Bachelorarbeit

Milena Hippler

Robot Control Paradigms for reactive systems implemented in modern C++ standards

Milena Hippler

Robot Control Paradigms for reactive systems implemented in modern C++ standards

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. rer. nat. Stephan Pareigis
Zweitgutachter: Prof. Dr.-Ing. Andreas Meisel

Eingereicht am: 5. Januar 2017

Milena Hippler

Thema der Arbeit

Roboter Kontrollparadigmen für reaktive Systeme implementiert in modernen C++ Standards

Stichworte

Roboter Kontrollparadigmen, Roboter Kontrollarchitekturen, Reaktive Systeme, Robot Operating System

Kurzzusammenfassung

In diesem Dokument werden die deliberativen, reaktiven und hybriden Roboter Kontrollparadigmen, anhand einer oder mehrerer repräsentativen Roboter Kotrollarchitekturen, in Hinsicht auf Qualitätsmerkmale, reaktiven Systemanforderungen, empfohlener Einsatzbereiche sowie unterschiedlicher Implementationsansätze untersucht und miteinander in Vergleich gestellt.

Milena Hippler

Title of the paper

Robot Control Paradigms for reactive systems implemented in modern C++ standards

Keywords

Robot Control Paradigms, Robot Control Architectures, Reactive Systems, Robot Operating System

Abstract

Within this document will be the deliberative, reactive and hybrid robot control paradigms by reference to one or more representative robot control architecture with regard to quality characteristics, reactive system requirements, recommended fields of application as well as distinct implementation approaches examined and compared to each other.

Contents

List of Tables

List of Figures

Listings

1 Introduction

1.1 Motivation

Nowadays it is not possible to imagine a world without robots. They facilitate the daily life in several different areas. They range from industrial, to medical, to military and to domestic areas. Robots facilitate the domestic life in form of autonomic vacuum cleaner or gutter cleaner, they secure human life by bomb disposal, they relieve humans by the more qualified and quantified use in factories, and facilitate and entertain the human life by multiple other services in various areas. In summary the use of robots ranges a road field of applications.
Out of that reason it is both interesting and important to be aware of the imposed requirements of an application and how a robot must be architectural designed to meet those. It is therefore essential to be acquainted with basic robot paradigms, their representative architecture models and their advantages and disadvantages concerning different criteria.
For what kind of tasks is a reactive robot control architecture recommendable and when is a deliberative control architecture suitable? Do architectures, based on the same robot control paradigm, differ in their advantages and disadvantages? And is the combination of both to the hybrid robot control paradigm the catch-all ultimate solution? These questions will be considered in this thesis as well as a detailed overview which of the huge amount of architecture possibility is suitable for the own use-case.

1.2 Ambition

The aim of this work is to enable the reader to distinguish between the three robot control paradigms and to identify corresponding architectures, by the knowledge of their composition and structure. Moreover is the aim to provide a detailed analysis of the requirements of an architecture so that the reader is capable of making fundamental decision what kind of robot architecture is suitable for which intended use. Another ambition is to introduce the reader to the robot operating system ROS, which is enabling a facilitate implementation by its helpful complex environment, so that, after the architecture decision process is completed, the reader is able to start implementing the robot system without any greater research.

1.3 Classification

The thesis analyses the three basic robot control paradigms, which are based on the components sense, plan and act.
The deliberative robot control paradigm describes a hierarchical sense-plan-act cycle. This paradigm will be represented by the nested hierarchical controller architecture developed by Meystel [Meystel] and by the NIST real-time controller system architecture developed by Albus [Albus (b), Albus (a), Albus (c)]. Both Meystels and Albus' architecture designs will be analysed and compared to each other.
The reactive robot control paradigm describes a tight sense-act coupling and disregarded the plan component. This paradigm will be represented by Brooks' subsumption architecture [Brooks (b), Brooks (a)]. Also Brooks architecture will be analysed regarding specific criteria. Moreover will the subsumption architecture be adjusted and implemented in modern C++ standards.
The last robot control paradigm is the hybrid paradigm, which describes also a tight sense-act coupling but also considers the plan component as a part of the paradigm. Therfroe a combination of deliberative and reactive paradigm. The hybrid paradigm is represented by the autonomous robot control architecture developed by Arkin [Arkin (a), Arkin (b)]. Arkins architecture will also be analysed according to specific criteria. In more detail how both including paradigms are combinable and how their single advantages and disadvantages affect the combination.
Each architecture will be compared to the requirements of a reactive system and evaluated whether they accomplish them and why. Furthermore all of the previous mentioned architecture will be linked to the robot operating system and their structure and communication discussed.

1.4 Structure

The structure of this thesis is divided into three main parts.
The first part introduces to the the topic of this thesis in chapter 1 and provides an general overview of the basic knowledge about robotics and reactive systems in chapter 2, about the robot operating system ROS in chapter 3 and about robot control architectures in chapter 4.
The second part represents the main part of the thesis, the three different robot control

paradigms. Each paradigm will be first introduced by basic knowledge and its characteristics. For each paradigm will be at least one representative architecture introduced and analysed in respect of the main software principles for architecture evacuations. Moreover will be the communication and the structure decisions for an possible implementation of the specific architecture in the robot operating system discussed.
Chapter 5 will be concerned about the deliberative robot control paradigm which will be represented by the nested hierarchical controller architecture (NHC) and the NIST real-time controller architecture. In chapter 6 will be the reactive robot control paradigm introduced and represented by Rodney Brooks' subsumption architecture. This chapter will additionally contain, besides the usage of ROS communication and an architectural analysis, an implementation example of the subsumption architecture. In chapter 7 will be the hybrid robot control paradigm introduced and represented by an managerial architecture, the autonomous robot architecture.
The last part of this thesis consists of the chapters 8 and 9 and will provide a summary and a ultimate comparison of the analysed architectures and therefore control paradigms. Furthermore will be a perspective provided for prospective works.

2 Robots and Reactive Systems

This chapter provides the definition of a robot and on which kinds of robots this works is concentrated. Furthermore it introduces reactive systems and combines the requirements of a reactive system with robot control architectures and outlines what reactivity means in robotic systems.

2.1 Robots

The question how to define a robot differs a lot depending on the sources and orientation. this work specializes on intelligent autonomous mobile robots.
Murphy defined an intelligent robot as "an intelligent mechanical creature which can function autonomously" [cf. Murphy]. The "mechanical creature" implies that a robot consists of a combination of mechanical parts, which also includes that a robot consists of actuators which convert electrical signals into mechanical movements (see subsection 2.1.2). "Intelligent" describes the ability of a robot to process with a certain reason and well considered decisions and is able to adapt and react to situations instead of processing in a "repetitive way" [cf. Murphy]. "Autonomously" implies that a robot is able to process without support of human beings and that the robot is able to react to environmental condition by itself. This also implies the sensory of a robot which observe the environment (see subsection 2.1.1).
Arkins definition of a robot if following "a machine that is able to extract information from its environment and use knowledge about its world to move safely in a meaningful and purposeful manner" [cf.Arkin (b)]. This definition is also implying that it gathers data via sensory from the environment and is able to use this knowledge to move by its actuators autonomously through its environment.
In summary a robot is a machine that senses its environment, plans intelligent and acts autonomously. In chapter 5, 6 and 7 will be the function and their combination analysed.
The sensing is based on sensors, which are defined below in subsection 2.1.1 and the action is based on actuators, which are defined in 2.1.2.

2.1.1 Robot Sensory

A robot consists of a myriad number of various sensors. Most sensors are comparable to animal or even human senses. The aim of sensors is to achieve a simultaneous confrontation to the world and itself like biological creatures do. In some areas sensors are even more effective, but in most of the cases sensors still lack in time management and reactivity. Robots need sensors to receive information from the external environment around it and also to monitor its internal environment.
Those two different types are called: exteroceptive and proprioceptive sensors.
The exteroceptive sensors obtain information about the robots external environment. They are comparable to the biological five senses: vision, audition, gustation, olfaction and somatosensation. The most important robot senses are vision for example for wandering around and fulfilling tasks and also audition for example for speech recognition. For the vision includes a robot system usually a camera and robot vision software designed for tasks as detecting edges, enhancing contrasts and most important for regular robot tasks recognizing objects. In this category also includes obstacle detectors. Those operate using ultrasound or lasers, they emit a signal and receive the echo from an object. Audition sensors detect sound are essentially microphones with complex signal-processing capability. Moreover are exteroceptive sensors, sensors which measure the environments condition like temperature, humidity, radiation, pressure, etcetera. A sophisticated robot system is constituted of a cooperation with these sensors.
The proprioceptive sensors monitor the robots internal environment. Internal environment includes the monitoring of the robots extremities like angle of a leg or an arm. This observation is quite similar to biological creatures. Proprioceptive sensors that differ from them are for example monitoring the robots battery voltage, since a biological system is not composed of those energy input. An other example is the observation of the robots wheel rotation, since wheels are no part in biological organisms. The aim of proprioceptive sensor is to control the robots inner states to identify and correct faults.

2.1.2 Robot Actuators

Robots are also composed of multiple actuators which enable the robots to interact physically with their environment. Actuators convert energy into physical motion. They need as input a control signal to execute an action. Moreover they need power to function. The state of the control signal determines the type of the actuators. The signal can be hydraulic, pneumatic, thermal, magnetic, mechanical or electrical (for information [Actuators2015]). In mobile robotics the electrical actuators are the most common actuators the electric motors. Electric motors provide the locomotion by powering wheels or legs and provide the manipulation by actuating robot arms.

2.2 Reactive Systems in Robot Control

Since a numerous of robots are operating in most cases in an open world, which is changing dynamically, it is important to assure the reactivity. Wieringa [cf. Wieringa] described a reactive system as "a system that, when switched on, is able to create desired effects in its environment". The nowadays aim of a robotic system is also to create effects in the environment be it intelligent planned or reactive acted effects.
The reactive manifesto [cf. Boner u. a.] imposes flowing requirements :

- Responsiveness:
  The response to the external stimuli shall take place as immediate as possible to ensure real-time.
- Resilience:
  The system shall be robust against internal and external failures.
- Elasticity:
  The system shall be able to process reliable in case of varying stimuli.
- Communication:
  The communication shall be based on asynchronous messages between the components.

A reactive system shall continuously interact with its environment in a stimuli-response process. The stimuli is the in robotics the sensory input, it occurs when the environment changes. The response process is equivalent to the act component of a robotic system. Reactive systems are often structured in a number of processes which will be processed concurrently. This is similar to the robots behaviours which are also concurrent processed and defines the reaction to the input in real-time. The behaviours and the real-time response as well as the mentioned requirements are, depending on the architecture, usually realised in the reactive (see chapter 6) and in the hybrid paradigms(see chapter 7). The real-time requirement is also partially realised in deliberative architectures (see chapter 5).

3 Robot Operating System

This chapter introduces to the ROS - Robot Operating System, which simplifies the software development of modern robot systems by its collection of software frameworks. This collection consists of tools, libraries, hardware abstractions, device drivers, visualisations, communications, packages and conventions. Therefore its advantage is that it provides lots of infrasructure, tools and thereby capabilities.
ROS is an open source meta - operating system for robots. ROS requires an operating system. Linux is recommended. Windows and MacOS are not supported very well at this time. ROS is not a real-time framework, although it is possible to connect real-time component with each other. It supports programming languages like C++, Python, Lisp and Java (where C++ and Python are the most supported ones).
By the use of ROS it should be kept in mind, that it is not recommended for system with high demands on security or scalability, since those are not the first class concerns of ROS yet, in first case it is about an easy use of tools and algorithms.
This chapter solely describes the basics like the structure and system environment of ROS. In the following chapters, the subsections 5.1.1, 5.2.2 and 6.1.2 are specific use cases concerning the communication implementation and also file system in ROS explained.

3.1 Supported Robot Platforms and Sensors by ROS Packages

ROS is used by multiple kind of robots. The peripheries of a robot system and therefore also for ROS are hardware and multiple sensors.
ROS supports the main categories of robotics which are listed in table 3.1.

The by ROS packages supported sensors are 1D and 2D range finders (for example distance calculation), 3D sensors for also range calculation, RGB-D cameras and cameras in general. Moreover motion capturing and position estimation sensors which are as well as the before mentioned sensors mainly for the localisation and locomotion of the robot system and its environment necessary. Also Audio Speech Recognition sensors are supported by packages to ensure communication interfaces for mostly social robots. Environmental sensor which are used for measuring environmental conditions such as temperature, gas, pressure and humidity

Category	Description
Mobile manipulators	Robot system compound of a robot arm and a autonomous base.
Mobile robots	Capable of automatic locomotion
Manipulators	Enables the physical interaction with the environment. It is the movable part of the robot, which does the mechanical work of the robot. The kind of robots which are commonly associated with industrial robots.
Autonomous cars	Autonomous driving vehicles.
Social robots	An autonomous robot which communicates and is based on behaviours and rules according to its role.
Humanoid	A visually human resembling robot
UAVs	Unmanned Aerial Vehicles.
AUVs	Unmanned Underwater Vehicles.

Table 3.1: Robot Categories

are also supported. Moreover power supply and RFID sensors and other sensor interfaces (view a full list at ROSwikiSensors2016) are supported.
In conclusion are the most basic robots and sensors supported by packages. If not there is the possibility to add it or to ask the community which extends ROS permanently.

3.2 Structure

ROS is composed of three basic conception layers. Those layers are the file system (see subsection 3.2.1), the computation graph (see subsection 3.2.2) and the community. Where the community represents the software and information sharing with other developer and is not relevant for this work, thereby not further described. For more information about the ROS community see [ROSwikiCommunity2016].

3.2.1 File System

The ROS file system consists of stacks, packages, message types and service types. Moreover manifests for stacks and manifests for the packages. These manifests comprise meta data, license informations and dependencies.
Stacks are composed of multiple packages which have the same functionality and thematic. Stacks can have dependencies to other stacks. Therefore they create modularity by condensing functionality and connection to other functionalities, which makes the file system clearer.

The packages shall provide a specific functionality. This functionality has to be reusable. Due to the aim of re-usability the packages must be as general as possible so that the same mechanism is usable for more scenarios and applications. Packages include libraries, data sets, configuration files and more particularly the nodes which are containing the main processes and will be elucidated in section 3.2.2. According to this the packages represent the software organisation of ROS and the software processes.
Message types and service types are also included in the file system and are located in the respective package they are used in. Both represent a data structure for the communication. The message types are for the topic-based communication and the service type for the service-based communication which will be explained in section 3.3.

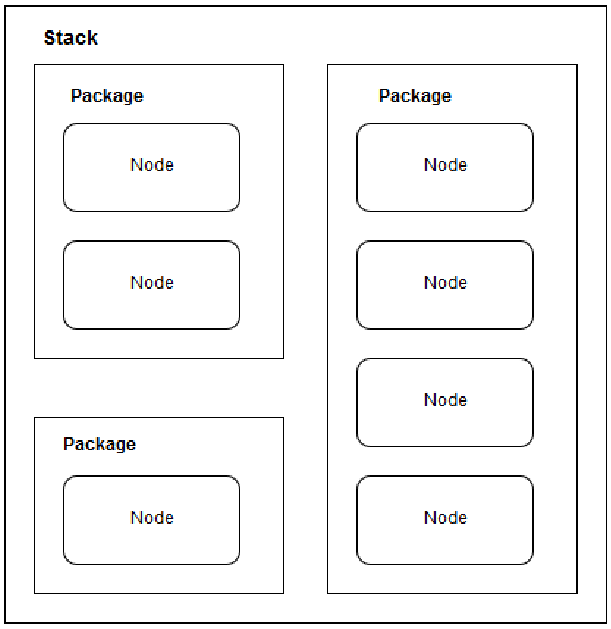


Figure 3.1: Abstract composition of the ROS file system

Summarized with regard to functionality, execution is the main structure as visible in fig. 3.1. The Stack includes multiple packages (in this example two packages) and each package includes a various number of nodes. This structure should be kept in mind for the visualizations in the following chapters, since their composition is optimized away due to further abstraction.

3.2.2 Computation Graph

The computation graph represents the interaction of all components. The ROS wikis definition of the computation graph is that it is a peer-to-peer network of ROS processes, which fulfil a task together [cf. ROSwikiGraph2014].
Since ROS was developed for service robots with high complexity requirements, an efficient way for the data switching had to be found. Commonly robot networks are designed, that multiple computers in the robot itself are connected via Ethernet and transfer data via wireless LAN to high-performance data processors, where complex computations like image processing and speech recognition are performed. This wireless LAN connection affects the performance of the system and is the bottle-neck of it. By using the in the definition mentioned peer-to-peer architecture with input buffer and output buffer this deceleration can be prevented. For ROS are sorely mechanism for processes necessary for finding their communication partner while runtime.
The computation graph consists of components which represent the handling of the system and also of the systems communication. Those components are the nodes (see subsection 3.2.2), messages (see subsection 3.2.2), topics (see subsection 3.3.1), services (see subsection 3.3.2), bags (see subsection 3.2.2) and the master (see subsection 3.2.2).
The communication between these components or by these components are demonstrated and explained in section 3.3.

Nodes

The ROS nodes are the executing instance which are basically processes which do a specific computation / functionality. Nodes can be related to a sensor, a motor or any sort of algorithm. In a robot system multiple nodes exist which are independent from each other. These independent nodes are able to work together and have two options of communication which are specified in section 3.3. The independent but cooperating nodes represent the modularity of ROS. Resources (node) can be easily replaced or changed without affecting the whole system. Each node has to register itself to the master (see information about the master at 3.2.2).

Master

The master is a server that tracks the network addresses of all nodes and the other information like parameters. It manages the registration of the nodes. It also provides the service that a node can find other nodes because of the registration service. Thereby each node knows every other node because of the master and through the master. The master saves which

node publishes a topic and which topics it subscribes (for more information about topics see section 3.3), therefore it has information about the communication and about the nodes. Acoording to this the master also provides a service for finding communication partners of the nodes while run-time, since it has every node in the system registered. It also informs the subscribers about other nodes publishing on the same topic. In conclusion it means, that without a master there will not be any communication possible. When running a node (with $\$rosrun\ [package\_name]\ [node\_name]$), the node must know about the address of the master. Therefore the first thing to do in ROS before starting to run the nodes, is to start the master by the command $roscore$. The IP-address of the master is hard coded in the system and can be used by the constant ROS_MASTER_URI.
The concept of the master is shown in following figure 3.2 which represent an example of the master concept by a topic communication. $Node0$ informs the master on which topic it is publishing and its IP-address. $Node1$ requests subscription to that specific topic and the master provides it the information about which node is publishing on it.

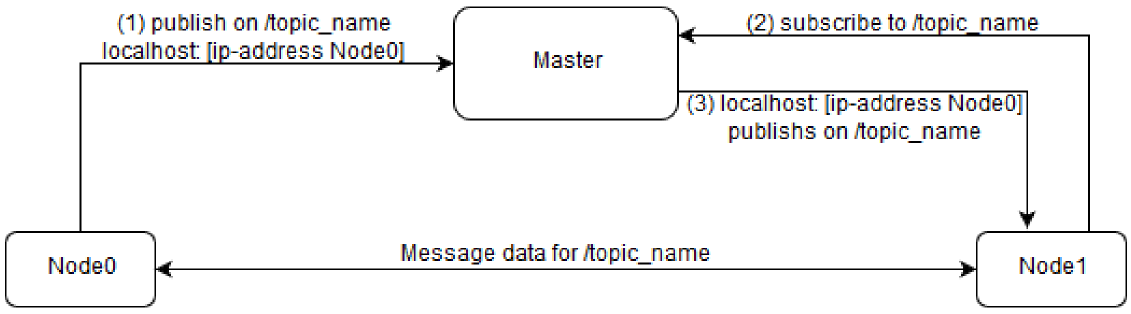

(1) publish on /topic_name
localhost: [ip-address Node0]
Master
(2) subscribe to /topic_name
(3) localhost: [ip-address Node0]
publishs on /topic_name
Node0
Message data for /topic_name
Node1

Figure 3.2: Publisher and Subscriber Mechanism for Topics

Bags

Bags store and load ROS messages. For example it is possible to save sensor data information of a system. The sense in reloading those information is that it enables developers to debug the system retrospectively. Also analysis on original data are possible and realised by bags.

3.3 Communication

Nodes communicate through messages. A message is a data structure which includes the standard data types of the message parameters. Also arrays are supported. Messages can also

contain any number of nested structures.
ROS offers two different kinds of communication. A synchronous communication is realized by services. Services allow bidirectional communication between two nodes. The complement of the synchronous communication are the communication by use of topics. Topics represent the asynchronous communication and realises a many-to-many relation between multiple nodes. It is also based on a unidirectional communication. All in one can a node use both topic and service for communication with other nodes.

3.3.1 Topics

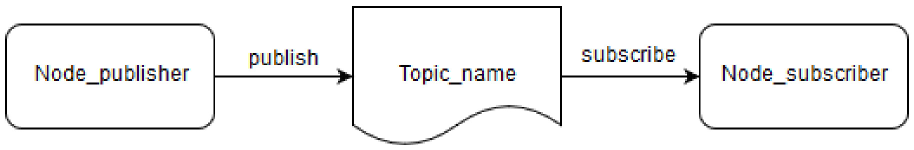


Figure 3.3: Publisher and Subscriber Mechanism for Topics

Topics are used by nodes for information sharing. They represent the asynchronous communication in ROS. The communication is unidirectional which means that the writer node and the reader node do not need to know about each other. The topic communication is comparable with the communication via memory cells.
This mechanism is called publisher and subscriber system. A node publishes information to a topic (see fig. 3.3 left side of $Topic\_name$). The topic can be identified by its name and the master saves the information which node is the publisher for this topic. If another node is interested in this topic and its messages, it subscribes to it via the master who saves also the subscriber of the topic (see fig. 3.3 right side of $Topic\_name$).
The asynchrony causes that the subscriber can read the published information whenever it wants.

As in fig. 3.4 visible one or more nodes can publish information to a topic (see $Node0$ and $Node1$ publish to $Topic\_name1$). Also visible is that one or more nodes can subscribe to those information (see $Topic\_name1$ subscribed by $Node2$ and $Node3$). Moreover nodes can also function not only as a publisher but also as a subscriber at the same time (see fig. 3.4 $Node4$ publishes and subscribes to $Topic\_name0$).
In conclusion one node can publish multiple topics and subscribe multiple topics at the same time, this is called the asynchronous many-to-many communication.

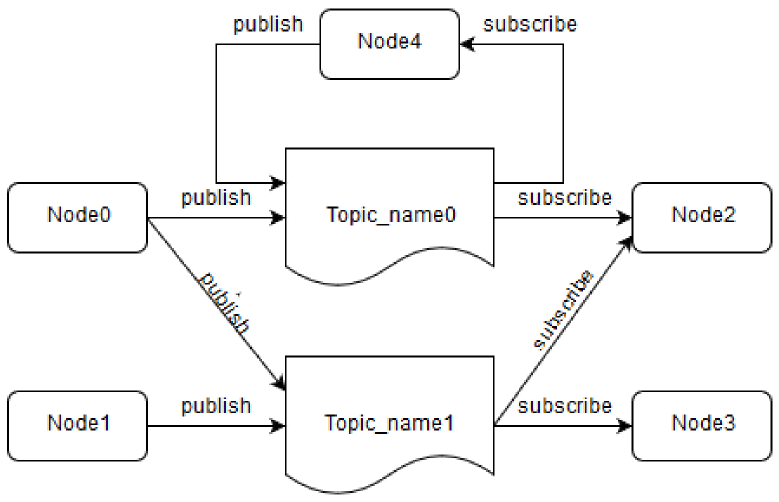


Figure 3.4: Publisher and Subscriber Mechanism for Topics

3.3.2 Services

As distinguished from the asynchronous many-to-many topics, a service offer a direct communication between two nodes. This bidirectional communication is synchronous. It is a request and response interaction between two nodes and is comparable to the remote procedure call. One node offers a service, which is identifiable by its name (see fig. 3.5 *Node0* and *Service_name*). The node which is interested in the service sends a request and waits for a response from the service offering node and then receives its response (see fig. 3.5 *Node1* and *Service_name*).

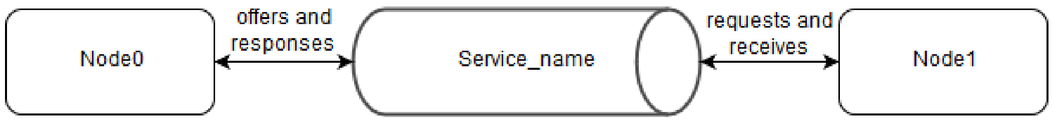
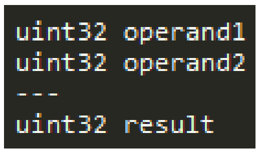


Figure 3.5: Service

The structure of the request and response messages is defined in the already mentioned service types (svr-file) (see. 3.2.1). For example a node wants the other node to solve a calculation by corresponding via a service. The svr-file would look like in fig. 3.6 where the first two lines represent the request message and the line after the three dashes (last line in figure) represents the response message type.



```
uint32 operand1
uint32 operand2
---
uint32 result
```

Figure 3.6: Service Type Message

A client node requests the execution of an additional function and transmits the parameters via the message which shall be executed and the server node responds by using the result parameter.

3.4 ROS Environment

ROS was implemented in a micro-kernel design to reach high stability and minor complexity. Therefore ROS does support a variety number of tools which are responsible for the execution of multiple components of ROS.
In the following are some most used tools of ROS. These tools are supporting mostly different kinds of visualisation which simplify the use of ROS. *Gazebo* is a 3D simulation environment. It simulates robots and their interaction with their environment by physical processes. This simulated sensor data, robot states and objects in the environment are published by ROS. This published data can be received and used by nodes. Gazebo offers a graphical user interface which enables a visual monitoring and interaction with the simulated environment. the complement for a 2D simulator is the tool *Stage*. The tool *Rviz* is also a 3D visualisation system, but in difference to Gazebo it has no motor, it visualizes the view of the robot. ROS supports *OpenCV* which is used for image processing.
ROS supports QT. QT is a library for cross-platform programming of graphical user interfaces and is named rqt in ROS. It is mainly used for the depiction of the peer-to-peer topology. This is provided by the *rqt_graph* package. It allows to see the connection between the nodes.
Another visualisation provides the *rqt_plot* which enables a live plotting of published data to ROS topics.
Moreover the in section 3.2.2 introduced bag has a visualisation package named *Rqt_bag* which enables to see the recorded data in the bag files and offers multiple editing tools.
Besides visualisation ROS supports several command line tools. That would be a navigation through the packages (*roscd*), listing of the transmitted data via topics (*rostopic*) and via services (*rosservice*) and information about the nodes (*rosnode*). Also offers command lines

for file system installing and multiple command lines for interaction and debugging the running system.

4 Robot Control Architectures

An architecture is based on several strategical decisions about the organisation of a system. It is composed of multiple components and represents the relationship between those components and how as well as about what they communicate with each other. An architecture defines structure of the system and may also define the structure of its sub-systems, its components. Moreover is an architecture is a method of implementing a robot control paradigm.
The Oxford Dictionary [cf. OxfordDict] defines a paradigm as follows: "A typical example or pattern of something; a pattern or model". A paradigm represents therefore an architecture since it is used to structure element (here components) in a certain way.
Murphy [cf. Murphy] described that control paradigms are based on the relationship between the three basic components: sense, plan and act. The way of how they are organized represents the specific paradigm. A specific paradigm can be represented by various architectures as long as they keep the paradigms structure. In terms of paradigms for robot control we speak of three basic paradigms which differ in the disposal of the components sense. plan and act.
The first robot control paradigm is the deliberative or also called hierarchical paradigm (see chapter 5), which focuses on automated reasoning and knowledge representation. This paradigm is represented by Meystels [cf. Meystel] nested hierachical contoller architecture (NHC) and by Albus [cf. Albus (a)] NIST real-time controller system architecture (NIST RCS).
The second paradigm for robot control is called reactive or behavioural paradigm. In opposition to the deliberative paradigm it does not plan its actions, it reacts to conditions of the environment and thereby pursues to act like a reactive system (see explanation of the definition of reactive systems introduced in chapter 6).
Now there's a robot control paradigm which plans next steps and one which reacts. The result of the combination of these two robot control paradigms is the third paradigm of robot control, the hybrid. A hybrid robot control paradigm is able to plan steps automatically, but it is also reactive if necessary. Modern robot architectures are based on the concept of hybrid paradigms. This paradigm will be introduced in chapter 7.
The evaluation and analysis of these paradigms and their representative architectures is based on several system engineering principles. Principles or criteria such as the robustness of the

architecture. Is the architecture robust enough and especially secure enough to operate in an open world and how it will be analysed if the architecture was ever successfully used for robotic systems and especially mobile robotic systems. Another aspect is the support for modularity. Is the system decomposed by functionality and is therefore maintainability ensured. Is the architecture expandable by new components and how does is react to removal of components. Moreover is the portability an analysed aspect. Is it possible to use the system for other robots, would the code be re-usable or is an amount of code re-writing necessary to port it to another robot. The following chapters will analyse these issues and give an overview of which architecture suitable for specific use cases and environmental conditions.

5 Deliberative Paradigm

The deliberative paradigm is based on the classical sense-plan-act cycle (see fig. 5.1). It is also called hierarchical paradigm, because of the hierarchical order of the instances.

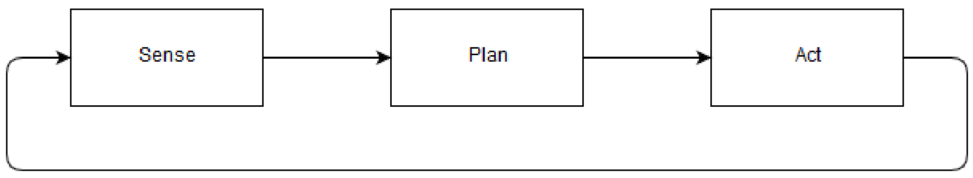


Figure 5.1: SPA-Cycle

The basic model contains three components: sense, plan and act.
The concept is that the robot senses the world, updates its world model by the sensed data, then generates a plan to fulfil a specific task and finally executes the plan. This process will be repeated until the task is accomplished.
The sense component receives data from the sensors. According to this data it modifies its world model. The world model represents the robot's internal global map of the environment, it gathers all sensing data. This map can be either given by default (in case of a static environment) or created by algorithms. Since robots, except for industrial ones in a production plant, do have a dynamic environment nowadays, we assume that the robot has to explore the world itself. This leads to a slightly different model of the SPA-cycle which is shown in figure 5.2.

Below in figure 5.2 the sense component is split into two parts: Perception and map-building/ localisation. Perception is the part where the robot processes the sensor data and merges it with the update data from the last cycle iteration. This dataset influences the localisation and map-building instance. This instance locates by the data the robot in its world map and updates the world model. Localisation and map-building contain the algorithms. The second SPA-cycle component is the plan component. As input it receives the current world model. It contains the strategy of generating the less expensive way to fulfil a given task. It is so to speak a problem solving component. Planning is the part with the most computational expense. The more complex a world model is or gets, the longer the generation of a plan will take. That is because

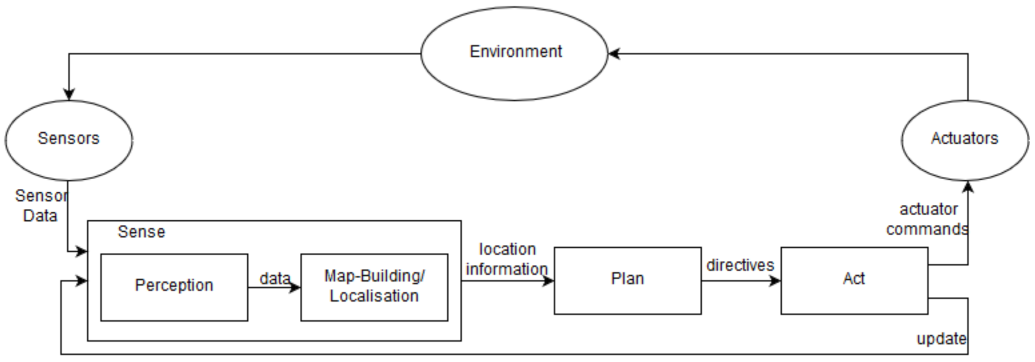


Figure 5.2: SPA-Cycle with Sensor Expansion

it tries to find the less expensive possible plan in the whole world model, like for example the shortest way to an object (Dijkstra algorithm [(Cormen u. a.)]).
The output of plan are directives for the act component which is the last one in the cycle. It actuates the actuators to execute the computed plan. After executing the plan the process starts again. This will be repeated until a task is fulfilled.

5.1 Nested Hierarchical Controller Architecture (NHC)

The nested hierarchical controller is one of the best known architectures which represent the deliberative paradigm. It was developed by Meystel [cf. Meystel]. Three hierarchical ordered components sense, plan and act compose the nested hierarchical controller architecture. Therefore it keeps the sense-plan-act procedure of the paradigm. The major contribution of the architecture of the nested hierarchical controller is the decomposition of the plan component into three different functionalities.

As visible in the yellow box in figure 5.3 the sense component contains the world model. The world model receives all sensor data of the environment from the various sensors, combines them. The world model creates according to those data an internal map on which the planning and therefore the actions are based on. Because of that dependency it is important that the world model is always up to date and has no wrong information about the environment. After each execution of an action the sensors sense again and the world model gets updated. The world model can also contain a knowledge base of the world, for example a basic map, and then complement the knowledge by the sensor and update data.
The procedure of the plan component (see the blue box in fig. 5.3) is parted into three functions: the Mission planner, the navigator and the pilot. Each function is connected to the world

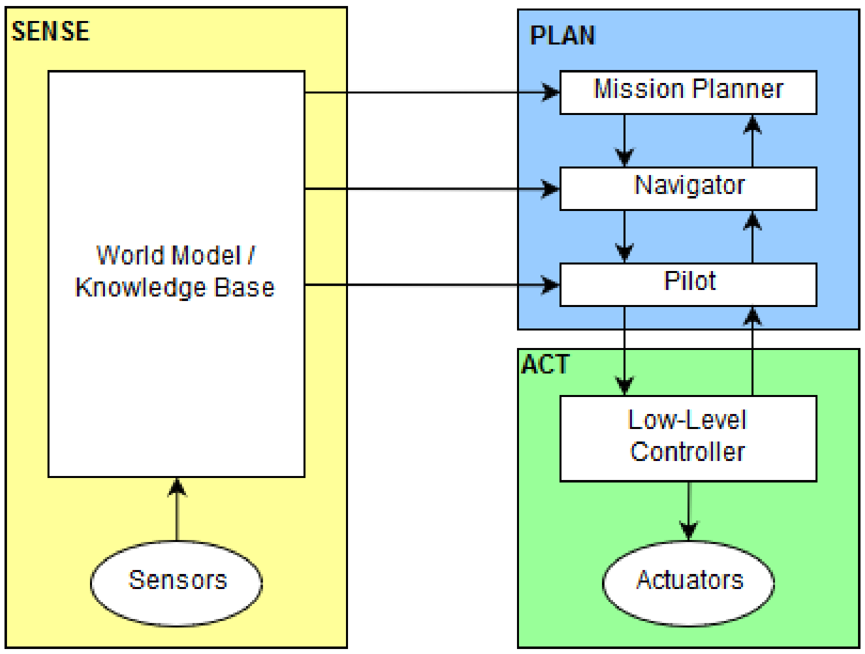


Figure 5.3: Nested Hierarchical Controller Architecture

model, therefore each instance is able to work on the current world model data.
The mission planner receives a mission or it generates the mission itself. The generation of the mission happens if a new mission occurs. The given mission will be translated into terms which are understandable for the other functions. The mission planner uses the world models map to locate the robot and to locate the goal (see in fig. 5.4 the R for the robot location and the G for the goal location).
The mission planner passes these location information to the navigator (see fig. 5.4 $positions$). The navigators task is to generate a path from the robots current position to the position of the goal. This path can also be a number of segments of the whole path to achieve a sub-goal if the goal is not reachable in one straight calculation (for example in case it needs to avoid objects and therefore change directions). The generated path is handed to the pilot (see fig. 5.4 $path$). The pilot generates which action the robot needs to execute the planned path or path segment by the received plan(for example turn left / right, move one meter forward). In figure 5.4 the path segment from the robot R to the sub-goal SG is visible, which will be executed first before reaching the main goal G.
The act component (see the green box in fig. 5.3) includes the low-level controller and the diverse actuators. The action commands generated by the pilot that let the robot execute the goal or a sub-goal, are handed to the low-level-controller which translates them into actuator

control signals and puts the actuators in action by these signals. The actuator and therefore the robot execute the plan.

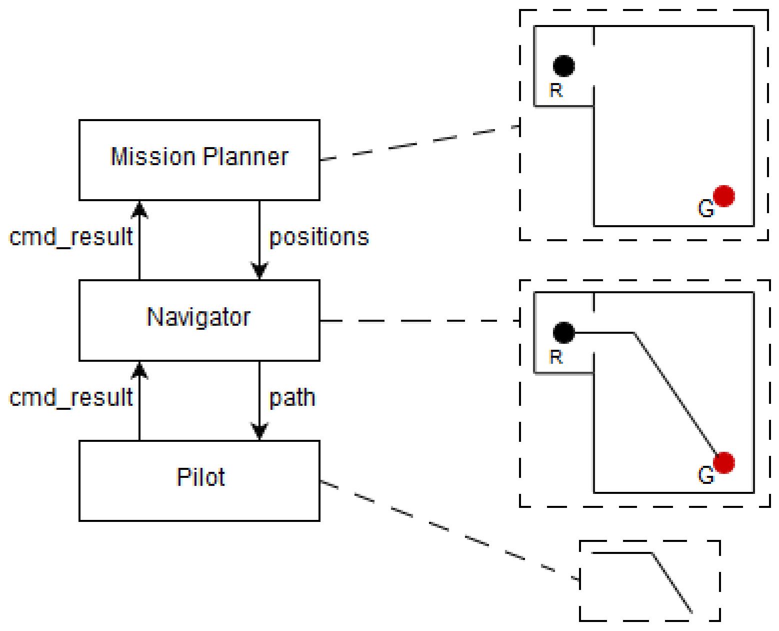


Figure 5.4: Plan Component of the Nested Hierarchical Controller Architecture

After that the sense component will be executed again and the world model gets updated. Unlike in figure 5.2 the whole planning process will not be repeated. The pilot takes the updated world model and checks if the goal or the sub-goal is accomplished. This check will lead to three possible results. The first result is visible in figure 5.5 a), that means that the robot (see R) has reached the goal (see G). In this case the pilot informs the navigator about the success and reverse forwarding is the mission planner informed by the navigator (see fig 5.3 cmd_result). In case b) the goal was not reached, but a sub-goal (see SG). The pilot needs now a new path from the reached sub-goal to the main goal or to another sub-goal, therefore the pilot informs the navigator that a new path is required. In the last case c) the robot encountered an obstacle (see O) and cannot completely execute the planned path. To receive an alternative path to avoid the obstacle, the pilot informs the navigator about the obstacle, the navigator uses the updated world model and includes the obstacle in its calculations and generates a new path to the goal or sub-goal with avoidance of the obstacle and hands it back to the pilot. In either case b) and c) the mission planner is not included in the path planning process, because the mission remains the same. The concept of reverse communication between the functions saves a lot of computation time since the mission planner does not need to plan the mission again although its still current.

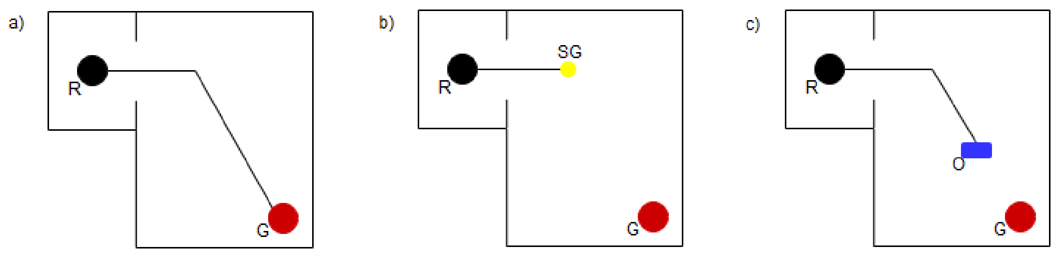


Figure 5.5: Message Possibilities of the Plan Component

Especially case $c)$ implies that the nested hierarchical controller architecture functions conditional in a dynamic world, since the pilot observes the world model directly after giving the execution command to the actuators and while the world model gets updated by the sensors. It is not very reactive, but still the system is able to react to some events in the world.
The decomposition of the plan component into the functionalities of mission planning, navigating and piloting is inherently hierarchical in intelligence and scope. The mission planner has the highest scope and abstracted view since it is aware of the mission. The navigator has a more decreased scope than the mission planner, but has the details increased. The pilot has the most detailed view but the most decreased scope.

5.1.1 Nested Hierarchical Controller Architecture and ROS

To suggest an implementation of the nested hierarchical controller architecture in ROS the nodes must be identified first. Each of these components world model, mission planner, navigator, pilot and low-level controller are represented as single node, because the all define a specific functional process. The mission planner, navigator and pilot node need information from the world model about the sensed environment. It must be decided whether a communication via service (see definition 3.3.2) or via topic (see definition 3.3.1)is more useful.
It is more useful to let the world model publish its information to a topic so that each node is able to extract the information when they need it. Moreover is a topic an unidirectional communication and since the world model only gives information and does not receive any, it is suitable to use this communication as a topic. That the topic allows an asynchronous request is the main advantage for the use of it in this case, the world model is able whenever it receives new data, to publish them to the topic and the plan component nodes are able to subscribe to the environment data topic and can retrieve the necessary information about it

for the planning (see fig. 5.6).

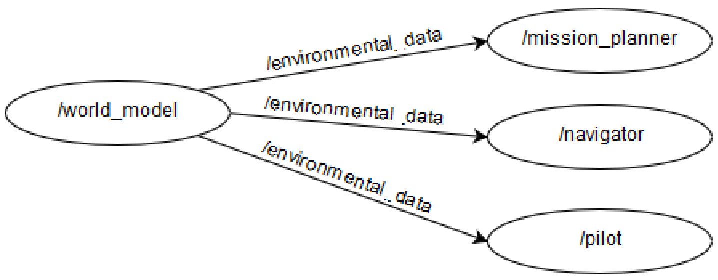


Figure 5.6: Communication of the Nested Hierachical Controller Architecture via Topic

The other is communication within the plan component between the mission planner, navigator and pilot will be realised by a service between the mission planner node and the navigator node and between the navigator node and the pilot node (see fig. 5.7). It is recommendable to use a service, because it offers a synchronized bidirectional communication between nodes. Such a communication type is needed for example when the pilot receives the feedback and has the result to the executed path that an obstacle appeared. The navigator must be informed about that result very fast and has to send a path for the avoidance as response back to the pilot in time. Due to the connection establishment it can be assured that the request will be processed immediately. The pilot for example requests a new path for obstacle avoidance with the "obstacle" parameter which is a string and includes the result and the navigator responses by sending the new path segment for example by giving it two-dimensional coordinates (see service type message in fig. 5.8).

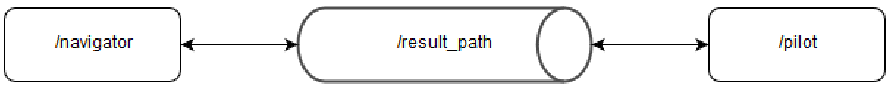
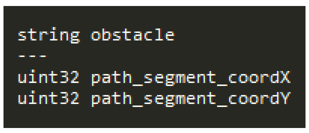


Figure 5.7: Communication of the Nested Hierachical Controller Architecture via Service



```
string obstacle
---
uint32 path_segment_coordX
uint32 path_segment_coordY
```

Figure 5.8: Service Type Message of Nested Hierachical Controller Architecture communication via Service

5.1.2 Analysis of the Nested Hierarchical Controller Architecture

The use of the nested hierarchical controller architecture is recommended if the robot concentrates exclusively on navigation task. The decomposition of the plan component induces a lot of advantages, but also the disadvantage of this decomposition being exclusively suitable for navigation tasks. Because of the strong binding of each functionality in the plan component to localisation and path finding solutions, the component is not able to solve more complex tasks as for example stacking boxes. Or as Murphy [cf. (Murphy)] describes the specialisation on navigation tasks: "the division of responsibilities seems less helpful, or clear, for tasks such as picking up a box, rather than just moving over it". Those use cases would involve more functions in the plan component which would also have to be combined with the navigation functions. To create that a lot of work would be necessary and the result architecture would not be efficient enough in comparison to the next introduced NIST real-time controller system in section 5.2 which is able to accomplish more goals than navigation.
Parts of the nested hierarchical controller architecture are very often included into other architectures as either hierarchical or hybrid. The just mentioned NIST real-time controller architecture is conceptual based on this architecture by taking advantages of the knowledge base and the idea of plan decomposition into subtasks. Moreover the nested hierarchical controller architecture has been used for many application like vehicle guidance. Also this architecture is used for in game and computer simulation development for virtual agents and non-player characters (NPCs).
One of the mentioned advantages of the decomposition of the plan component is that it causes modularity since it is partitioned by functionality. The mission planner is responsible for the mission and localisation, the navigator for generating a plan for the path and the pilot to generate a actions for the received plan. This modularity increases also the maintainability, since these parts are easily to identify in case of malfunction and to be replaced or repaired.
Speaking of maintainability leads to question the robustness of this architecture. The nested hierarchical controller architecture is not primary focused on robustness. The only feedback it receives is by the sensors which observe the process of execution the plan by the actuators. The architecture has no integrated previous simulation to ensure the robustness of a plan or to be specific a path. On the other hand the architecture enables the robot to react to the sensory input without rerunning the whole plan component. For example if suddenly an object occurs along the robots path segment which is already in progress, the sensors sense the object and the pilot receives the information and requests a new plan from the navigator to avoid the obstacle. This process still takes some time, but is a closer step to reactivity, since the mission planner is not rerun and the pilot gets the feedback of the action in first place. Murphy [cf.

(Murphy)] described this as an interleave of the plan and the act component.
A last point which should be discussed is the portability of this architecture. Since this architecture is exclusively focussed on navigation tasks the reuse of the plan component code becomes more simply, because the plan a path principle persists constant, sorely the environment changes. Therefore must the knowledge base be rewritten to hand the world model beside the sensing additional information about the environment. Due to the lack of complexity portability is ensured.

5.2 NIST Real-Time Controller System Architecture (NIST RCS)

NIST stands for National Institute of Standards and Technology, this is where Albus developed the real-time controller system architecture ([Albus (c)]). Murphy described Albus' aim to develop an architecture "to serve as a guide for manufacturers who wanted to add more intelligent to their robots" [cf. (Murphy)]. This architecture was also adapted to a television version called NASREM, which will not be further mentioned in this work (for more information about NASREM see [Albus u. a.]). The real-time controller system is similar to the in section 5.1 discussed nested hierarchical controller architecture, since it uses some parts of it in its design. The main difference between these architecture is that the real-time controller system introduces a preprocessing step between the sensors and the world model. The architecture consists of four basic processing modules: the sensory processing module (SP), the world model module (WM), the value judgement module (VJ) and the behaviour generation module (BG).

Sensory processing Module

The sensory processing module receives the raw data from the sensors. Instead of directly fusion these data and integrate it into the world model the sensory processing module processes the sensor data by different algorithm depending on the area of application, such as filtering, masking, differencing, correlation, matching and pattern recognition algorithms.
The sensory processing module represents the sense component of the hierarchical sense-plan-act cycle (see figure 5.1), since it is responsible for the manipulation of the sensed data.
The output of the sensory processing is passed to the world model module which fusions the received sensor data into its knowledge database.

World Model Module and Knowledge Database

The knowledge database is included in the world model. It stores and maintains all necessary information about the environment and the world model retrieves the ones it needs in a

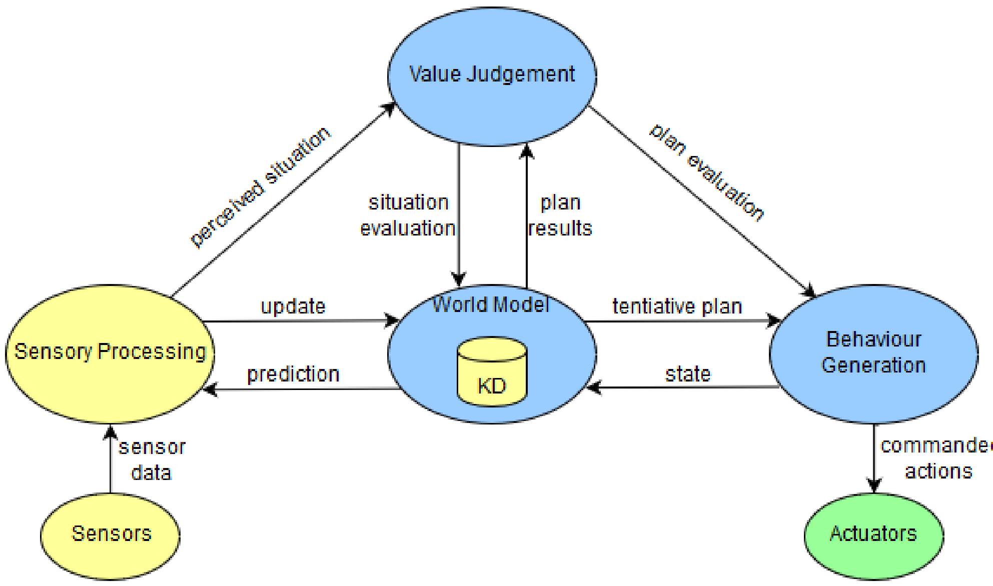


Figure 5.9: NIST Real-Time Controller System Architecture

specific scenario. That kins of information can be for example in form of images, maps, entities, states or attributes. On base of the knowledge database, the world model simulates the by the behaviour generation module proposed plan and sends the result of these simulations to the value judgement module. Moreover the world model generates predictions about the results and passes them to the sensory processing module which considers it in its algorithms. Because of the interaction between the sensory processing module and the world model module a variety of filtered and situation based data can be generated, which makes the process more efficient. In summary the world model functions as simulator and predictor and is the center of the architecture. This world model concept is partly similar to the world model of the nested hierarchical controller architecture, it also corresponds with a knowledge base and fusions it with the sensor data and acts as the base for the planning component. The difference is that this world model is more distinct and is able to simulate results and repercussions. Moreover is a robot with the real-time controller system, as distinguished from the nested hierarchical controller architecture, able to do more tasks than navigation, therefore denotes the data collection a complex diversity.
The world model module can be classified either as part of the sense component of the sense-plan-act cycle (see figure 5.1), because of the knowledge database or as plan component, since it participates at the planning process by simulating the proposed plan.

The world model module passes the plan results of its simulation to the value judgement module as already mentioned before.

Value Judgement Module

This module judges the simulation results by computing criteria like costs, risks and benefits. The value judgement module is able to evaluate the situation which it directs to the world model and which is able to consider those evaluation especially in its prediction for the sensory processing module. Moreover the value judger passes the evaluated plan back to the behaviour generator which will analyse its proposed plan by the evaluation information. To that effect facilitates the value judgement the determination of a final plan for the execution by its estimation. Due to this procedure the architecture is able to discover the most efficient possible plan. This component is not comparable to the nested hierarchical controller since that architecture does not contain a cost estimation in this discrete form.
The value judgement module is classified as part of the plan component in the standard hierarchical paradigm diagram (see figure 5.1).

Behaviour Generation Module

The last of the basic processing module is the behaviour generation module. Note that in Albus' earlier architecture designs and therefore papers this module was called "Task Decomposition" function. This function is still included as a sub-function of the behaviour generation module. Beside the task decomposition, which includes the planner, the behaviour generator contains the job assignment and the executor.
The behaviour generator module receives information about the relevant state of the environment by the world model. It also receives a command input. The command input can be a task or a goal which shall be achieved. The job assigner is the function which receives the command and passes it to the task planner function. The behaviour generator is able to decompose a task into several sub-tasks to generate recursively a plan for accomplishing the task. Therefore the planner function generates a set of possible plans. This is the part where the world model and especially the value judger take action, because the behaviour generator passes a tentative plan to them (to the world model which passes it to the value judger) to evaluate the best possible plan according to specific criteria. It receives the cost evaluation of the plan by the value judger. According to this evaluation the planner can choose the best suitable plan for execution. If the plan is chosen the planner will lead it to the executor which is equivalent to the low-level controller of the nested hierarchical controller architecture (see section 5.1). In summary the behaviour generator decides the plan based on the received task

commands and based on the current state of the environment.
The behaviour generation module represents the plan component of the hierarchical sense-plan-act cycle (see figure 5.1), since its highest responsibility is to generate a plan to accomplish an input command. The executor function represents the act component since it translates the given plan into actuator commands and passes those to the actuators.

5.2.1 Multiple Hierarchical Layers

The basic modules and their interoperation were explained in the previous sections. This knowledge is necessary in this section, in which the architecture becomes more complex by layering these components hierarchically. The architecture is expandable by typically up to six types of levels which all represent a certain functionality and which differ in many criteria such as time, detail, spatiality and scope. The levels are hierarchically ordered. A level consists of a number of nodes which consist of sensory processing, value judgement, world model and behaviour generation modules. All levels are joined by a global memory through which the representational knowledge is shared.

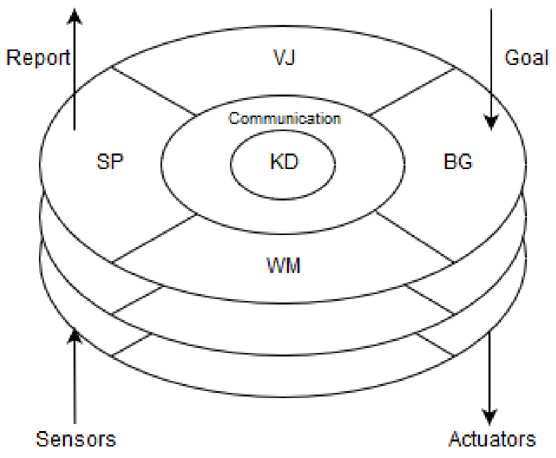


Figure 5.10: Abstract Hierarchical Layer Organisation of the NIST Real-Time Controller System Architecture

Figure 5.10 shows an abstracted view on a node. The abbreviations were explained in the beginning of the real-time controller architecture section 5.2. It is visible that the sensor data input is directed to each sensor processing module of each level and each sensor processor generates a report as output. This report consists of the sensor data and additional information including the predictions of the world model. From the lowest level, which receives the sensor

data first, since it is the closest to the sensors (see also the yellow arrows in fig. 5.11) to the highest level, the scope of the sensor information is increasing and enables the planning of future activities. The transfer of the sensor data from level to level implies that the world model module of each node works on the same knowledge basis as the other world module of the same layered nodes.
Retrievable from the table 5.1 below, in the sensory knowledge row, are the sensor views on the information of each level. The lowest for example receives points as information base for task planning, the second level combines these points and creates thereby lines, the next creates out of these lines surfaces and the level four combines these surfaces to objects and has therefore as already mentioned a more increased scope. Higher levels than level four contain as sensory information the contexts of other objects at a specific time in the nearest environment. This process, that incoming sensor data (see the yellow lines in fig. 5.11) gets passed from the lowest to the highest level, builds a graph.

Level No.	**Level Title**	**Time Period Example**	**Sensory Knowledge**
1	Servo	0.03 sec	Points
2	Primitive	0.3 sec	Lines
3	Elementary Move	3 sec	Surfaces
4	Individual	for next 30 sec	Objects
5	Group	for next 5 min	Task context for next 5 min
6	Group2	for next hour	Task context for next hour

Table 5.1: Level Overview with Time and Sensory Information

In contrast to the sensor input, the command input of the behaviour generation module starts from the highest layer and is leaded forward to the lowest level and at last to the actuators for action execution (see fig. 5.10). The behaviour generators of each level decompose the task into subtasks. The time period for computing the subtasks are shorter the lower the level is positioned (see example times in the table 5.1 above). Therefore the subtask of the lowest layer is more detailed than the highest. On the other hand the highest level is able to look-ahead and has more information about the main goal, while the lower ones simply perform obtuse task which can be used for any bigger goal. This also reflects in Albus' following declaration that "planning horizons at high levels takes longer, while planning horizons at the bottom level typically are less than 50 milliseconds" [cf. Albus (a)]. The bigger goals take longer to accomplish than the sub-goals which are sorely leading to the accomplishment of the complete goal.

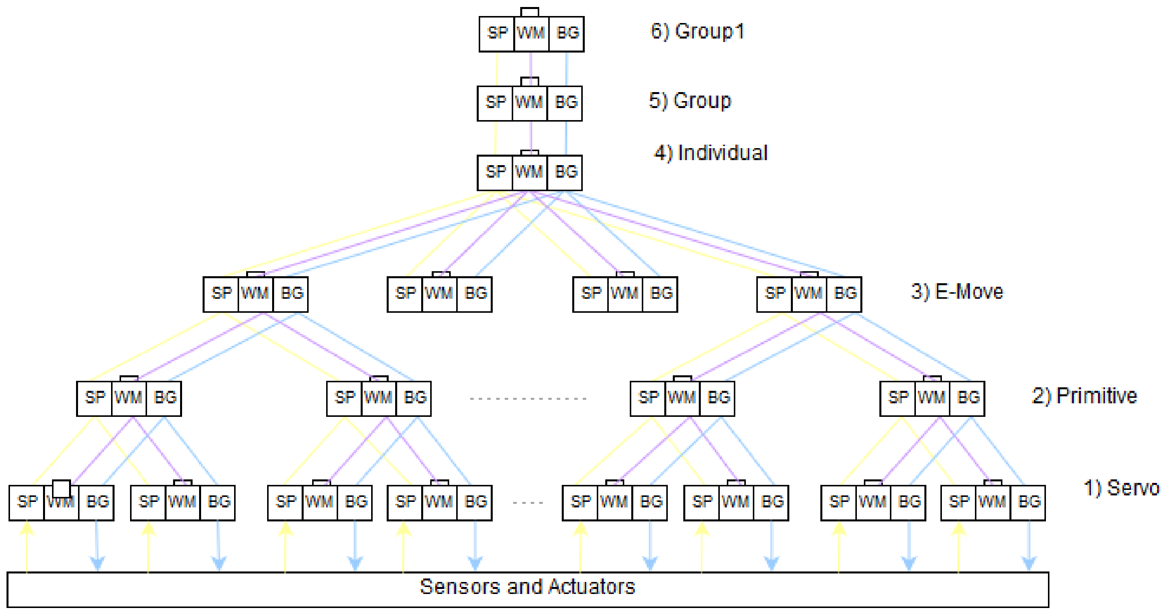


Figure 5.11: Reference Model of the NIST Real-Time controller Architecture

In figure 5.11 the plan actions are illustrated by the blue lines. Also visible is the tree structure which represents the planning process of each level of the behaviour generation module of this architecture. The highest levels (see also table 5.1 at level number 5-6) are called Groups. There are more higher groups addable to the system depending on the extent of the application. "Group2" (level 6) is responsible for the coordination for several groups and a higher "group3" would coordinate the several group2's of systems. The more level the more complexity has the system and the higher is the scope. Level 5 is the base group and coordinates the actions of a small group in the direct near of this system, to avoid future coordination problems. The title of the fourth level is depending on the task. To keep it universal it is called "individual". Its purpose is to accept the given input commands from level 5 and to convert these into suitable tasks for the subsystem which is composed of level 1,2 and 3. It splits the task as visible in the example in figure 5.11 into four sub-tasks, each with a different functionality. These four subtasks are visible in the Level 3, the elementary move level (or abbreviated e-move as in the figure). This level accepts the given tasks and computes the main plan which shall be executed to execute the task (the equivalent of the in section 5.1 described navigator of the nested hierarchical controller architecture). Level number 2 is called primitive level. It generates according to the received plan the action commands (equivalent to the pilot of the nested hierarchical controller architecture in section 5.1) for the first level. The first level is the closest level to the actuators and sensors. Like the low-level-controller of the nested hierarchical controller architecture (section 5.1) it transforms the given commands into actuator control commands. Finally the computed task gets executed by the actuators which form the basic level of the tree structure from the behaviour generation module.

5.2.2 NIST Real-Time Controller System Architecture and ROS

Represented as nodes are following modules: sensory processing, value judgement, behaviour generation and world model as well as the knowledge database, because each of these nodes are having a certain functionality as a process.
A recommendation of the implementation of the communication of the real-time controller architecture in ROS is visible in figure 5.12. This figure is simply a diagram of all communication types joined in one and that the standard notation of services and topics may differ.

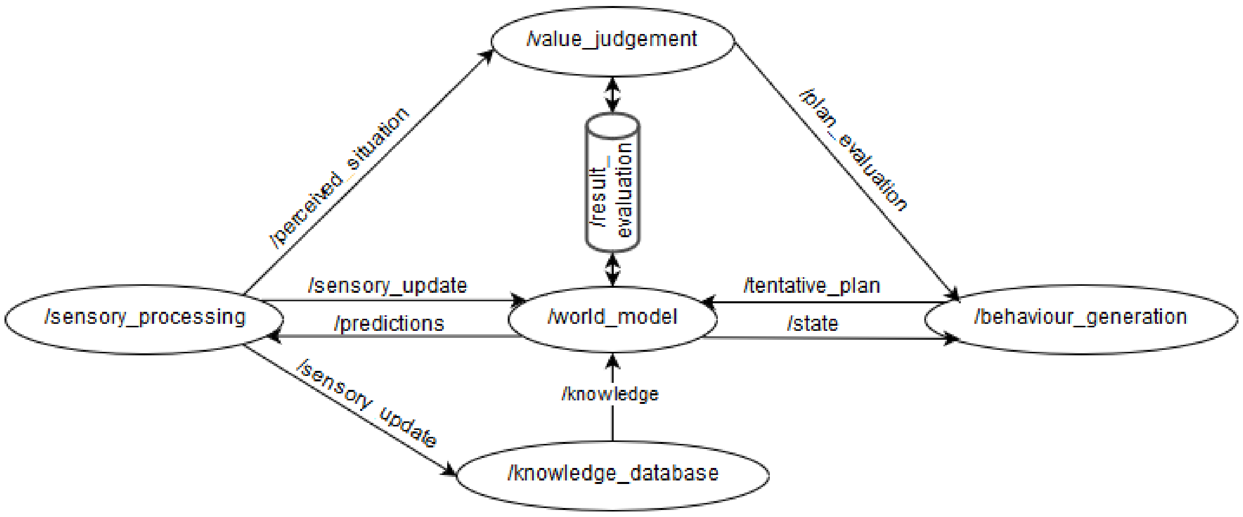


Figure 5.12: ROS Communication of the NIST Real-Time controller Architecture

The sensory processing node publishes on the topic */perceived_situation* and this topic is subscribed by the value judgement node which extracts the latest sensor data from it. This asynchronous communication is recommended because it enables the sensory processor to publish directly after it finished modifying the raw sensor data and the value judger works on the most current data. Moreover the asset that a topic is a unidirectional communication lead to the decision to a topic since exclusively the sensory processor gives information and does not receive any from the value judger.

The sensory processing node also publishes on the topic /*sensory_update*. This topic is subscribed by the world model node and the knowledge database node. The decision of two publishing to two topics depends on the internal structure. The value judgement node can also subscribe to the /*sensory_update* topic and consequently the publishing on topic /*perceived_situation* would not be necessary any more so that there is only one topic the sensor processor publishes on and which is subscribed by three nodes. As mentioned that depends on the internal structure. This decision is based on the assumption that the sensory processor uses different algorithms on the raw sensor data for the value judger than for the world model and knowledge database node. If it is using the same data for all three nodes it is recommended to publish sorely on one common /*sensory_update* topic.
It may attract attention that the knowledge database is a independent node and not included into the world model. This is also based on design decisions. The knowledge database behaves in this case more like a common database and it is more clear to plot it in that way. Moreover the knowledge database is accessible for each level in the hierarchical layered view (see 5.10) and this independent representation signalises it. The knowledge database is filled with information about the environment by the sensory processing node, combines those information with existing information and offers its knowledge by a topic /*knowledge* to the world model. The world model subscribes to it and retrieves the necessary information. Speaking again of design decisions it is possible to provide a service between world model and knowledge database instead of the topic. This may more clarify the request-response relation of a common database.
The world model node also functions as a publisher by publishing predictions, based on simulation, to the /*predictions* topic and the sensory processing node is subscribed to this topic. Another topic on which the world model acts as a publisher is the /*state* node which is subscribed by the behaviour generation node.
The communication about the situation assessment part is discussed now, it remains the communication about the planning. As mentioned before the main planning consists of three components. The behaviour generation node, which generates tentative plans and imparts them via the /*tentiave_plan* topic to the world model node. Then the world model, which passes the results of this plan to the value judgement node via a service. And finally the value judgement node, which provides the plan evaluations by publishing to the topic /*plan_evaluation*, which is subscribed by the behaviour generation node. The decision of the service between world model and value judgement node is based upon the dependency, that the world model sends the result of the plan to the value judger and requests thereby the evaluation, this evaluation

based on the plan is necessary for the world models predictions and therefore a service is more recommendable for this communication.

5.2.3 Analysis of the NIST Real-Time Controller System Architecture

The NIST real-time controller architecture has been successfully in use for applications like vehicle guidance and mining equipment. Therefore it shows that it is applicable and suggestive to use as an architecture for robot software development. In comparison to the nested hierarchical controller architecture (see section 5.1) it is much broader in its areas of possible applications. The architecture enables the developer to integrate various algorithms for different use cases. The architecture is usable to accomplish tasks such as navigation or adjustment of objects and object recognition.
Since it functions as a closed-loop control system, which means that it receives its computed output (in form of actions) also as direct input (in form of sensor data) as a feedback whether the computed result was accomplished or failed in progress which will be observed in the sensory processing module and attempts to be prevented for the next upcoming tasks or sub-tasks. This closed-loop control schema provides the architecture with robustness. Another aspect of robustness provides the world model module. Since the world model simulates proposed plans on base of current sensor data of the knowledge database before execution, it can be guaranteed that the robot reacts the way it is supposed to. Guaranteed under the constraint if we assume a close world. In an open dynamic world it is only assumable that the plan will proceed successfully, because there are several dependencies which cannot guarantee a progress without incidents. For example the plan computation and evaluation have to process in an adequate fast time as the world might change quickly and the simulation is not any more based on the latest sensor data. The robustness also gets benefit by the value judgement module which is able to evaluate not only the costs, but also the risks of a plan. Therefore the behaviour generator has a reliable feedback whether the execution is safe or not.
The evaluation process of the value judging module is also in favour of the efficiency of the architecture. Because of the evaluated tentative plans the behaviour generator is able to trade off on plan against another and is able to choose to execute the most efficient for the application. The the real-time control architecture surely full-fills the criteria of modularity, since it decomposes functionality and tasks by the hierarchical layering and modulating the planning (see section 5.2.1). The organisation of nodes enables adding and removing levels, if the necessary interfaces will be implemented. Moreover it is expandable by more layers (the groups) to make the architecture more complex.
Another advantage would be that the architecture is able to react in real-time to external

sensed events. The reaction still occurs with a delay because it must run through the planning and thereby simulation section, but it can be assumed that because of the decomposition of the tasks and the time durance of the lowest layers, the reaction happens faster than a robot based on the nested hierarchical controller architecture would.
On the other hand exactly this complex planning instance is causing an uncertainty. It is not ensured that the robot is able to avoid fast upcoming objects early enough, because especially the simulation and evaluation part is the bottleneck of the architecture and it cannot be accurately predicted if the robust avoidance plan will be executed in time. Another disadvantage is the portability. Although the architecture is expandable, it does not provide integrating a new functionality easily. The reuse for another type of application would include rewriting a lot of code.
In summary it is recommended to use this architecture if the robot should be able to do more than navigation tasks. Also if the robots task is only to navigate through its environment without any further interaction, the more complex the environment gets the more suitable is this architecture since it is able to manage complex environments and commands and still ensures functional robustness. It is also suited for the communication between multiple robots based on this architecture, since the group levels are able to foresee the actions of other robots by communication (mostly LAN, for higher group level WAN).
This architecture is based on the hierarchical paradigm and it full-fills its standard by passing sense plan act hierarchically and also the intern structure of the plan component follows the hierarchical order. But it is already a step closer to the paradigm which will be introduced in the next chapter 6 as the reactive paradigm in which decomposing behaviour by functionality is also used for reactivity.

6 Reactive Paradigm

The reactive paradigm contains only two instances: sense and act. As distinguished from the SPA-cycle of the earlier discussed deliberative paradigm (see chapter 5), this paradigm has no planning instance. This causes that there is a tight mapping between perception and action (see fig. 6.1).

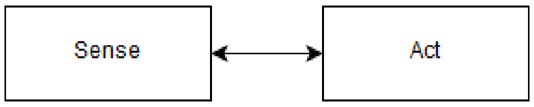


Figure 6.1: Direct coupling of sense and act

Since the sensor data input is directly mapped to the actuators output, the system is very reactive to changes in sensor data. The highest aim is to let the robot response in real time to occurrences in the environment. Speaking of environment, unlike the deliberative paradigm where the robot has its own world model, the reactive paradigm abdicates an own world model and is able to find its way in an unstructured world like the real one. This can be simply accomplished because of the direct reaction of the sensor data. Creating a world model would take too long time and the paradigm would be less timely reactive. This control by what is happening in the environment via sensor information, makes a memory unnecessary. The robot does not need to try to look ahead, because it does not plan next steps.
The expression "behaviour-based architectures" is commonly used in relation to reactive paradigms, this proceeds from the sense-act couplings which are called behaviours. Behaviours serve as basic building blocks for robotic actions. The robot is composed of multiple instances of behaviours. These behaviours are sequential or concurrent processes which all receive the same sensor data and compute their actions. Each process is unrelated to the other ones, they terminate by themselves independently. The main action executed by the robot is a combination of all behaviour outputs (see fig. 6.2).

In figure 6.2 the concurrent behaviours are shown. These behaviours can represent for example following instances: build map, explore, wander, avoid obstacles, etcetera. Each behaviour

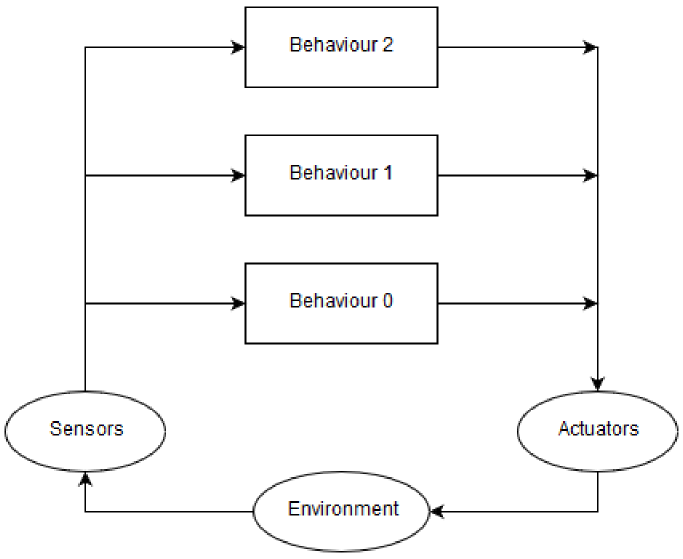


Figure 6.2: Sense-Act Couplings as Concurrent Behaviours

calculates its reaction to the same sensor data and the actuator command is a combination of the diverse behaviour outputs. This combination can be processed by different algorithms and depends on the implementation of the robot. Using behaviours causes modularity which makes the structure of the architecture not only easy to understand but it also enables to plug-in more behaviours or delete some. Hereby it causes a more dynamic and individual use of the architecture and allows an easy modification.

A further addition is necessary before leading to Rodney Brooks subsumption architecture, which represents the behaviour-based paradigm the best. The figure 6.2 of the basic process can be expanded by an arbitration instance to point out the creation of a single actuator command. This expansion is visible in figure 6.3.
The arbitration instance contains the specified algorithm which computes the final command for the robot's action of all behaviour outputs. The implementation of this arbitration depends on the use case. A possible algorithm could be a combination of all outputs or another possibility would be a priority based use of only one output. Using only one output would lead to an inhibition of the other outputs. Such an architecture is based on Rodney Brooks subsumption architecture in following chapter 6.1.

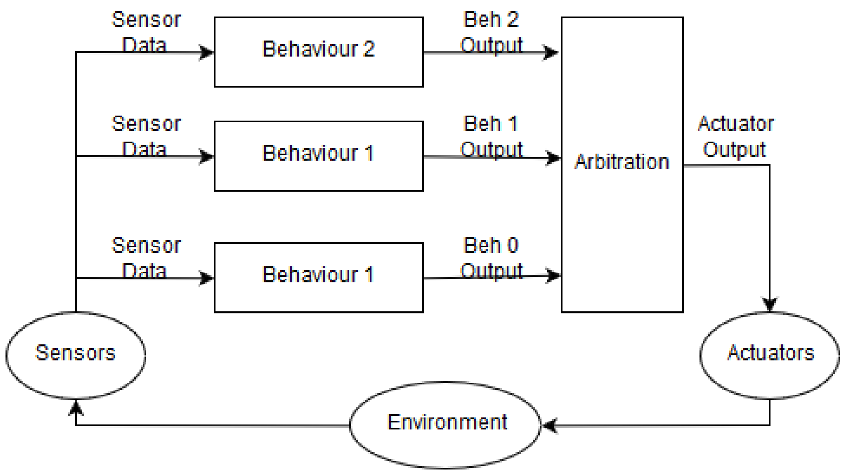


Figure 6.3: Arbitration Creates one Actuator Command

6.1 Subsumption Architecture

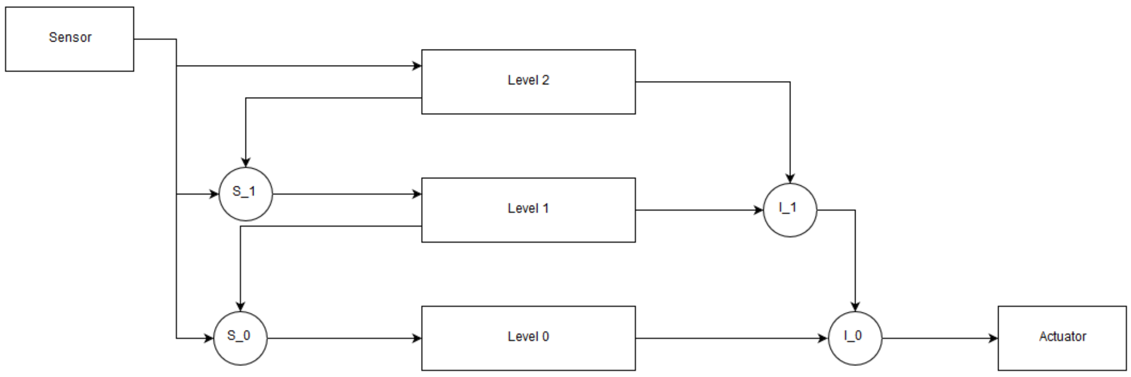


Figure 6.4: Rodney Brooks Model of Subsumption

The subsumption architecture is a layered methodology for robot control systems. It consists of coupled components and their hierarchical behaviours. One component is superimposed on the other as single layers. Layers or also called levels, in the context of Rodney brooks subsumption, represent behaviours which were introduced in chapter 6. Each layer disposes of an arbitration scheme which empowers higher layered behaviours to manipulate the input and output of lower behaviours by suppressing the input or inhibiting the output (see fig. 6.4). The inhibition of the output is a substitution of the arbitration instances in figure 6.3, therefore it is more specifically defined in the subsumption architecture. Also visible in figure 6.4 is that each layer receives data from the sensors and the actuator receives one calculated output for the control, independent from the number of registered layers (already detailed described in chapter 6 before).

6.1.1 Subsumption Architecture Implementation Analysis

The idea of Brooks' subsumption architecture is from 1986. It still has influence in nowadays robotics since many instances of robots and autonomous systems are still built using this philosophy. Today's environment puts high requirements on a reactive system. For example it has to be extremely performant, needs to be flexible and especially safe under all circumstances. The subsumption architecture is well known for being very reactive and robust. As mentioned the idea is from 1986, robot technologies are changing very fast. Not only requirements are increasing and getting more complex, but also technical tools are getting more comfortable and ensure an efficient use. Consequently it has a significance to implement the subsumption architecture according to new standards and modern views. Therefore a new implementation of the subsumption architecture will be introduced in this chapter. This implementation concentrates on the memory management by taking advantage of C++14, the modularity (therefore testability) and the use of effectively modern patterns which reduce the complexity.

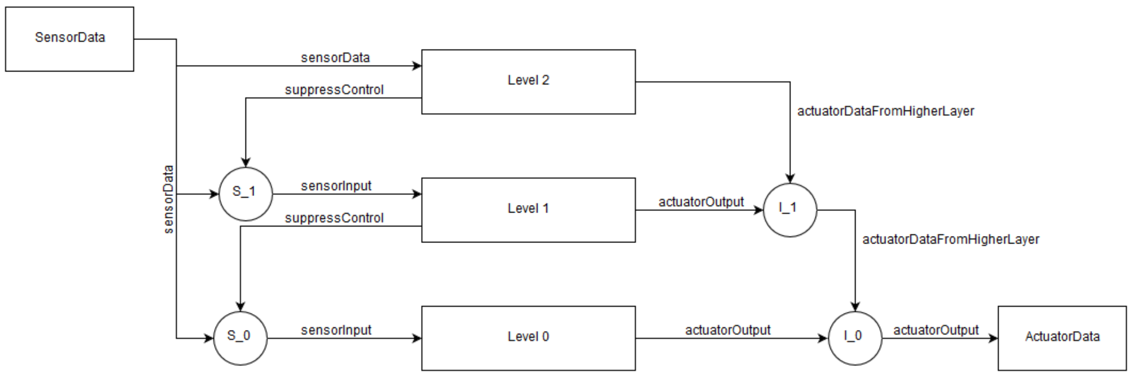
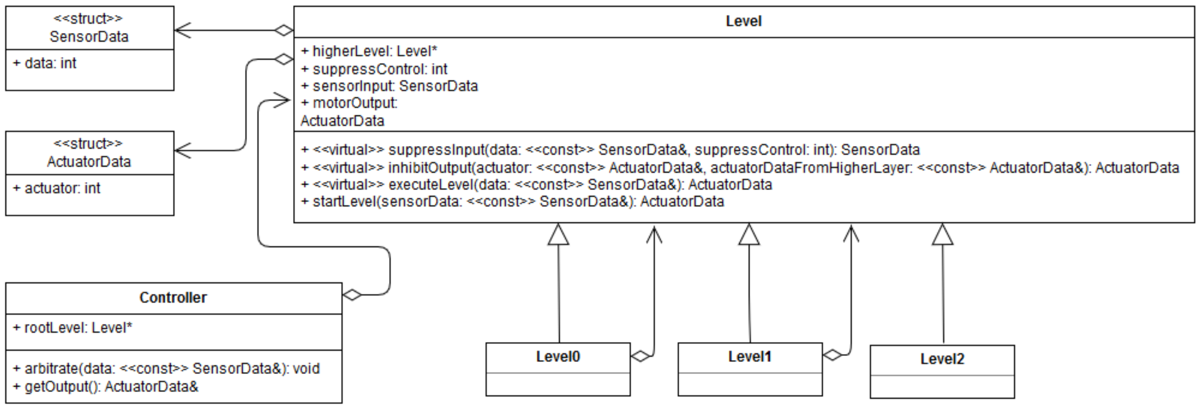


Figure 6.5: Subsumption Adjusted to Implementation

The implemented subsumption architecture is composed of different components. These components are the same as in figure 6.4, but for the following description figure 6.5 is used, since it is adjusted to the implementation. Components are the sensor, the suppressors, the levels, the inhibitor and the motor. The sensor offers the input data which it sensed from the environment. The suppressors (see fig. 6.5 S_0 and S_1) influence the input data for the current level by suppressing it by the higher layered level. The levels (see fig. 6.5 $level_0$, $level_1$, $level_2$)

include the individual behaviour. Its input is either the sensor data or the suppressed input from the higher level. It has two outputs, one is for the input of the suppressor of the lower layer and the other one calculates the level output for the motor. The inhibitor (see fig. 6.5 I_0,I_1) influences the output data from the current level by the level output from the higher layered one. The motor receives the output from the lowest level. Therefore the arbitration is based on the inhibitors. The highest levels have a higher priority than the lower ones which leads to a priority based arbitration.



<<struct>>
SensorData
+ data: int
<<struct>>
ActuatorData
+ actuator: int
Level
+ higherLevel: Level*
+ suppressControl: int
+ sensorInput: SensorData
+ motorOutput: ActuatorData
+ <<virtual>> suppressInput(data: <<const>> SensorData&, suppressControl: int): SensorData
+ <<virtual>> inhibitOutput(actuator: <<const>> ActuatorData&, actuatorDataFromHigherLayer: <<const>> ActuatorData&): ActuatorData
+ <<virtual>> executeLevel(data: <<const>> SensorData&): ActuatorData
+ startLevel(sensorData: <<const>> SensorData&): ActuatorData
Controller
+ rootLevel: Level*
+ arbitrate(data: <<const>> SensorData&): void
+ getOutput(): ActuatorData&
Level0
Level1
Level2

Figure 6.6: Class Diagram

The class diagram (see fig. 6.6) consists of four components which are related to the components of Rodney Brooks model (see fig. 6.5 and fig. 6.4). SensorData corresponds with sensor, ActuatorData with the motor, the Level with either suppressor, level and inhibitor. The fourth component, the Controller, is an additional component which manages the levels. The level component represents an interface for all levels (see fig. 6.6 $level_0$, $level_1$, $level_2$). Therefore it is expandable by a various number of those levels. New levels derive from the interface and implement its three virtual functions. Each level has its own local SensorData for the input and for the output its own local ActuatorData object. The levels are not aware of the Controller. The Controller has a pointer to the root level with which it is able to control the subsumption procedure by the composite pattern. The reason for this certain abstracted implementation is that within this structure the adding of single levels is easier than for example with an implementation in which each component is a single class. Because of the Controller the registration of the new created level became very dynamic and flexible, only one line is necessary to add the new level. There is no further changing of already existing code, therefore the other levels still function as they would without the new one which also leads to the testability. Each level can be tested individually. Consequently this ensures the functionality of the architecture and facilitates the debugging if a problem occurs.

Subsumption Code Analysis - Call Procedure

Listing 6.1: Call Procedure

```
MotorData startLevel(const SensorData& sensorData){
        if( higherLevel == nullptr){    // this is the top level
                motorOutput = executeLevel(SensorData);
        }
        else{   // this is an intermediate level
                //start the higher level at first
                higherLevel->startLevel(sensorData);
                sensorInput = suppressInput(sensorData ,
                                higherLevel->suppressControl);
                motorOutput = executeLevel(sensorInput);
                motorOutput = InhibitOutput(motorOutput ,
                                higherLevel->motorOutput);
        }
        return motorOutput;
```

The main process of the subsumption architecture is implemented in the startLevel function of the class Level(see fig. 6.1). The basic progress is from the highest level to the lowest and from left to right. Therefore the calling order for all levels except for the highest should look like this: Suppressor (see 6.1 *suppressInput*, line 36), the Level (see 6.1 *executeLevel*, line 37) and Inhibitor (see 6.1 *inhibitOutput*, line 38).

The highest level solely calls the *executeLevel* method (see 6.1, code line 31-33). Since the highest level has no higher level as consequence *higherlevel* is a null pointer. Another consequence is that neither suppressed inputs nor inhibit outputs exist. Code lines 31-33, according to the composite pattern, are for the primitive object which has not any children. The functionality about the composite design pattern is retrievable from this source [Gamma u. a.]. In lines 34-39 is the progress for all composite level objects. The procedure starts with the root level. This can be either a leaf (in case there is only one level), then the procedure will simply execute the level (see fig. 6.1 code line 32) or it can be a composite (if more than one level). If the root level is a composite, the first thing to do is calling the *startLevel* procedure for its higher level in line 35. This calls recursively all higher levels of the root level until the highest level, the leaf, is reached. From that point on the previously described function calling order is employed. In line 40 the output of the lowest level will be returned. It is the output on which each level had an influence. Because of the recursion the controller simply needs to call this function one time to provide the whole subsumption procedure.

Subsumption Code Analysis - Controller

The class *Controller* is the component which organises the registration of the levels and starts the arbitration for the actuator output. It has a pointer to the root level or to put it another way to the lowest level in the architecture, therefore it can address any registered level because of the composite pattern.

Listing 6.2: Level registration by the Controller

```
//create a linked list of levels
void registerLevel(Level* level){
        level->higherLevel = rootLevel;
        rootLevel = level;
}
```

In the function *registerLevel* (see 6.2) the attribute *higherLevel* of the new level is set as the known root level of the controller (see code line 60). And the root level pointer will be updated and receives a pointer to the new level (see code line 61). In the end all levels are linked to each other by knowing their upper level and the controller only needs a pointer to the lowest level. This data structure of a singly linked list is much more efficient than for example maintaining a list of levels, because the controller only needs to know one level and no loops are required.

Listing 6.3: Arbitration

```
void arbitrate(const SensorData){
        rootLevel->startLevel(data);
}
```

The *arbitrate* function (see 6.3) is the start method for the whole progress of arbitration. As explained the controller only needs to know the root level, it simply calls up the *startLevel* procedure which handles as in chapter 6.1.1) described the progress of the actuator output arbitration of all levels. The controller also processes the communication with the actuator for the level by offering a *getOutput* function which has the output of the subsumption architecture as return value. For this function is also only the root level necessary, because as shown in figure 6.5 all upper levels make an impact on the output of the lower levels thus the output of the root level is the output which is supposed to be used for the actuator.

Subsumption Code Analysis - Adding Levels

To create a new subsumption level which shall be added to the structure/architecture the user needs to derive from the abstract class *Level* (see fig. 6.6). For each level the following functions (visible in 6.4) denoted by *virtual*, which implies that the method is implemented at another location, must be implemented: *suppressInput* (code lines 20 - 22), *executeLevel* (code lines 23 - 25), *inhibitOutput* (code lines 26 - 28).

Listing 6.4: Virtual Functions

```
virtual SensorData suppressInput(const SensorData& data ,
                        int suppressControl){
        // every level has its own copy of data
        return data;
}
virtual MotorData inhibitOutput(const MotorData& motor ,
                        const MotorData& motorDataFromHigherLayer){
        // every level has its own copy of motor
        return motor;
}
virtual MotorData executeLevel(const SensorData& data){
         //every level has its own copy of motorData
        return motorOutput;
}
```

The function *suppressInput* represents the suppressor of the level (see fig. 6.5 S_0, S_1). The task of this function is to evaluate the input for the level (see fig. 6.5 *data*) by the influence of the suppression from the higher level (see fig. 6.5 the *suppressControl*) on the current sensor data. An explicit algorithm depends on the intentional use for the architecture and needs to be added by the user. Note that this function will not be called if the level is the top level. The top level gets directly the unsuppressed data from the sensors (see chapter 6.1.1 code line 31-33).

The function *executeLevel* includes the main algorithm for the data processing and depends on the use case. As an input argument it gets the pure data from the sensors if top level or the suppressed data from the suppressor.

The function *inhibitOutput* is similar to *suppressInput*. It represents the inhibitor of the level (see in fig. 6.5 I_0,I_1) and it evaluates the actuator output in conjunction with the current level output and the output of the higher level (see fig. 6.5 the *actuatorDataFromHigherLayer*). This function will not be called if the level is the top level. The top level sets the uninhibited data from the layer (see chapter 6.1.1 code line 31-33) directly as actuator output.

Listing 6.5: Level Registration

```
...
// register levels from highest to lowest
controller.registerLevel(new Level2);
controller.registerLevel(new Level1);
controller.registerLevel(new Level0);
...
```

To add the new level to the architecture the method *registerLevel* is used and its argument is an instance of the new level (see 6.5). The description of this function was detailed explained in chapter 6.1.1.

6.1.2 Subsumption Architecture and ROS

The communication decisions are based on figure 6.5 to mainly show how the communication of the subsumption architecture in general would be implemented in ROS, but still in regard of the architectures suggested implementation.

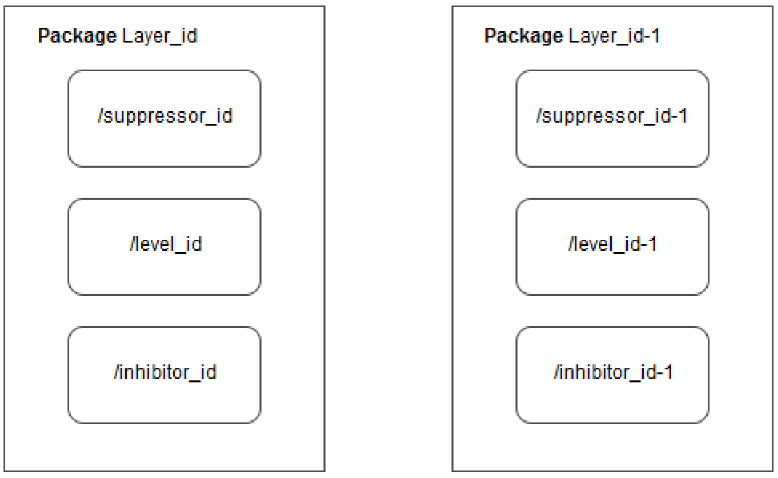


Figure 6.7: ROS File System Structure of Subsumption Architecture

The file system structure in ROS would look as visible in figure 6.7. Each layer of the subsumption architecture represents a package. Each package consists of three nodes: suppressor, level and inhibitor. All of these nodes must have a unique identification number as well as the packages. The number of packages in ROS equals the numbers of layers of the subsumption architecture. To use each package as a level enables to add or remove behaviours more simply. It is sorely necessary to complement or remove parts of the message files of the packages which represent the "neighbour" layers to which the new package shall cooperate.

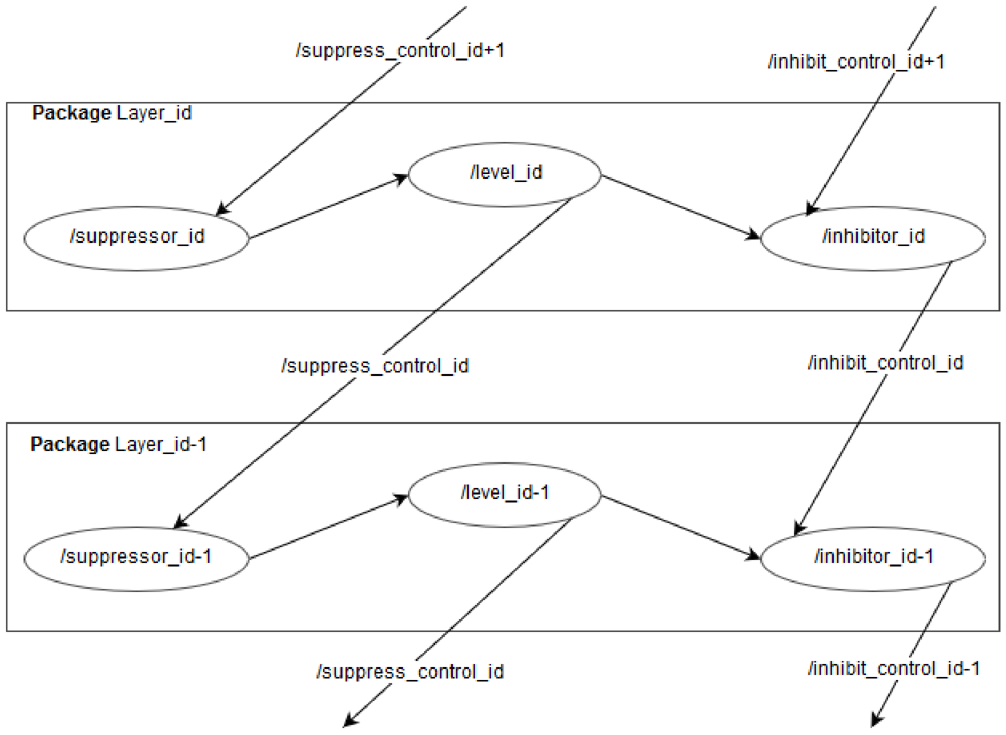


Figure 6.8: ROS Communication of the Subsumption Architecture

The ROS communication of the subsumption architecture is indicated in figure 6.8. As mentioned the package of the respective layer consists of three nodes. First the communication within the package will be analysed. The suppressor node with the identification of the respective layer number publishes the suppressed sensor input to the $/sensor\_input$ topic. The $level\_id$ node subscribes to this topic in order to receive the sensor input. The level node functions also as a publisher for the topic named $/level\_output$. Subscribed to this topic is the $/inhibitor\_id$ node.

Now following is the communication with nodes of other packages. The $level\_id$ node of the package $Layer\_id$ publishes on a topic $/suppress\_control\_id$. The $/suppressor\_id - 1$ node is the subscriber of the topic. This node is located in the package $Layer\_id - 1$. Moreover is the $/inhibitor\_id - 1$ node of this package subscribed to the $/inhibit\_control\_id$ topic of the $Layer\_id$ packages. This package itself communicates analogous to this with its lower

layer package (for example $Layer\_id - 2$).
The whole communication is based on topics because the topic communication is unidirectional and since the components do not need a bidirectional communication using topics is the best choice to implement the communication in ROS.

6.1.3 Analysis of the Subsumption Architecture

The subsumption architecture is used to realise a reactive real-time interaction with an open and therefore dynamic environment. Its usability has been demonstrated by the fact that is "have been implemented on various computational platforms in different robotic hardware structures" [cf Langton].
Modularity is supported in the subsumption architecture by the layers. Since each layer represents a specific behaviour and computes independently the individual output based on the common input. The layers are separated in functionality which is also in favour of modularity. Mainly this modularity causes the robustness of the architecture. In case of a failure of a layer, the system is able to continue running since the other layers are still able to run their functionality. The corrupted layer is also easy to debug due to modularity. In summary the subsumption architecture avoids centralized control by having self-centred autonomous and parallel processing modules of layers.
Another advantage is that the subsumption architecture is very reactive, which is the reason that it represents the reactive paradigm. In consequence of the suppressors and inhibitors the system is able to support immediate reactivity. A parallel execution of the layers increases the efficiency of the architecture.
Layers can be reused for other robotic systems if they have to implement the same behaviour. Despite this portability criterion of reusing behaviours, for the portability it is still necessary to rewrite some code. Inhibitor and suppressors algorithm changes with regard to another robot system. Also additional behaviours may be required. The basic behaviours, such as avoiding obstacles or wandering, are mostly reusable for other mobile robot systems. Summed up the portability is partly guaranteed, but still includes some new code writing.
Layers are solely able to communicate with their upper and lower layers via the inhibitors and suppressors. Therefore a combination of behaviours is impossible and often redundant code writing in the level component since it cannot access existing code as a sub-function from other levels. But the redundancy of code sections is a disadvantage which is negligibly, since the subsumption architecture is more focused on the modularity, which is only reachable with the disadvantage of redundancy.
The architecture is expandable by multiple layers. The more added layers the more complex

the architecture becomes. But since there is no official limit of extension, it is possible that the efficiency lacks by increasing complexity. Lower prioritized layers might not be able to execute their actions if there are too many higher prioritized layers suppressing them. Consequently it is important that the number of layers does not become too complex.
The reactive subsumption architecture includes no strategic planning or learning. Due to its high standards in accomplishing reactivity it simply is able to react to environmental conditions or by receiving goal inputs. An independent look-ahead planning and action selection is thereby impossible. For the integration of planning and learning processes to reactivity, the hybrid architectures are used (see chapter 7).

7 Hybrid Paradigm

The hybrid robot control paradigm is a combination of the deliberative/hierarchical paradigm and the reactive paradigm. This is also the reason why this control paradigm is also called deliberative/reactive paradigm. Since it is a combination of both paradigms it also consists of the sense, plan and act components. The organisation of theses components can be described as plan, sense-act. As visible in figure 7.1 the plan component is the one which is higher situated. Sense and act however on the same level, which symbolises the previously mentioned sense-act organisation.

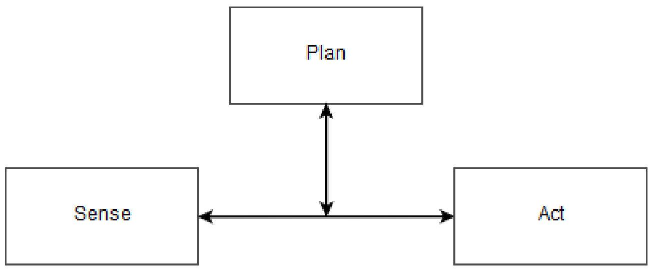


Figure 7.1: Sense, Plan, Act of the Hybrid Paradigm

The sense and the act component work directly together unlike it was the case in the deliberative control paradigm (see chapter 5, fig. 5.1). This close cooperation is based on the reactive paradigm which represents this tight coupling between perception (sense) and action (act) (see chapter 6, fig. 6.1). The deliberative paradigm is represented by the existing of the plan component. Summed up the hybrid paradigm is conceptually divided into a reactive part and a deliberative part.
Due to the reactive part the robot is able to react to environmental conditions like an emerging obstacle and is in the position to avoid it fast, on the other hand the robot is able to plan on longer-term goals and tasks. The reactive part owns the low-level control part which enables the reactivity of the architecture. It involves short time horizons which define the presents situation since this part does not contain any global knowledge. On the contrary the the deliberation component contains a global knowledge representation. It is responsible for more

complex tasks or goals, the higher level goals/tasks with increased scope of the environment. Therefore it is designed for long time horizons, which define the future actions.
The theoretical idea of the hybrid paradigm was to combine the advantages of both paradigms. In the field the experiences differ. By analysing a representative hybrid type of architecture later in this chapter, it will turned out that not only the advantages, but also the disadvantages will be combined and even disadvantages of one paradigm are able to shadow the advantages of the other one.
The question is how these parts can be combined in a architecture, especially how it is possible to combine the slow planning with the fast reactivity. The hybrid paradigm executes both the deliberative part and the reactive part at every specified architecture concurrently, otherwise it would not be reactive enough. Two basic abstracted possibilities exist for the combination of the reactive paradigm and the deliberative. The first one is visible in figure 7.2. In this case the deliberative component is integrated into the behaviour arbitration by the controller. It receives the same sensor input as the other components, updates its world model by it and starts the planning process by continuously observing the world model state. After processing its output takes part in the arbitration process like the reactive behaviours.

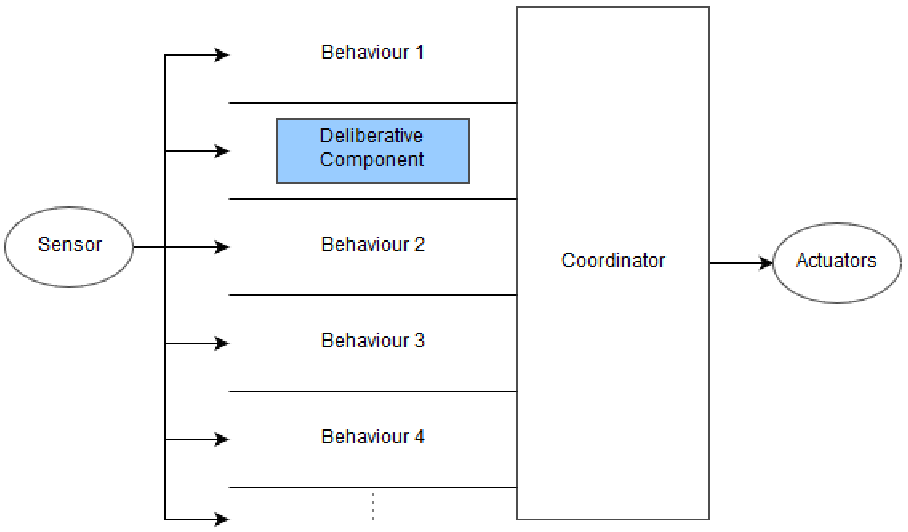


Figure 7.2: Deliberative Component Integrated into Behaviours

The other possibility is visible in figure 7.3. There is the deliberative component primarily segregated from the reactive process and functions as an activator of behaviours. Again the deliberative component and the reactive behviour components receive the same sensor data, the internal processes of the deliberation part remain the same as in the first introduced possibility. The output is transferred to a specific behaviour which is able to accomplish the output plan, this is the behaviour activation by the deliberator. Then the plan takes part at the actuator command arbitration with the other behaviours.

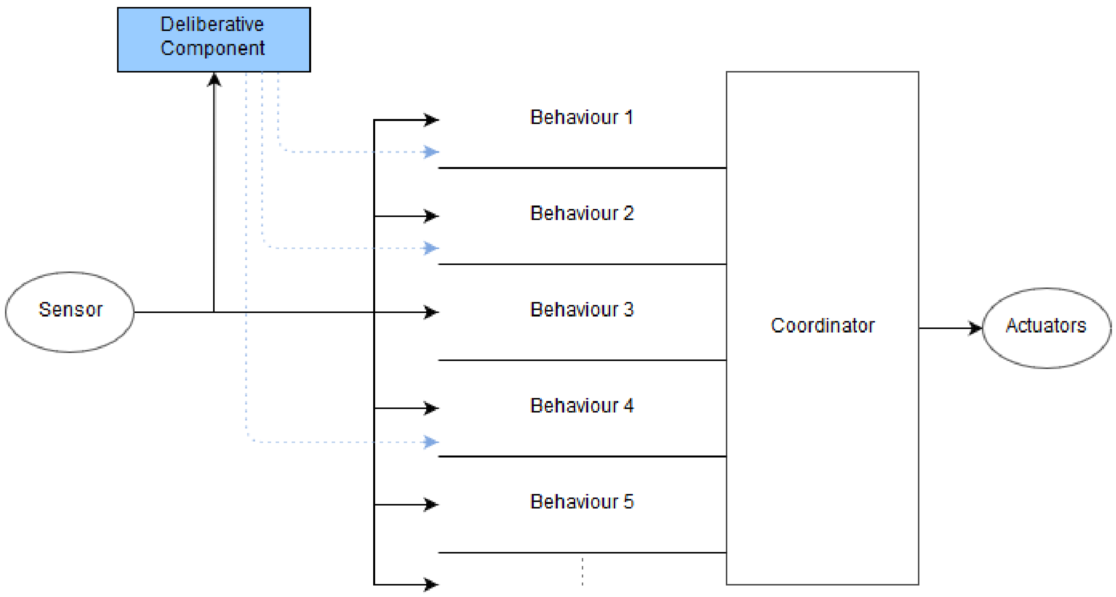


Figure 7.3: Deliberative Component Segregated from Behaviours

Those basic layouts represent an abstract way of how the paradigms can be combined. More detailed are the three different architecture designs of which nowadays hybrid robot systems are based on. These genera of architectures are called: the state-hierarchy architecture, the model-oriented architecture and the managerial architecture.
In all of the three architectures the following five basic hybrid components are integrated:

- Sequencer:
  Generates a sequence of behaviours
- Resource Manager:
  Allocates resource to behaviours, behaves like a controller
- Cartographer:
  Responsible for the maintainable of the map information, equivalent to the world model

- Mission Planner:
  Receives the command input and computes the mission

- Performance Monitor:
  Observes the planning progresses towards the goal

All these three architecture structures are represented by specific architectures. Furthermore the basic hybrid components will be identified for this architecture. The state-hierarchy architecture by the three-tiered architecture (see [Vladimir Kulyukin]), the module-oriented architecture by the task control architecture (see [Silva und H.Ekanayake] and the managerial architecture is represented by the autonomous robot architecture. This architecture will be detailed analysed in the following (see 7.1).
Since the hybrid paradigm is a combination of deliberative and reactive paradigm, this chapter includes some known and before discussed parts of those paradigms. Therefore a knowledge of those paradigms and their components is assumed and will not be explained again in detail. For following architecture the implementation of the ROS communication will not be discussed, since the similar parts it would be redundant. The only addition is the communication between the "Hierarchical Planner" and the "Motor Schema Manager", which would be realised in ROS by an service, since the feedback from the schema manger needs to be processed in time.

7.1 Managerial Architecture - Autonomous Robot Architecture (AuRA)

The managerial architecture has its name because of its composition of responsibilities which resembles a business management. A representative managerial architecture is the autonomous robot architecture (abbreviation AuRA) which was developed by Ronald C. Arkin.
The hybrid autonomous robot architecture consists of the following main components: the cartographer, perception, motor, homoeostatic control and hierarchical planner. All these components are visible in figure 7.4.

The sensors send the raw sensor data to the perception component. This component consists of multiple sensor processing modules, which are conform to the sensor processing modules of the NIST real-time controller system architecture. The sensor processor is responsible for preprocessing the raw sensor data for the cartographer and for the motor component, for more information about the sensory processing module see 5.2.
Those preprocessed data will be send to the motor and to the cartographer component. First

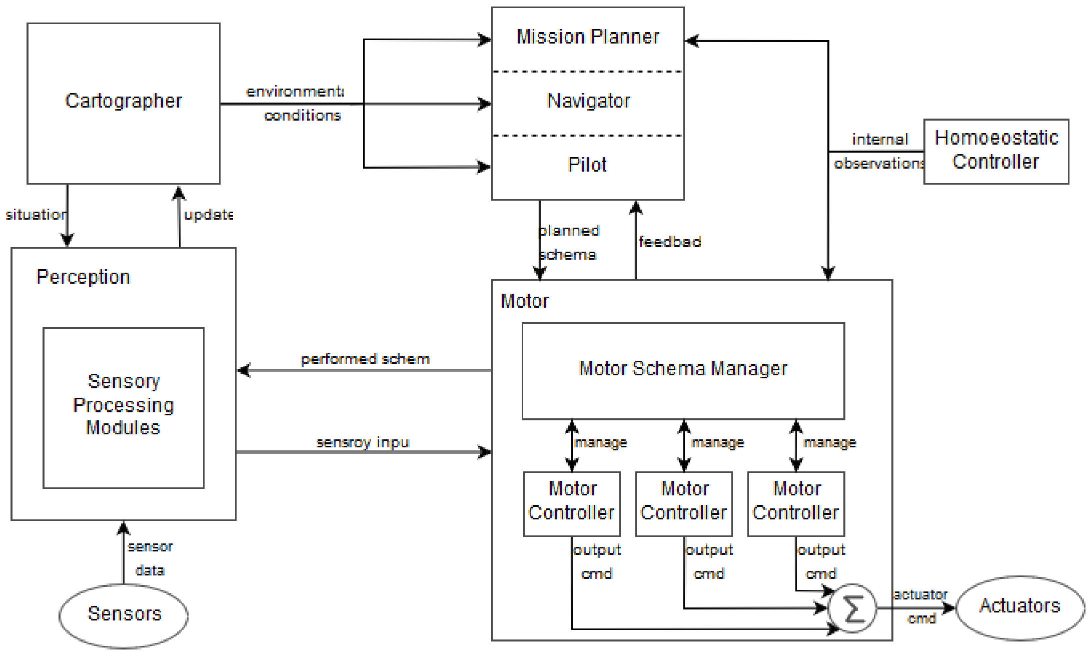


Figure 7.4: Autonomous Robot Architecture

the deliberative layer will be analysed, therefore the cartographer. The cartographer component is comparable to the in deliberative paradigms known world model. It includes the map of the environment based on the sensor processing data and other relevant information and conditions about the environment. It is responsible for maintaining and providing those information (more information see 5.2 and also 5.1).
The other deliberative component, the hierarchical planner, is based on the plan component of the nested hierarchical controller architecture (see 5.1). The planner therefore consists of a mission planner, a navigator and a pilot. The mission planner is in charge of processing the received mission under consideration of the external and internal conditions and imparts the mission goal to the navigator. It is the function of the hierarchical planner with the lowest time constraints. A difference between this mission planner and the nested hierarchical controller plan component (5.1) is that this mission planner receives three inputs. The map from the cartographer and the external command input (e.g. by a human interface), which represents the mission, are equivalent. The third input is additional and includes the current internal states and safety considerations by the homoeostatic control component.
At this point a short excursion to the homoeostatic controller component is appropriate. The homoeostatic controller gathers data for internal surveillance of the robot. It observes hazardous condition for example too high temperatures or low energy levels and is able to intervene

into the motor components processes and the mission planning, so that these factors can be considered. Since it is used for either the deliberative or the reactive layer, this component can be classified by both paradigms.
The second module of the hierarchical planning component is the navigator. Its timely constraints are increased in comparison to the mission planner. The navigators responsibilities are also equivalent to the navigator from the nested hierarchical controller architecture (5.1). Its computed path and necessary parameters are send to the pilot which tasks are similar as well to the aforementioned architecture.
The pilot focuses on a path segment of the received path and selects appropriate motor schemata, which are analogous to behaviours, to accomplish the planned goal and transfers them to the motor schema manager. The pilot is the module of the hierarchical planner with the highest requirements of temporality.
Both the navigator and the pilot represent the basic hybrid sequencer component, because they generate the sequence of schemata for accomplishing a goal. Moreover the hierarchical planner in total represents the performance monitor since it is able to observe whether the accomplishment of a goal progresses.
The schema manager is included into the motor component. It represents the hybrid basic component of the resource manager, because it is the facilitator between the deliberative and reactive layer. It is responsible for the execution of the pilots transferred plan in form of an schema and forwarding it, after processing, to the multiple motor controllers. Moreover it is responsible for the monitoring of the schema execution and is able to act upon this execution by starting and stopping it as well as introducing the planners schema. It also feedbacks the pilot if the current situation is changed and the plan must be modified to it. In this case the feedback information transferring process within the hierarchical planner is the same as in the nested hierarchical controller architecture (see 5.1). More detailed information about motor schemata and the motor schema manager can be read in [Arkin (c)].
The motor component is part of the reactive layer and consists of the aforementioned motor schema manager and also of various motor controllers. Each of these concurrent motor controllers receive the sensor data from the sensory processing modules, computes its output by the use of a schema and transforms this output into actuator commands. The in figure 7.4 visible sigma sign receives all actuator commands and arbitrates or combines them to one actuator command. This process is comparable to the reactive subsumption architecture (see 6.1).

7.1.1 Analysis of the Autonomous Robot Architecture

The autonomous robot architecture has been successfully used by many mobile robots in different areas such as buildings, outdoor and in manufacture environments. It is not only able to accomplish navigation tasks, unlike the nested hierarchical controller architecture on which the plan component is based on, but also able to accomplish manipulation tasks, like the NIST real-time controller architecture of whose organisation the sensory processing module and the extended world model (here cartographer) is retrievable. Moreover the architecture is able to react immediately to environmental conditions by the advantages of the reactive paradigm since the sensory processor input is directly linked to the motor component which consists of motor schemes which are comparable to the behaviours. By the use of priorities of these schemata a fast reaction to emergent environmental conditions is guaranteed and the motor schema manager is responsible for embedding the planning actions into the execution arbitration.
This combines mostly the advantages of the two introduced and analysed deliberative architectures (see analysts of NIST real-time-architecture 5.2.3 and analysis of the nested hierarchical controller 5.1.2). This includes the modularity of the hierarchical planning component by functional decomposition which also facilitate the time management. Also the advantages of the reactive paradigm (see 6.1.3) such as reactivity, modularity by decomposition by functionally different behaviour/schemata are advantages of the autonomous robot architecture. Unfortunately also disadvantages exist, despite the "best of the hybrid aim to combine "best of both worlds". The portability is still not ensured, since for example the behavioural structure can only partly on low-level functions be re-used for other robot use-cases. And the robustness of the reactive paradigm gets minor affected by the robustness of the deliberative paradigm, because it does not involve active simulation since it has to match up with the scheduling of the reactive part of the architecture.
In conclusion the autonomous robot architecture represents a good aim to combine the reactive and deliberative paradigm thorough the motor schema manager. Because the manager, of which the type of hybrid architecture is named, is in charge of observing the execution of the reactive behaviour, is aware of the environmental conditions by current sensor data and is able to integrate planning-based schemata into the action process without compromise the reactivity.

8 Conclusion

In the beginning of this thesis were some questions declared, mainly about how can be differentiated between the paradigms and their representative architectures as well as the question of the usability in a specific use-case. Moreover were the advantages and disadvantages of each architecture an interest.
Within the analysis of two architectures of the deliberative robot control paradigm were following results, with regard to application area and answering leading questions, extractable.
The result of the analysis of the nested hierarchical controller architecture was that this architecture is recommendable for navigation only tasks. The functional decomposition of the planning function as well as the fact that the plan component, due to this modularity, does not rerun completely, makes this architecture timely efficient. Although the hierarchical planning component communicates directly with the act component and reacts to the action feedbacks, the architecture has restrictions of real-time. The use of this architecture is recommended for robots which shall only execute navigation tasks preferably in a closed world or an open world in which fast reactivity is not essential, like for not safety critical domestic robot projects.
The other deliberative based architecture which was analysed is the NIST real-time controller system architecture. This did distinguish itself from the nested hierarchical controller architecture by its plan simulation and sensor data preprocessing parts. This architecture is more robust and resilient than the previously mentioned deliberative architecture. Moreover it is qualified for not only navigation tasks, but also manipulation tasks. Due to the time spending simulation and evaluation part this architecture is able to select the plan with the best condition, but it lacks in temporal respect. Therefore this system is suitable for robots which shall be able to perform tasks reliable and with an optimum of costs and other criteria, but are not placed in a very dynamic world, since that would decrease their rigour and reliability.
In conclusion neither the nested hierarchical controller architecture, nor the NIST real-time controller system architecture can be considered as reactive systems since they do not accomplish all characteristics (see section 2.2). The nested hierarchical controller architecture is more time efficient, but not as robust and cost optimized as the NIST real-time controller system architecture. This answers the question in the introduction whether architectures of

the same paradigm differ from their characteristic and advantages. Although they used the same hierarchical component order and represent the sense-plan-act-cycle it is still necessary to consider their characteristics carefully. This applies not only for the deliberative paradigm, but also for the both other ones. It was simply only demonstrated on this paradigm.
In general the deliberative paradigms biggest asset is the planning component which enables the system to develop autonomous actions and goal. Unfortunately is exactly this component its bottleneck, which compromises its usage in reactive environments.
The result of the analysis of the subsumption architecture revealed that, regarding the questions and the criteria, this architecture does accomplish the standards of a reactive system. It is due to the direct mapping of sensing and action highly responsive, because of the multiple concurrent behaviours. Theses behaviours also favours the resilience and robustness by being facile to debug in case of failure. They also causing functional modularity and in case of an implementation in ROS was a asynchronous communication recommended between the behaviours. The usage of this architecture is recommended for robots which shall operate in a open world. The reactive paradigm and therefore the subsumption architecture are designed to operate in a dynamic world. Due to the necessity of external stimuli by a dynamic environment the reactive paradigm would not be as efficient as a deliberative system in a closed world. This is because of the not included plan component. It is not able to develop and refine independently tasks without the environmental input.
In conclusion the decision for the usage of the reactive paradigm for a robot shall be based on the the criteria whether the robot is supposed to be highly reactive, respectively shall operate in a real-time environment. If that applies and also safety in regard to reliability is desired, then this paradigm is the right decision for the robot architecture.
The result of the analysis of the autonomous robot control architecture, which represents the hybrid paradigm, conduced that by the combination of the deliberative paradigm with the reactive paradigm, not only the advantages are combined, but also their disadvantages. Furthermore are some advantages of one paradigm affected by the disadvantages of the other. For example in the autonomous robot architecture is the robustness of the reactive component decreased by the deliebrative, because it does not simulate the plan which shall be integrated into the behaviour arbitration properly so that it is prone to failure.
The question of the introduction whether the hybrid paradigm is the ultimate solution for the design of robot architectures is thereby answered. On the other hand it is the most used and most effective architecture if the robot shall be able to be both reactive and forward-looking.

In conclusion each paradigm has its advantages and its disadvantages depending on the requirements and the use-case. In some application areas is the use of a prospective, but less reactive architecture an asset, in others the usage of a reactive, but less intelligent architecture. If both is required, which is the nowadays aim, a hybrid is the right decision.
The world is not perfect and out of this reason robots and their architectures cannot be perfect either, the only aim to pursue is to make the right design decision and to approximate perfection by continuous development.

9 Perspective

This chapter gives a perspective of potential continuation based on this thesis.
Robot control paradigms offer an amount of possibilities to be extended on base of this paper, since robot control is an diversified application area.
A possible continuation would be the approach which was included in the subsection 6.1.1 of the subsumption architecture. There was an implementation model introduced for the subsumption architecture in modern C++. Now it would be possible to pursue this approach by implementing the other architectures in use of modern and effective C++ functionalities.
Another possibility is to extend the ROS implementation of these architecture in use of different ROS packages and simulation tools. The ROSA introduction (see chapter 3), and the communication as well as the structure of the analysed architectures can be used as a basis (see subsections 5.1.1, 5.2.2 and 6.1.2).
Another project would be to specialise on artificial intelligence for robots to expand the deliberative system, and thereby also the hybrid, by useful algorithms. Another possibility for expansion would be to offer an overview of useful robot control algorithms in general.

Bibliography

[Actuators2015] : *Actuator Types: What's the Difference between Pneumatic, Hydraulic, and Electrical Actuators?*. – URL http://machinedesign.com/datasheet/what-s-difference-between-pneumatic-hydraulic-and-electrical-actuators-pdf-download. – Zugriffsdatum: 2016-12-09

[OxfordDict] : *Oxford Dictionary*. – URL https://en.oxforddictionaries.com/definition/paradigm. – Zugriffsdatum: 2016-11-17

[ROSwiki2016] : *ROS Wiki*. – URL http://wiki.ros.org/. – Zugriffsdatum: 2016-11-17

[ROSwikiCommunity2016] : *ROS Wiki Community*. – URL http://wiki.ros.org/Get%20Involved. – Zugriffsdatum: 2017-03-01

[ROSwikiGraph2014] : *ROS Wiki Graph*. – URL http://wiki.ros.org/ROS/Concepts. – Zugriffsdatum: 2016-11-17

[ROSwikiSensors2016] : *ROS Wiki Sensors*. – URL http://wiki.ros.org/Sensors. – Zugriffsdatum: 2017-03-01

[Albus a] Albus, James S.: The NIST Real-time Control System (RCS): a reference model architecture for computational intelligence, URL https://pdfs.semanticscholar.org/cfc1/8920c8df27cb4b61fc46e1b6d340cbd06c02.pdf. – Zugriffsdatum: 2016-12-28, S. 23–41

[Albus b] Albus, James S.: The NIST Real-time Control System (RCS) An Application Survey, URL http://www2.ece.ohio-state.edu/nist_rcs_lib/rcssurvey.pdf. – Zugriffsdatum: 2016-12-28

[Albus c] Albus, James S.: The NIST Real-time Control System (RCS): an approach to intelligent systems research, URL http://ws680.nist.gov/publication/get_pdf.cfm?pub_id=820528. – Zugriffsdatum: 2016-12-27, S. 157–174

[Albus u. a.] Albus, James S. ; Quintero, Richard ; Lumia, Ronald: Overview of NASREM: The NASA/NBS standard reference model for telerobot control system architecturere, URL https://www.nist.gov/sites/default/files/documents/el/isd/NASREM.pdf. – Zugriffsdatum: 2016-12-28

[Arkin a] Arkin, Ronald C.: *Behavior-Based Robotics*. MIT Press. – ISBN 978-0262011655

[Arkin b] Arkin, Ronald C.: Intelligent Robotic Systems: Editorial Introduction, URL http://www.cc.gatech.edu/ai/robot-lab/online-publications/expert-intro.pdf. – Zugriffsdatum: 2016-12-27

[Arkin c] Arkin, Ronald C.: Motor Schema Based Navigation for a Mobile Robot: An Approach to Programming by Behaviour

[Arkin und Balch] Arkin, Ronald C. ; Balch, Tucker: AuRA: principles and practice in review, URL http://www.cc.gatech.edu/ai/robot-lab/online-publications/jetai-final.pdf

[Arkin und Mackenzie] Arkin, Ronald C. ; Mackenzie, Douglas C.: Planning to Behave: A hybrid Deliberative / Reactive Robot Control Architecture for Mobile Manipulation, URL http://www.cc.gatech.edu/ai/robot-lab/online-publications/ISRMA94.pdf. – Zugriffsdatum: 2016-12-27

[Arkin u. a.] Arkin, Ronald C. ; Riseman, Edward M. ; Hanson, Allen R.: AuRA: An Architecture for Vision-Based Robot Navigation, URL https://web.cs.umass.edu/publication/docs/1988/UM-CS-1988-007.pdf. – Zugriffsdatum: 2016-12-27

[Bekey] Bekey, George A.: *Autonomous Robots: From Biological Inspiration to Implementation and Control*. A Bradford Book. – ISBN 9780262025782

[Boner u. a.] Boner, Jonas ; Farley, Dave ; Kuhn, Roland ; Thompson, Martin: The Reactive Manifesto, URL http://www.reactivemanifesto.org/. – Zugriffsdatum: 2016-10-12

[Brooks a] Brooks, Rodney A.: How to Build Complete Creatures Rather than Isolated Cognitive Simulators

[Brooks b] Brooks, Rodney A.: A robust layered control system for a mobile robot, URL http://people.csail.mit.edu/brooks/papers/AIM-864.pdf. – Zugriffsdatum: 2016-12-27

[Butler u. a.] Butler, G. ; Gantchev, A. ; Grogono, P.: Object-oriented design of the subsumption architecture, URL http://users.encs.concordia.ca/~gregb/home/PDF/bgg-spe2001.pdf. – Zugriffsdatum: 2016-12-27

[Cormen u. a.] Cormen, Thomas H. ; Leiserson, Charles E. ; Rivest, Ronald L. ; Stein, Clifford: *Introduction to Algorithms*. 03. The MIT Press. – 595–601 S. – ISBN 9780262033848

[Gamma u. a.] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Design Patterns*. – ISBN 3826697006

[Huag und Messina] Huag, HUI-Ming ; Messina, Elena: NIST – RCS and Object – Oriented Methodologies of Software Engineering: A Conceptual Comparision, URL http://www2.ece.ohio-state.edu/~passino/RCSweb/rcsisd96_9doc.pdf. – Zugriffsdatum: 2016-12-27

[Langton] Langton, Christopher G.: *Artificial Life: An Overview*. MIT Press. – ISBN 978-0262621120

[Meystel] Meystel, A.: Nested hierarchical controller with partial autonomy, URL https://ntrs.nasa.gov/archive/nasa/casi.ntrs.nasa.gov/19890017110.pdf. – Zugriffsdatum: 2017-01-02, S. 251 – 270

[Murphy] Murphy, Robin R.: *Introduction to AI Robotics*. MIT Press. – ISBN 978-0-262-13383-8

[Quintero und Barbera] Quintero, Richard ; Barbera, A.J.: A Real-Time Control System Methodology for Developing Intelligent Control Systems, URL https://www.nist.gov/sites/default/files/documents/el/isd/ks/RCS_Methodology.pdf. – Zugriffsdatum: 2016-12-27

[Silva und H.Ekanayake] Silva, L. D. ; H.Ekanayake: Behavior-based Robotics And The Reactive Paradigm A Survey

[Simpson u. a.] Simpson, Jonathan ; Jacobsen, Chrisitan L. ; Jadud, Matthew C.: Mobile Robot Control - The Subsumption Architecture and occam-pi, URL ftp://ftp.cs.kent.ac.uk/people/staff/phw/.old-1999/tmp/CPA-225-Simpson.pdf. – Zugriffsdatum: 2016-12-27, S. 225–236

[Vladimir Kulyukin] Vladimir Kulyukin, Adam S.: Instruction and Action in the Three-Tiered Robot Architecture, URL http://facweb.cs.depaul.edu/asteele/Research/Papers/Steele_ISRA2002.pdf. – Zugriffsdatum: 2017-01-04

[Wieringa] Wieringa, Roel: *Design Methods for Reactive Systems*. 01. – ISBN 9781558607552

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 5. Januar 2017 Milena Hippler