ed.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.