Stefan Loigge, BSc

Unified and Dependable Robot Control Architecture based on ROS

MASTER'S THESIS

to achieve the university degree of

Diplom-Ingenieur

Master's degree programme: Telematics

submitted to

Graz University of Technology

Supervisor

Ass.Prof. Dipl.-Ing. Dr.techn. Gerald Steinbauer

Institute for Software Technology

Graz, September 2016

This document is set in Palatino, compiled with pdfLaTeX2e and `Biber`.

The LaTeX template from Karl Voit is based on KOMA script and can be found online: https://github.com/novoid/LaTeX-KOMA-template

AFFIDAVIT

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis dissertation.

Graz,	________________	________________________
	Date	Signature

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe. Das in TUGRAZonline hochgeladene Textdokument ist mit der vorliegenden Masterarbeit identisch.

Graz,	________________	________________________
	Datum	Unterschrift

Abstract

In this thesis an universal software architecture for autonomous robots with the ability to observe, diagnose, and repair software components is presented. In terms of autonomy the robot system should be able to operate autonomously as long as possible without human interventions. Therefore, the system must be able to detect faults, including changes in the environment and wrong behaviors of components of the robot. Goals for the robot are given by human or a central server. The goals are used by a planner to generate a plan to achieve that goals. The maintenance also includes the functionality to generate additional goals by its own in order to maintain the autonomy. The resulting plan contains different tasks, which are behaviors of the robot. Each task requires a system configuration which is related to different software components that need to be configurated before the task is executable. The execution of the task itself is supervised by the task executer.

The second important part of this work is the monitoring, diagnosis, and repair of components at runtime. Observers are used to monitor relevant aspects of the robot. Therefore, several observers are started with each running component based on their properties. The results from the different observers are collected by the diagnosis engine. In combination with the diagnosis model which is updated with each start and stop of a component, the diagnosis engine is able to identify which components are faulty. But this is only necessary, if some observers report inconsistent observations. For repairing a system, the components that are identified as faulty are restarted, which is supervised by the task executer. For repairing the task executer has also to care about the dependencies between components. Furthermore, the task executer has to deal with components that stay faulty even after a restart. The task executer has to replace the faulty component with an alternative one or cancel the execution of the related tasks if no alternative exists.

Finally, the proposed architecture including the observation, diagnosis, and repair was realized as a proof-of-concept implementation. Additionally, an evaluation with an use case and different scenarios demonstrate the mentioned skills.

Acknowledgements

I would first like to thank my supervisor, Gerald Steinbauer, for his support for this master thesis and nearly my entire study. We had a lot of interesting discussions on this work but also on other robotic related topics. His office door was always open whenever I ran into a trouble spot.

I would also like to thank the company *incubedIT* for funding this thesis.

Further I would express my gratitude to my friends and colleagues from the RoboCup Rescue Team *TEDUSAR* for their time and support. Especially Clemens Mühlbacher, which was also my co-supervisor. He invested hours in interesting discussions about all topics concerning robotics. Also his support in implementation details was really helpful.

Finally I would like to thank my family for their continuous encouragement throughout my study and my whole life. Without their support many of my projects would not have been possible and many of my goals would not have been reachable for me.

Thank you.

Stefan Loigge
Graz, Austria, September 2016

Contents

List of Figures

1 Introduction

This thesis presents a new approach for an universal software architecture for autonomous robots with the ability of monitoring, diagnosis, and self-repair.

Robots are used since decades to support humans or do operations dangerous for humans. They are used at production lines for cars or consumer electronics to increase the productivity or do work with accuracy humans not able to. These robots are very inflexible. They are build for a very defined and static environment and cannot react to uncertainties if they can sense their environment at all.

In the last years increasingly autonomous robots were developed. They are used to do more complex tasks. For example, they can work in logistics environments by carrying objects. This is possible because these robots are using many different sensors to sense their environment. Also the computing power of small computers increases in the last few years. This makes it possible to interpret the sensor data, do complex actions, and estimate behaviors.

These complex systems need also to be stable for a long time to guarantee autonomy. For this, components of the robot need to be observed which represents the current state of the robot. If the system is in a forbidden or impossible state, the system needs to react. This can be the case if components are faulty or the environment has changed unpredictably. Therefore, a diagnosis need to be calculated to find the faulty components. With this information it is maybe possible to repair the them. Even if the robot is not able repair the faulty part, it can call for help.

1.1 Motivation

Many of the todays robots are used for complex tasks which claims for some self organization skills This means they have to sense their environment, interpret all the inputs, and need to solve complex reasoning problems. All this increases the complexity of autonomous robots in hardware and in software. Many different parts can be faulty and can prevent the robot from doing its job. Thus autonomous robots need the ability for self monitoring, diagnosis, and repair.

Defining an universal software architecture for autonomous robots can simplify the development and reduce the maintenance costs. Even though these robots have their own specific working environments and capabilities, the architecture design reduces all software components and interactions to the bare essentials. This design pattern for software components offers the possibility for observation and diagnosis working in the background.

1.2 Goals and Challenges

In the following the goals and challenges are presented. The main aspect is to define a general architecture for autonomous robots which is system independent and can provide the described capabilities above. It should provide the possibilities to use it for new and already running robots.

The observation need to be done at different levels of the hierarchy of the robot system, see Figure 1.1. It is necessary to find the possible faulty components and to find possible solutions to solve the problem. The most promising solution is taken first. If the problem cannot be solved in the current layer the error handling escalates upwards the hierarchy. This structure looks similar to the TDL (task description language) [1] which describes a concept for

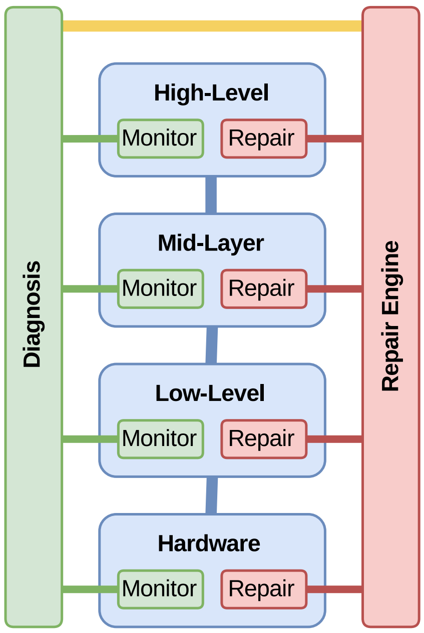


Figure 1.1: Overview of the different levels of the architecture.

handling exceptions in a layered architecture. Section 3.1 contains more information about TDL.

It is possible that faulty components are not repairable. Therefore, an alternative way to achieve the goals needs to be found. Maybe there exists redundant components that can do the same work as the faulty ones even if the quality of service degrades.

Detect faulty behaviors To detect errors, it is necessary for the robot to know how the system should be acting in precise situations. This can concern hardware components, which can be defect, polluted, blocked, or it is just a crashed driver. Therefore, different properties of components need to be monitored. This can be properties of the component itself like memory consummation, but also observations of output data of the components like frequency or content of the output.

Another possible scenario is that observed components seems to be working correctly, but combining the output data causes a conflict. For example, if the movement of a robot is measured with different sensors, but the data are different because the robot skids on a slippery surface.

Find faulty components and search for solutions Faulty properties are detected, but to guarantee a long autonomous and reliable operation time, the robot has to determine which components causes these faults. If components are continuously estimated as faulty the robot has to use redundant components if one exists. The robot need knowledge about its components and how they are connected among each other. Only with such knowledge the robot is able to repair faulty and depending components.

Use an universal architecture An universal architecture is required to supervise, monitor, diagnose, and repair software components of the robot. Therefore, the architecture includes several parts for planning, resource management, and execution are necessary.

Controlling different software components is only possible, if all them use a specified design pattern, which is also known by the architecture. Hence the management parts of the architecture can interact with the software components over a defined interface. Additional, the reusability of software component can be increased.

1.3 Contribution

This thesis contributes a general concept for an universal architecture for autonomous robots, which is easy to manage and expand. Each software module, no matter whether is is a mid-layer or low-level module, is well specified. With this specification it is possible to add further software modules, which are automatically usable by the robot. New skills can be added, which expand the capabilities of the robot. Also the reliability can be improved by adding modules similar to already existing modules.

Another improvement of this universal architecture is its observation and diagnosis capabilities, which work in the background and are transparent for the software modules. New added software modules need further description to define what can be observed and how the diagnosis has to interpret it. This is also required for all existing software modules. With this little more information each software module is known and can be processed by the observer manager, the diagnosis engine, and the rule engine.

Parts of this thesis are also published in [2] which presents an approach for supervising of hardware, software and behavior of autonomous industrial transport robots. The theoretical part of this paper includes the observers, the diagnosis engine, and the rule-based repair engine which were mostly designed and implemented in course of this work. In addition, the paper presents a practical part with a real world use case, where different fails are injected during runtime and the faults had to be detected by the system of the robot.

1.4 Outline

The following sections are arranged as follows: In Chapter 2 the problem formalization is presented. Chapters 3 discusses the related research. This includes research topics about different architectures, knowledge bases for providing, collecting, and storing all kind of information, and different works about agent control. Prerequisites are discussed in Chapter 4. This includes, besides the used frameworks, the used method of consistency based diagnosis. In Chapter 5 the architecture concept is described. This includes the architecture design with its high-level, mid-layer, low-level, and knowledge base. This Chapter also outlines the different observers for monitoring, the diagnosis engine, and a concept for fault handling. The next Chapter 6 gives some details of the implementation including class diagrams and explanations for observer and diagnosis engine configurations. Chapter 7 presents different use cases. The finally Chapter 8 includes the conclusion of this thesis, possible improvements, and the further work.

2 Problem Formulation

A robot has to achieve a set of given goals. A goal $\mathcal{G}$ is defined as a temporal database $\mathcal{G} \colon = \Phi$ as defined in [3, Page 311]. Moreover, a temporal qualified expression *tqe* [3, Page 311] is defined, which has the form $p(\zeta_i, \ldots, \zeta_k)@[t_s, t_e)$ where p is a flexible relation, $\zeta_i, \ldots, \zeta_k$ are variables of objects or constants. The temporal variables t_s and t_e assert that $\forall t, t_s < t < t_e$ the relation p is holding. A temporal database Φ is defined as a pair $\Phi = (\mathcal{Z}, \mathcal{C})$ where $\mathcal{Z}$ is a set of *tqe*s and $\mathcal{C}$ is a set of constraints for objects and timings.

The composition of a set of goals $\{\mathcal{G}_1, \ldots, \mathcal{G}_m\}$ is defined again as temporal database $\mathbb{G} := (\mathcal{Z}, \mathcal{C})$, with a conjunction of *tqe*s and constraints of the defined $\mathcal{G}_i$:

$$\mathcal{Z} \colon = \bigcup_{i=1\ldots m} \mathcal{Z}_i \qquad\qquad \mathcal{C} \colon = \bigcup_{i=1\ldots m} \mathcal{C}_i.$$

A task $\mathcal{T}$ is a tuple $\mathcal{T} = (\mathit{name}(\mathcal{T}), \mathit{precond}(\mathcal{T}), \mathit{effects}(\mathcal{T}), \mathit{const}(\mathcal{T}))$, where $\mathit{name}(\mathcal{T})$, $\mathit{precond}(\mathcal{T})$, $\mathit{effects}(\mathcal{T})$, and $\mathit{const}(\mathcal{T})$ are defined similar to operators as in [3, Page 314]:

- $\mathit{name}(\mathcal{T})$ is an expression $\mathcal{T}(x_1, \ldots, x_k, t_s, t_e)$, where $\mathcal{T}$ is an operator symbol, $x_1, \ldots, x_k$ are object variables used in $\mathcal{T}$ together with temporal variables in $\mathit{const}(\mathcal{T})$.
- $\mathit{precond}(\mathcal{T})$ and $\mathit{effects}(\mathcal{T})$ are *tqe*s.
- $\mathit{const}(\mathcal{T})$ is a conjunction of constraints.

A plan $\pi = \{a_1, \ldots, a_k\}$ is defined as a set of actions and each action is a partial instance of a task, as described in [3, Page 320]. The planning algorithm has to produce a plan that meet all the given goals based on $\Phi_0 = (\mathcal{Z}, \mathcal{C})$, which is a temporal database describing the initial state.

$\mathcal{F} = \mathcal{F}_S \cup \mathcal{F}_{NS}$ describes a set of all possible functionalities of a system, where $\mathcal{F}_S = \{r_1, \ldots, r_m\}$ defines a set of shareable functionalities and $\mathcal{F}_{NS} = \{r_1, \ldots, r_m\}$ defines a set of non-shareable functionalities. Furthermore $\mathcal{F}_S \cap \mathcal{F}_{NS} = \emptyset$ are disjoint sets. A task $\mathcal{T}_i$ requires a set of functionalities $\mathcal{N}_i \subseteq \mathcal{F}$. With the limitations of the functionalities contraints are added to the tasks which further restricts the plan.

To get a fully instantiated plan it is necessary to ground all variables of the plan.

Each task $\mathcal{T}_i$ is mapped to a behavior $\mathcal{B}_j$, where $\mathcal{B}_j$ has the same variables as $\mathcal{T}_i$ and represents a Teleo-Reactive (TR) program [4].

3 Related Research

This chapter contains different discussions about current available literature related to this thesis. The research topics and works are grouped to four different sections. Section 3.1 is about similar architectures for autonomous robots that uses a layered design. Then different approaches of knowledge processing are discussed. The third section contains some research work on agent control. Finally, literature dealing with monitoring and diagnosis is discussed.

3.1 Architecture

Laas Alami, et al. [5] describe an architecture for mobile robots for planning tasks with temporal and domain constraints and for controlling the execution of task corresponding actions in real-time. The architecture is also able to react to possible events, which can be changes of the environment or properties of the robot itself.

The described architecture comprises three layers: a decision level, an execution control level, and a functional level. The functional level is used for the basic low-level function processing like control loops. It is used for all the basic actions and perception capacities of the robot. This level is close to the hardware of the robot because of its interaction with sensors and actors. Also the monitoring of events is done by this level. Functionalities are embedded into modules which can perform specific tasks. These modules follow a standardized module structure and are modeled with G^{en}oM[6].

The second level is called execution control level and is used as mid-layer. It is used as interface between the abstract high-level and the low-level which works with concrete actions. Depending on the task requirements this level controls and coordinates the of low-level modules. It should be noted that this level is a purely reactive system without planning capability. It gets sequences of actions that should be executed from the decision level and selects, configure, and synchronizes necessary functions of the functional level.

The decision level is the last of the three described layers. It represents the high-level, which need knowledge about the tasks and the execution context to generate the task plan. The execution of the plan is also supervised by the decision level by using PRS [7].

Another work from Lemai and Ingrand [8], forces on interleaving temporal planning and execution in robotics domain. They describe an architecture, which has to perform in dynamic environments with limited resource capabilities. Therefore, the generation of plans has to respect deadlines and resource constraints. Also mechanisms are described, for adapting the plan in terms of execution timings while plan execution.

A big benefit of this layered structure is that the decision level does not need to care about properties of other layers, like basic actions and perception capacities of the robot. This thesis also uses a similar layered structure to benefit from the same advantages.

TDL As mentioned in the introduction, Simmons and Apfelbaum [1] defined TDL. It is a language for which supports the coordination of actions, environment monitoring, and exception handling. This is done by representing tasks as task trees, which are hierarchical decompositions of tasks into subtasks. In addition, task trees also include constraints for task synchronization and links different exception handlers to nodes. Each node of the tree represents an action which performance can succeed or fail. To that nodes can also be used for monitoring, which trigger an event if conditions are fulfilled. Exceptions (thrown by actions) are used as description of a fail. The system searches for an exception handler up the task tree that can handle the exception. Exception handlers are able to add new nodes to the tree or terminate existing ones. If the handler is unable to handle the exception, the system continuous the search for exception handlers. This procedure allows a local repair without affecting the global task tree.

fault-tolerant robot architecture Crestani, et al. [9] presents an approach for an architecture for fault-tolerance autonomous robots. It is designed for robots working in dynamic environments and handling faulty or unforeseen situations. This is a two-layered architecture including a decisional and an executive layer. In the executive layer they uses the technique of FMECA (failure modes, effects and criticality analysis) [10]. If a module is faulty a detection module, which accompanies the module, reports a fault signature. Fault signatures are mapped to dedicate faults by using an incidence matrix. They also use a database to store all module stated, which are permanently updated. This database is accessible for all layers and is necessary for dealing with faults. Repairing is done by recovery actions of recovery modules managed by the recovery supervisor. Among others, these actions are able to reconfigure modules, make autonomy adjustment, or even stop the mission.

3.2 Knowledge-Base

CRAM The work of Beetz, et al. [11] focuses on a cognitive-enabled middle-ware for higher-level capabilities like learning and knowledge processing for better action planning. It is named Cognitive Robot Abstract Machine (CRAM). The resulting control programs of the robot are more flexible, reliable, and efficient. They are using lightweight reasoning mechanisms which are inferring control decision. Hence it is not required to preprogram decisions.

This toolbox contains mainly two parts, the CPL (CRAM Plan Language) and a knowledge processing system. For CPL they use no new plan language, but extended the Common Lisp language, so they were able to use existing Lisp compilers.

As knowledge processing system they use KnowRob.

KnowRob Another work of Beetz, et al. [12, 13] is about knowledge processing for cognitive robots, in short KnowRob. It describes how knowledge can be processed and how it can maybe used with robots to give them the needed knowledge for everyday tasks. They had to solve some challenging tasks like grounding the information, representing uncertain knowledge, acquiring knowledge for everyday tasks, finding suitable expressive representations which allows fast reasoning, predicting effects of actions, and interacting with humans while interpreting their actions.

In their knowledge base many different types of information are included. On one hand they include commonsense knowledge, for instance a cup can break if the grasping force is too high or coffee may be spilled if the cup is moved too fast or is turned over. This type of information is very important even if it is obvious for humans. On the other hand, an encyclopedic knowledge like Wikipedia is used. Hence they have to use different knowledge representation techniques to store information, for example positions, ontologies, but also procedural knowledge.

Knowledge is acquired from several sources:

Internet Informations from the Internet can be extracted [14]. Therefore, natural language need to be interpreted.

Natural language A syntax tree is generated, with known actions, involved objects, and additional information about location, time constraints and much more.

Observations Generate knowledge from observing humans [15].

Exchange knowledge Providing knowledge for other robots, like the RoboEarth project [16]. KnowRob is a central part of that Web-based knowledge providing system.

RoboEarth The Web-based knowledge providing system RoboEarth presented by Beetz, et al. [16] explore a knowledge base shared for robots. It is a common representation for robots and provides ontologies for data, skills, objects, actions, gasping, models of objects and much more.

3.3 Agent Control

IndiGolog In the work of De Giacomo, et al. [17] a high-level programming language called IndiGolog is described. IndiGolog is an extension for Golog [18], which is a logical programming language. Golog works off-line, which is one disadvantage. This extension provides nondeterminism, supports sensing actions, and programs are executed on-line. Therefore, it can be used for planning while operating of autonomous agents, which need to sense their environment.

Golog uses the situation calculus [19] which is a first order language for representing dynamic domains and reasoning about effects of action. Reasoning is used by the interpreter to generate a plan, which is legally executable.

IndiGolog has also some interesting features like the support for parallel execution of actions and that it can make non-deterministic decisions.

PRS The following paper presents PRS which is high-level supervisor and control language. PRS strands for procedural reasoning system and is already presented in 1996 by Ingrand, et al. [7]. PRS is not a planner for high-level tasks but it is for execution of predefined plans, while making decisions at runtime and checking for policies. Even if PRS is no planner it is often referenced as reactive planner. PRS is based on the model of Belief-Desire-Intention (BDI) [20].

A goal is a description of a state the system should reach. The PRS kernel has to find and execute sequences of actions to reach the given goal.

The main elements of kernel are:

database This is the system view of the world, which is automatically updated. The Database include not only symbolic information but also numeric information like robot position.

library of plans These are sequences of actions that achieve certain goals under certain conditions. Because PRS is no planner it does not plan with actions, it searches for plans that can perform the given task.

task graph The last part is used for monitoring. It keeps track of the currently running tasks and the state of execution.

Because each action-sequences achieve a certain goal it may be necessary to add subgoals to achieve the primary goal. This subgoals are used as gain to find other possible sequences of actions.

Teleo-Reactive Programs In [4] Nilson presented Teleo-Reactive Programs. This is a formalism for computing and organizing of actions which are used for autonomous agents operating in a dynamic environment.

Each Teleo-Reactive (T-R) program contains a sequence of actions, where all actions direct the agent towards a defined goal. The reactive part selects the actions, hence the current, perceived state of the environment has to be sensed.

As a simple working example an ordered set of rules is given. If a condition is hold the corresponding action can be executed, see Figure 3.1. Lets assume at the beginning that K_G and K_A are not *true* but K_B, see Figure 3.2. The rule list is scanned in top-down order as long as the conditions of the selected rule line does not fit to the beginning assumption of K_G,K_A, and K_B.

Illustrated with Figure 3.2, the first two rule does not fit, its conditions are K_G but given is $\neg K_G$ and K_A but given is $\neg K_A$. The third

$$\begin{array}{lcl} K_G & \rightarrow & nil \\ K_A & \rightarrow & \alpha_1 \\ K_B & \rightarrow & \alpha_2 \end{array}$$

Figure 3.1: Example of a Teleo-Reactive Program. K_G, K_A, and K_B are conditions and α_n are actions that can be executed if condition(s) is(are) fulfilled. For this example, the condition K_G defines that the agent took a picture. K_A defines that the agent has to be on a defined position and K_B is *true* as long as the agent is not at the defined position. The action α_1 will take a picture, α_2 will move the agent towards the defined position.

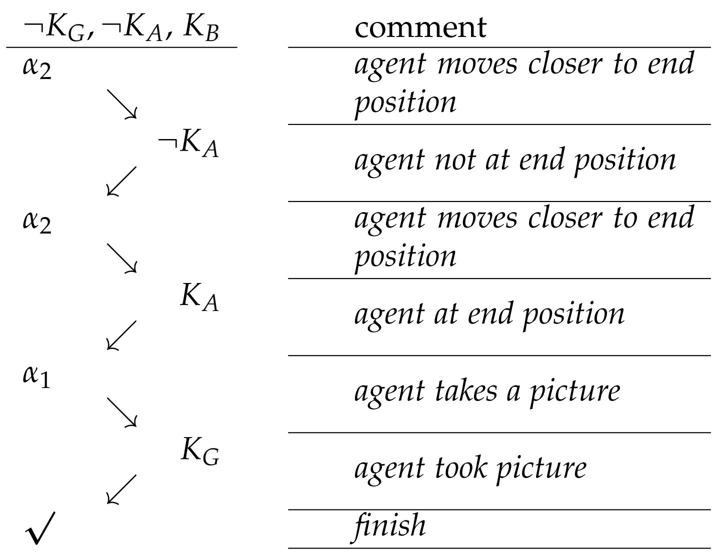


Figure 3.2: Example execution of a Teleo-Reactive Program. The first line shows the given situation.

rule fits because K_B is *true*, hence the action α_2 is executed, which effect move the agent into the direction of the defined position. But K_A is still *false* after the the first execution of α_2. Because K_B is still *true*, action α_2 is executed again, which effect now changes K_A to be *true* because it is sensed that the agent now on the defined position. For the third round the second rule fits and α_1 is executed. With the execution of α_1 the condition K_G changes to *true*. For the next round the first rule is taken because K_G is *true*. This will terminate this program because of *nil* which is the termination term of T-R programs.

The main advantage is that this action execution is robust and works well if retry helps. A T-R program are able to always achieve the goal, but only if two properties of the sequence are given. The first on is that the sequence has to be complete, which means that at least on condition of the sequence is true. Secondly the sequence has to respect the regression property. Therefore, it is necessary that each condition K_i is a regression of higher conditions reachable with the action α_i.

3.4 Monitoring and Diagnosis

Zaman, et al. [21] describe in their work a diagnosis and repair architecture for ROS-based robot systems by following the approach of model-based diagnosis and repair. This work deals with the detection and repair of faulty hardware and software components that negatively affects performance and the autonomy of a robot. They implemented a diagnosis and repair system that can be integrated into a running system. This work uses the robot operating system (ROS), which is also used by this thesis, see Section 4.1. The provided introspective functionalities of ROS are relayed to this work for monitoring. Their system includes several parts:

Diagnosis Board The hardware diagnosis board measures the voltage and current of each used hardware component and also switch them on and off. This makes hardware observation and even repair possible.

Observer Observers are implemented as ROS nodes and monitor different properties of the system. Different observer types are described, where each supports observation of a particular aspect of the robot.

Diagnosis Engine The diagnosis engine uses all the information of the observers and a diagnosis model to calculate a diagnosis, resulting with a set for faulty and a set for correctly working components. This engine uses the approach presented in [22], which is also described in Section 4.2.

Diagnosis Model Server The diagnosis model server provides the diagnosis model for the diagnosis engine. The description of the model is defined by a configuration at start time.

Repair Engine The repair engine takes current observations and diagnoses, transform the information into a planning problem and tries to solve the problem.

This description defines an observation and diagnosis system, which is transparent for the components of the robot, works in the background, and allows to add observations for further properties.

4 Prerequisites

In this chapter some basic prerequisites for this work are presented. That includes the used software framework, which is introduced as first. A detailed part about diagnosis is presented secondly.

4.1 Robot Operating System

Quigley, et al. [23] introduced the robot operating system (ROS). ROS[1] was initially released back in 2007 by Willow Garage[2] and the STAIR project at Stanford University[3] and is provided as open source project.

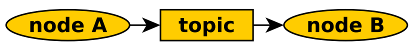
ROS is not a traditional operating system, it is a collection of frameworks for robotic software development. It is a robotics middleware and provides functionalities very similar to an operating system. It provides a structured communication layer above the host operating system. A big advantage is its support for heterogeneous cluster of computers. All units of the cluster are connected via a network interface in a peer-to-peer topology.

ROS is designed to be a language-independent framework. Therefore, it support C++, Python, and LISP. Some other language ports are additionally available but not all of them are completed till now.

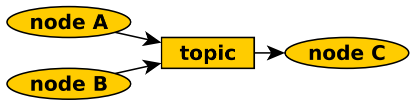
[1]`http://www.ros.org`

[2]`http://www.willowgarage.com/`

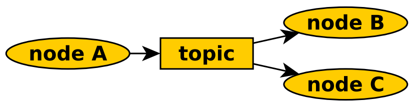
[3]`http://stair.stanford.edu/`



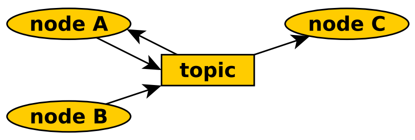
(a)Only one node publishes on a topic, which has only one subscriber.



(b)Several nodes publish on a topic, which has only one subscriber.



(c)Only one node publishes on a topic, which has several subscribers.



(d)Only one node publishes on a topic, which has several subscribers. One of the subscriber is also again the publishing node. Hence nodes can subscribe on their own published topics.

Figure 4.1: Here some possible connections are illustrated to show the topic-based publisher-subscriber model used by ROS. Illustrations are adopted from `http://wiki.ros.org/`.

The main concepts of ROS, which are nodes, messages, topics, and services, are described below.

nodes Each node has its own process to perform computation. Typically, a system comprises many of this nodes which are basically software modules.

messages Messages are typed data structures to offer the possibility of communication between nodes. This includes standard primitive types like integer or double, but also arrays and compositions of other messages are possible.

topics The communication between nodes is done by practicing publish–subscribe chaining. This technique offers a node to publish a message, without knowing the nodes that subscribes. As illustrated in Figure 4.1 publishing and subscribing nodes are not aware of each others.

services For synchronous transactions between nodes a pair of strictly defined messages for request and response are used. This communication is different to topics, it is limited to a many-to-one connection. Only one node can advertise a service and but several nodes are allowed to call the service.

actionlib Because ROS does not support asynchronous services Marder-Eppstein and Vijay Pradeep [24] developed the actionlib. With this extension it is possible to send requests to a node which will response, similar to services. The difference to services is that the requesting node continuous without waiting for the response. The requesting node is able to receive a periodic feedback from the progress of the request and it will also be able to cancel the execution of the request.

Because this framework is framework is open source, there exist a large community where many different software modules are available.

4.2 Consistency Based Diagnosis

In the work of Reiter [22] he described an approach for model-based diagnosis for a functional system description. The goal is to observe a real system, compare observations with a given model, and if there are some discrepancies, determine which system components had caused this behavior. It is also possible to determine different system components for the same faulty situation, hence diagnosis do not need to be unique.

System model Reiter defines a system as a pair (SD , COMPONENTS) where SD stands for the system description and COMPONENTS for the system components. The system description is a set of first-order sentences and contains information about the behavior of the component and the structure of the system. The system components are used as a finite set of constants. The behavior for each component is described by logical relations between input and output and additionally includes an “abnormal” predicate

$AB(x)$ for each component. The description for a correct working component c therefore includes the literal $\neg AB(c)$.

Observation The real system needs to be observed, to be able to compare it with the model. Therefore, the observation OBS is a finite set of first-order sentences.

Example To show the procedure the following example is used. Figure 4.2 shows a circuit with two logic gates. Therefore, the system uses the $COMPONENTS = \{AND1, AND2\}$.

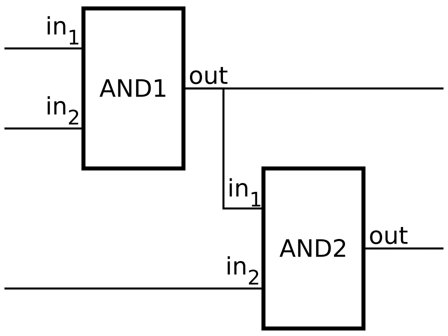


Figure 4.2: Circuit for consistency based diagnosis example.

This circuit has the following Boolean axioms:

$$SD = \begin{array}{l} AND(x) \wedge \neg AB(x) \rightarrow out(x) = and(in_1(x), in_2(x)) \\ AND(AND1),\ AND(AND2) \\ out(AND1) = in_1(AND2) \\ in_1(AND1) = 0 \ \vee \ in_1(AND1) = 1 \\ in_2(AND1) = 0 \ \vee \ in_2(AND1) = 1 \\ in_2(AND2) = 0 \ \vee \ in_2(AND2) = 1 \end{array}$$

The last three axioms are necessary to specify the inputs of the circuit to be binary.

$$OBS = \begin{array}{l} in_1(AND1) = 1 \\ in_2(AND1) = 1 \\ in_2(AND2) = 1 \\ out(AND1) = 1 \\ out(AND2) = 0 \end{array}$$

This is the observation set with example observation results:

With this observation the real system does not correspond with the model, because $SD \cup \{\neg AB(c) | c \in COMPONETS\} \cup OBS$ is inconsistent. The goal is to retract the set of abnormal components to get the formula consistent again.

The corresponding conflict set C for this faulty system is $C = \{AND1, AND2\}$. The first possible solution is that all components are faulty, which is no good solution. It is necessary to reduce the set of faulty component to a minimum.

This is done by assuming that one or more components of the conflict set are abnormal. If the system still stays faulty a new conflict set is created. This procedure is repeated till the system turns valid or no more combinations of abnormal components can be assumed.

All this conflict sets are used to find a hitting set, which should also be minimal. As a result, several diagnoses establish different minimal hitting sets.

For this example the minimal hitting set H will be $H = \{AND2\}$.

PyMBD

For the implementation part of this thesis, the software from PyMBD, developed from Quaritsch, et al. [25] was adopted. Modifications were necessary, because continuous diagnosis calculation and the support of continuous changing observations were required. Furthermore, a new way of adding new observation types was implemented. These types of observations are described in Section 5.2.2.

5 Concept

In the following sections of this chapter the concept of the dependable architecture for autonomous robots is presented. This concept defines the basic organization of the functionality and the basic structure of software components for monitoring, diagnosis, and fault handling.

The concept comprises three main parts. The first one is the architecture, which defines the needed basic software components on the one hand and the knowledge base, storing information for automated administration, on the other hand. Monitoring and diagnosis is the second part. It contains a description, who the system is supervised and how this information is used for diagnosis. Last but not least the fault handling is described.

5.1 Architecture

The architecture is divided into four parts. These are the high-level, the mid-layer, the low-level, and the knowledge base, as shown in Figure 5.1. The Figure also shows the possible interactions between the different layers.

5.1.1 Overview

As already mentioned the architecture is divided into four parts. The first one is the high-level. Its main task is to generate plans to reach given goals. Such goals are given by an user but can also be

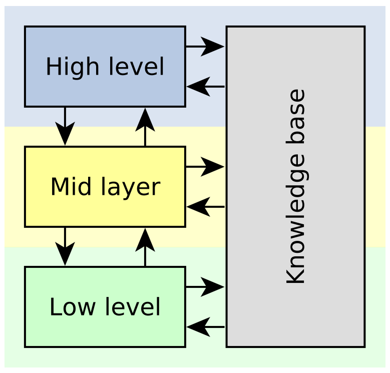


Figure 5.1: Architecture overview and the possible interactions.
high-level (blue), mid-layer (yellow), low-level (green), knowledge base(gray)

generated by the robot itself, to maintain its autonomy. Generated plans contains **tasks**, which are executed and supervised by the **plan executer**.

Tasks are located in the mid-layer. Each **task** requires defined **functionalities** before its **behavior** is executable. The combination of all **functionalities** of a **task** is called system configuration. The **task executer**, which is also part of the mid-layer, has to schedule and prepare the system configurations. Also the execution of the **behaviors** of the **tasks** are supervised by the **task executer**.

Behaviors and **functionalities**, which are used for system configurations, are components of the low-level. They may contain real-time control loops, object recognition, manipulation and other interactions with hardware components.

The fourth part of the architecture is the knowledge base. It stores all needed information about the different layers. Among others this includes goals, plans, configurations for components used for there initialization, configurations for observers, and parameters used by the diagnosis engine. Also return values of **behaviors** and **tasks** useful for the **task executer** and the **plan executer** are stored in the knowledge base. **Behavior**, **functionality**, and **task**

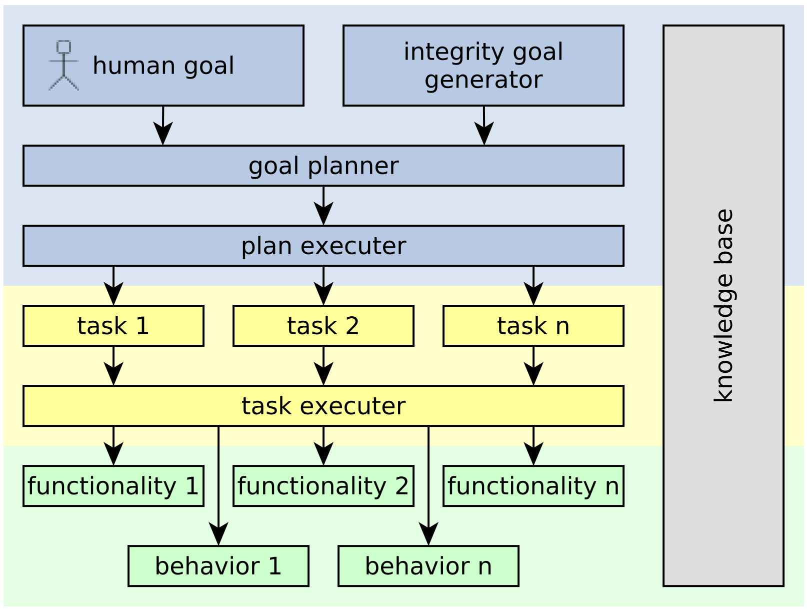


Figure 5.2: Detailed architecture overview and the possible interactions. high-level (blue), mid-layer (yellow), low-level (green), knowledge base(gray)

related values, used during their execution, can also be stored in the knowledge base. For low-level components the knowledge base may be not fast enough, for example to store video streams, but state information can be stored in the knowledge base in order to perform the execution. Last but not least, configurations of the observers are also stored in the knowledge base.

5.1.2 High-Level

The high-level consists of three parts, as shown in Figure 5.2 in blue. The first one is the **goal planner**. It generates plans to reach the given goals. As second, the **integrity goal generator** which is used to generate goals which insures the integrity of the robot. The

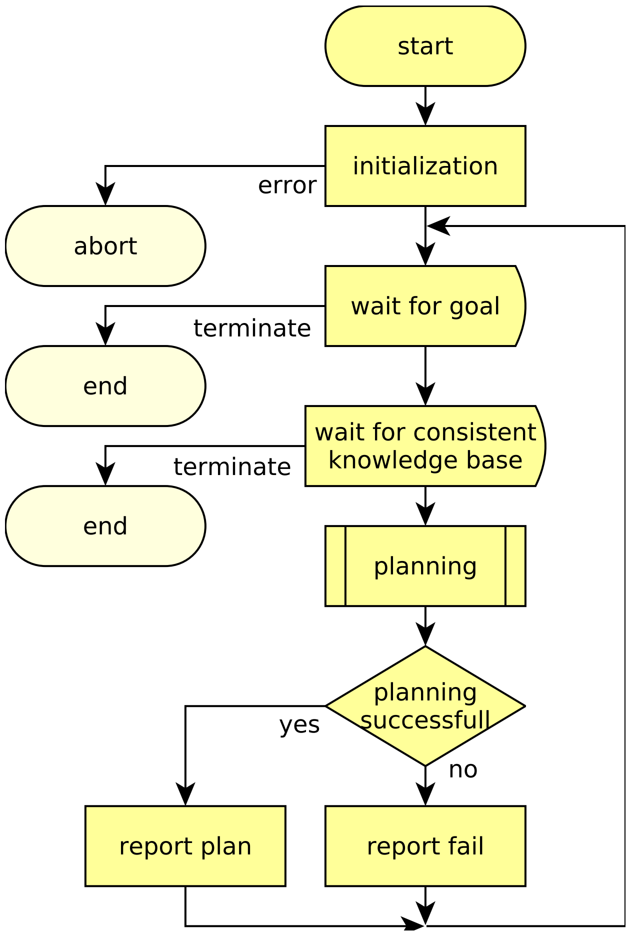


Figure 5.3: Flowchart of the **goal planner**.

task of the last one is to execute the generated plans and therefore it is called the **plan executer**.

Goal planner

The main task of the **goal planner** is to generate plans for given goals. Figure 5.3 shows a flowchart of the **goal planner**.

After starting and if initialization is done, it waits for new goals which are stored in the knowledge base. New goals can be stored in the knowledge base by an user or by the **integrity goal generator**.

Generating plan If new goals are available, the **goal planner** needs a consistent knowledge base. Therefore, it sends a request to the **plan executer** to pause execution. The **plan executer** is used to execute and supervise the execution of generated plans. It will be described in more detail later in this section. After the **plan executer** confirms the request, the planner generates a plan to reach the goals. As long as goals are not reached, they stay in the knowledge base. Reached goals are removed by the planning process of the **goal planner**.

In the survey of Weld [26], he shows different techniques for planning with such conditions by planning with graphs. Among others, he includes the GRAPHPLAN from Betz and Furst [27]. GRAPHPLAN is an algorithm for generating a shortest partial-order plan in STRIPS-like domains. Different conditions (mutex relations) between propositions, actions, and effects are used to represent their dependencies.

The planner need to consider about similar restrictions:

- Preconditions of **tasks** must be fulfilled, otherwise it can not be executed.
- **Tasks** can be executed parallel if the effects of the **tasks** do not interfere with each other. Weld describes this condition as *Inconsistent Effects*.
- A **task** can be executed parallel to other **tasks** only if its effect does not influence their preconditions. Weld describes this condition as *Interference*.

Another condition of Weld, which is called *Competing Needs*, describes preconditions that are mutually exclusive usable by actions. The **goal planner** does not include this condition, because the **task executer** cares about system configurations.

The planner needs to guarantee that the parallel execution of **tasks** of a plan is possible.

The generated partial-order plan only fulfills the before described conditions. The **task executer** (mid-layer), which executes and supervises all **functionalities** and **behaviors**, will make further restrictions later.

The plan is stored in the knowledge base and from where the **plan executer** fetches it. Otherwise if no plan can be generated, the planning fail will be reported.

Finally the **goal planner** goes back to the beginning and waits for new goals. It will loop till its process is terminated.

Integrity goal generator

The **integrity goal generator** is also part of the high-level. It is not a planner, but it generates and posts goals that are necessary to keep the robot autonomous as long as possible.

Figure 5.4 shows a flowchart of the **integrity goal generator**.

After starting and if initialization is done, it waits for interrupts.

This can be solutions from the diagnosis engine and repair system, which searches for solutions if components are not working correctly. If needed a goal is generated and stored in the knowledge base.

Finally the **integrity goal generator** goes back to the beginning and waits for a new interrupt. It will loop till its process is terminated.

Plan executer

The **plan executer** is the last of the tree parts of the high-level. It is used to execute the generated plan from the **goal planner**.

Figure 5.5 shows a flowchart of the **plan executer**.

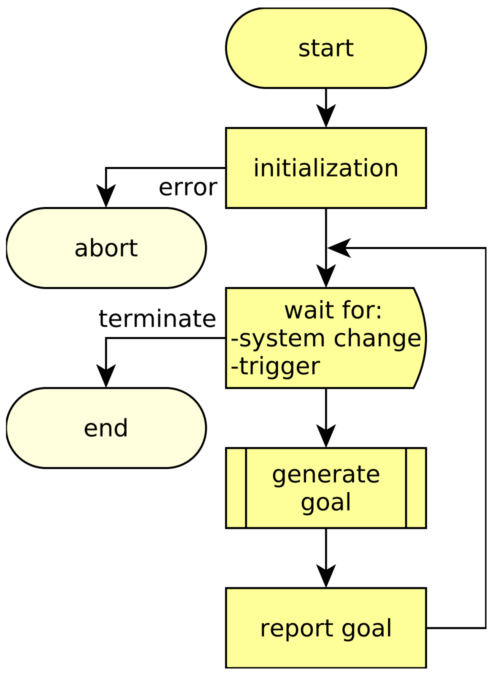


Figure 5.4: Flowchart of the **integrity goal generator**.

After starting and if initialization is done, it waits for a new plan. If one is available in the knowledge base, is will be fetched and executed. The plan contains **tasks**, which should be executed, and constraints, which controls the execution of the **tasks**.

Before a **task** is executed the **plan executer** updates the diagnosis model and prepare the observers. Updating the diagnosis model is necessary to keep the model up to date. The diagnosis model is the internal representation of all running **tasks** and low-level components. Otherwise observations can not be interpreted by the diagnosis engine. It is also necessary to update the diagnosis model if a **task** is stopped. For more details see Section 5.2.3.

After the diagnosis model update, also the observation is configured. Observer configurations for all **tasks** are stored in the knowledge base. For each executed **task** the observer manager is updated with the configurations of the current **task**. For more

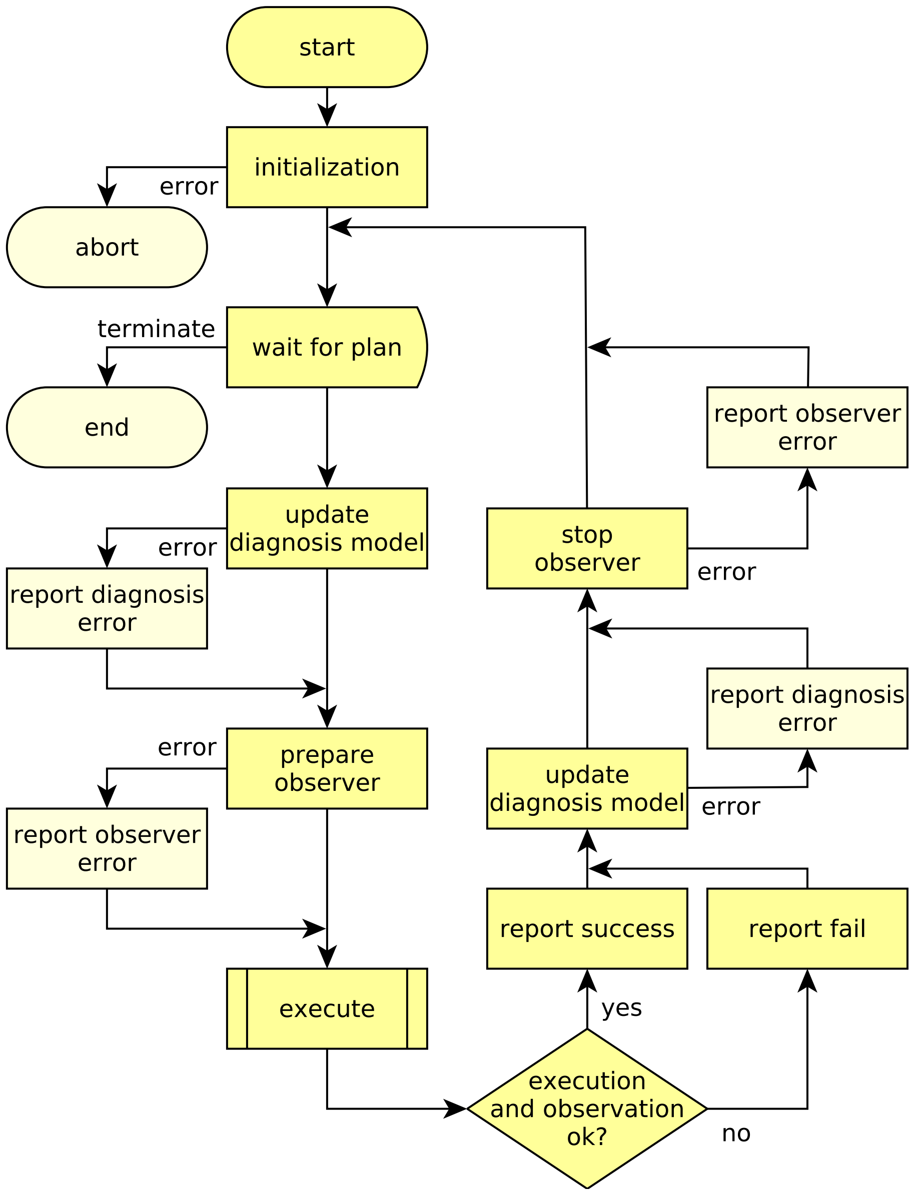


Figure 5.5: Flowchart of the **plan executer**.

details about observers and how this update procedure work see Section 5.2.2. Please note that currently there is no observer for **tasks** defined, but observers for low-level components.

In the execution step all **tasks** are executed at their defined start times. Several sequence diagrams are shown with Figure 5.6 and illustrates different possible scenarios for **tasks** and their required system configurations, **behaviors** and start times.

The work-flow of the **plan executer**, **tasks**, **task executer**, **functionalities**, and **behaviors** is always the same:

- the **plan executer** receives a plan and executes the **task** depending on their constraints,
- each **task** requests a system configuration which are send to the **task executer**,
- the **task executer** prepares the system and sends a response,
- after that the **task** requests the execution of **behavior** by the **task executer**,
- the **task executer** executes the **behavior**,
- after finishing, the **behavior** returns and may return a result,
- the result is forwarded to the **task** and to the **plan executer**

The first sequence diagram (Figure 5.6a) shows a simple scenario with only one **task**. The **task executer** receives a system configuration and starts the corresponding **functionalities** $f1$ and $f2$. After the system is prepared the **task** gets the response.

Figure 5.6b shows a sequence diagram with two **tasks** and both need the same system configuration. $t1$ and $t2$ do not have any execution time constraints, therefore both **tasks** are started simultaneously by the **plan executer**. This happens because the **plan executer** does not care about limitations with **functionalities**. The **task executer** schedules the execution, starting with $t1$, which gets the response first. After execution of **behavior** $b1$ of $t2$ finishes, $t2$ gets the response for executing $b2$.

The last sequence diagram (Figure 5.6c) shows a simple scenario with two **tasks** and both need the same system configuration. $t1$ and $t2$ have the execution time constraints that both have to start at the same time, therefore both **tasks** need to be started simultaneously. The **task executer** tries to schedule the execution, but there is no possible solution, because both **tasks** request the same system configuration for the same time. The request of the **tasks** is canceled, which effects replanning because the whole plan has failed.

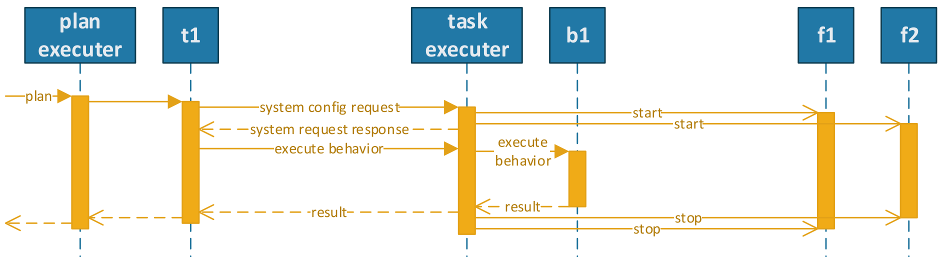
The sequence diagrams showed that it is necessary that **tasks** are least minimal constrained to give the **task executer** the possibility to shift the execution times of **tasks**. This is necessary, because **tasks** often require a defined system configuration, where conflicts in the configuration need to be avoided. For more details, see Paragraph **task executer** described in Section 5.1.3.

If the **task executer** of the mid-layer is not able to execute the given **tasks**, some of them reports an error, which is returned to the **plan executer**, illustrated in Figure 5.6c. The failing execution of a **task** stops the execution of the whole plan. The fail is reported, which requires replanning.

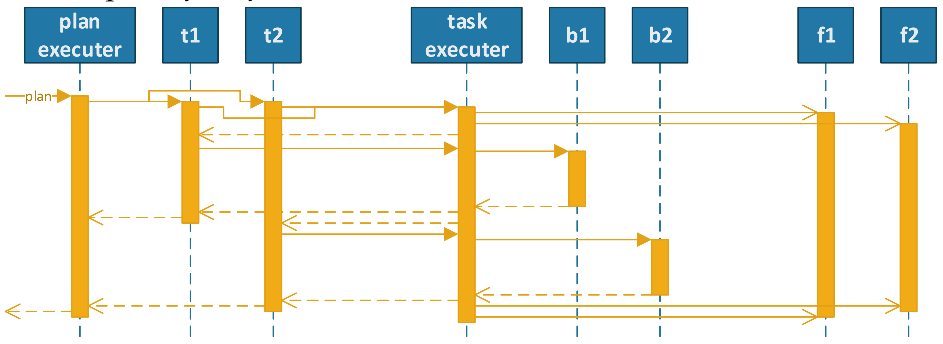
After the execution the diagnosis model is updated again and the observers for this **task** are stopped.

Interruption of running tasks As mentioned above, the **goal planner** requests to pause execution of **tasks** if it needs to generate a new plan. It is necessary to get a most likely consistent knowledge base while planning. Especially perceptional parts of **tasks** may negatively influence the planning procedure because the robot state may change during planning. Hence **tasks** need to be interruptible. If all **tasks** are paused, the knowledge base would not be updated any more by the **tasks**. The **plan executer** will send the response of the interruption request, so that all executions are stopped.

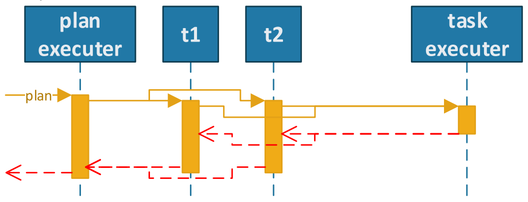
There is an exception for **tasks** that are not allowed to be interrupted and have to finish their job. These are **tasks** whose **behaviors** do not have interruption points because the whole execution would fail otherwise. Interruption points are specific moments in progress of the **behavior** that are possible for interruptions. To



(a)This sequence diagram shows a simple scenario with a plan that has only one **task** $t1$. $f1$, $f2$ are **functionalities** and $b1$ is a **behavior**. **Task** $t1$ requires $f1 \wedge f2 \wedge b1$.



(b)This sequence diagram shows a scenario with a plan that has two **tasks** $t1$ and $t2$. $f1$, $f2$ are non-shareable **functionalities** and $b1$, $b2$ are **behaviors**. **Task** $t1$ requires $f1 \wedge f2 \wedge b1$ and **task** $t2$ requires $f1 \wedge f2 \wedge b2$.



(c)This sequence diagram shows a scenario with a plan that has two **tasks** $t1$ and $t2$ which should be executed simultaneously. $f1$, $f2$ are non-shareable **functionalities** and $b1$, $b2$ are **behaviors**. **Task** $t1$ requires $f1 \wedge f2 \wedge b1$ and **task** $t2$ requires $f1 \wedge f2 \wedge b2$.

Figure 5.6: This sequence diagrams show scenarios with a plan that has one or two **task**(**s**).

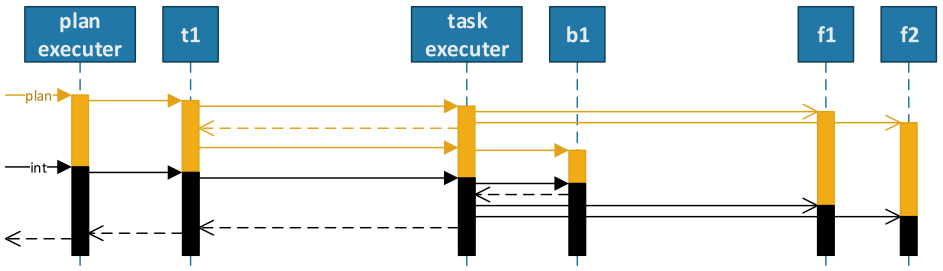


Figure 5.7: This sequence diagram shows the same example as shown in Figure 5.6. But this time an interrupt is send and *t*1 in an interruptible **task**. Therefore, the whole execution chain is informed to pause.

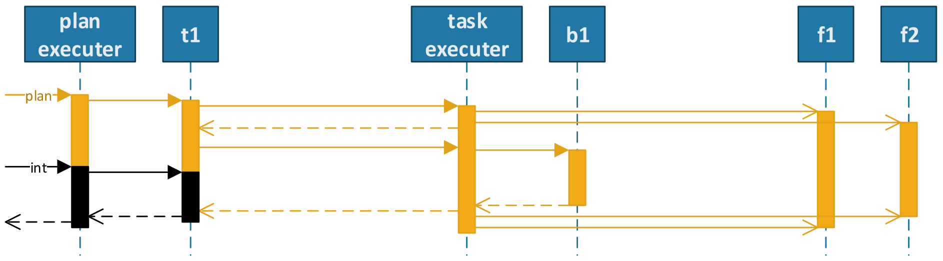


Figure 5.8: This sequence diagram shows the same example as shown in Figure 5.6. But this time an interrupt is send and *t*1 in a not interruptible **task**. Hence, the interrupt is noticed by the **task**, but the execution of the **behavior** is continued.

prevent such situations, **tasks** are allowed to ignore the interruption request as long as it is a critical **task** with a short execution time. This is necessary because the **goal planner** waits for the response of the interruption. For an illustration of interruptible and not interruptible **tasks**, see the sequence diagrams of Figures 5.7 and 5.8

After replanning, the **plan executer** gets the new plan. Some **tasks** of the new plan may be already started, but currently paused. This **tasks** are restarted or continued from their last state.

5.1.3 Mid-Layer

The mid-layer is shown in Figure 5.2 in yellow. Its main job is to execute all **tasks** given by the **plan executer** of the high-level. The start times of **tasks** are roughly defined, because the **task executer** need to be able to shift the execution times to make their execution possible. Therefore, the **task executer** has to prepare the system for the execution of each **task**. Some of the **tasks** are executed in parallel, but only if their required system configurations are not conflicting.

Task

Tasks are defined as the flowchart shows in Figure 5.9. After starting a **task** it needs to be configured. If this is done, a request for a defined system configuration is send to the **task executer**. This request will be blocking the **task** as long as the **task executer** does not response. The **task executer** gets also the constraints of the **task**, which are used to find a possible time slot where the system configuration is possible and reserved for this **task**. If there is no possible time slot the request of the **task** is canceled, which also cancels the **task**. An error is reported and the execution of the plan fails – replanning is necessary.

If there is a time slot found for this **task**, the **task executer** prepares the system. After that the **task** is informed by the **task executer** that the system is ready. This allows the **task** to start the execution procedure. But before the observers for this **task**, which are already configured, are activated. For more details see Section 5.2.2. Observers for this **task** are also deactivated if the execution of the **behavior** stops. Therefore, if the execution of the **task** is not running, the observers will not report the failing of a **behavior**.

The **behavior** that should be executed is send to the **task executer**, which supervises its execution. The execution of **tasks** can be stopped by various reasons. If the execution needs to be paused for the **goal planner**, it will be interrupted, if it is interruptible,

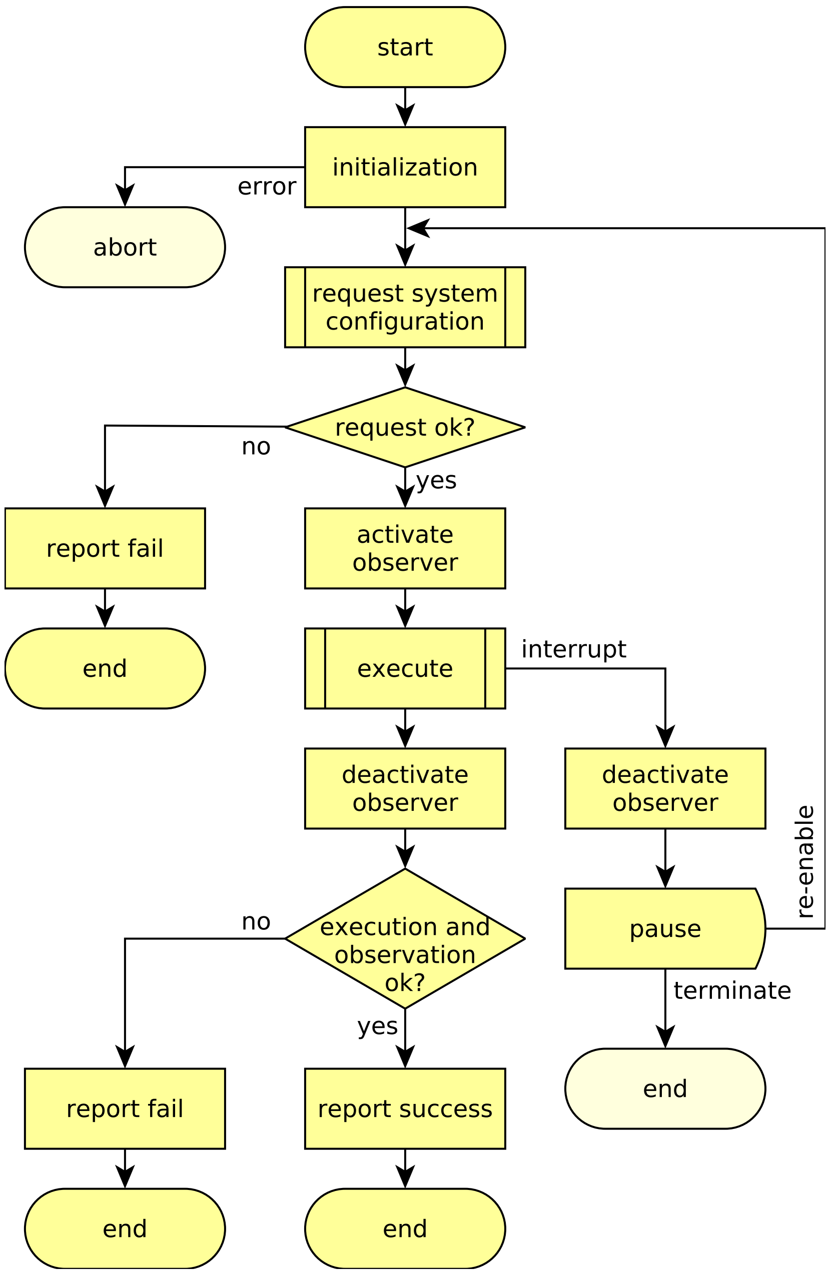


Figure 5.9: Flowchart of a **task**.

and it waits till execution is re-enabled again. For pausing the execution, it is necessary to inform the **task executer** to pause the corresponding **behavior**. If the **task** is not interruptible, the request is ignored – the **task** will continue till its finished. After re-enabling the execution of the **task** and if it is still part of the new plan, it is necessary to send another request for the system configuration to the **task executer**. This is necessary because other **functionalities** are also interrupted and need to be re-enabled. If the task is not part of the new plan it is terminated by the **plan executer**.

If the execution of a **task** fails or properties of the **task** are observed as faulty, it is necessary to report the failing execution of the **task**. This requests a replanning of the **goal planner**.

If the execution ends successful, the success is reported.

Task Executer

The **task executer** is a central part for the execution of **tasks**. This includes the following subjects:

- Order **tasks** in time slots, so that their required system configurations are possible without conflicts and to fulfill constraints of the **tasks** and the plan.
- Prepare the system with the necessary configuration for the execution of **tasks** at its time slots by executing and supervising execution of **functionalities**.
- Execute and supervise execution of **behaviors** of running **tasks**.

Managing System Configurations Each **task** requires a defined system configuration, where **tasks** are allowed to be executed parallel. Therefore, it is necessary to manage different system configurations applied at the same time.

At the beginning each started **task** (started by the **plan executer**) requests a system configuration, that is necessary for execution.

Additionally, the constraints of the **tasks** are also included in the request. **Tasks** can be started simultaneously or while other **tasks** are already running. Hence the **task executer** has to care about all **tasks** that are currently requesting a system configuration or are already running. This information gives the **task executer** the possibility to search for a time slot, where the system configuration can be applied without conflicts and all constraints are fulfilled. This is done by describing it as a Constraint Satisfaction Problem (CSP) [28], which tries to find a suitable timetable.

This timetable is used to prepare the system configurations at the given time slots, and inform the **tasks** that are still waiting for a response. The response of the **task executer** gives a **task** the permission to continue, because its requested system configuration is prepared.

With each new request over time, the timetable need to be rescheduled. If no solution can be found, the **task executer** is not able to prepare the system configurations for the **tasks**. **Tasks** that are still waiting for their system configuration are informed that their request has been canceled. This will cause a replanning, done by the **goal planner** of the high-level.

It is possible that during the scheduling procedure an interrupt for replanning arrives. This will cancel the scheduling procedure and all **tasks**, that are waiting for a system configuration are informed about the cancel of their request. But the **tasks** should already know about the interrupt, because they should have been informed by the **plan executer** by now.

Prepare System Configurations The second part of the **task executer** is to prepare the system configuration for the **tasks**. Each system configuration includes different **functionalities**, described in the next Section 5.1.4. Before the **task executer** can answer the system configuration request from a **task**, it needs to start all necessary **functionalities**, as shown in previous sequence diagrams (Figure 5.6).

The **task executer** has to supervise the execution of the **functionalities**. Because the functioning of **functionalities** and **behaviors** are the same the **task executer** does not distinguish between them.. Therefore, each execution of **functionality** is supervised from the same part of the **task executer** that also supervises the execution of **behaviors**, which is described next.

Supervise Behavior Execution The last part of the **task executer** is to manage the execution of all **functionalities** and **behaviors**, which are both low-level components. Additionally, it has to care about the right start procedure of each component, because each of them can have its own configuration, which is stored in the knowledge base. Hence the **task executer** has to load the configuration and prepare the started, but not running, low-level component. After finishing the initialization, the component is set to running.

If a **task** receives an interrupt to pause, the request may be forwarded to the **task executer** to pause the currently running **behavior**. Also each used **functionality**, which are used by this **task** may be interrupted. There is one case, where a **functionality** is allowed to continue even of the interrupt. The **task executer** does not forward the interrupt to the **functionality** as long as not interruptible **tasks** are using it. If all not interruptible **tasks** are finished, the **functionality** immediately gets the interrupt request.

To handle an interrupt, each **functionality** and each **behavior** that are used by interruptible **tasks** need a defined procedure to deal with this request.

For each execution of the low-level components the **task executer** also has to care about preparing the observers and update the diagnosis model. If observers are defined for components, they need to be started and configurated. Therefore, the **task executer** informs the observer manager about the required observers and observer configurations. The configuration of observers for each **functionality** and each **behavior** is stored in the knowledge base. For more details about this procedure see Section 5.2.2. Updating

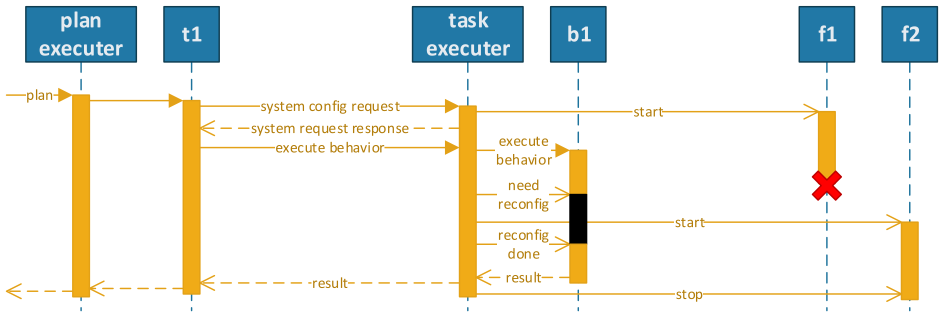


Figure 5.10: Sequence diagram of plan execution with faulty **functionality**. This sequence diagram shows a simple scenario with a plan that has only one **task** $t1$. $f1$, $f2$ are **functionalities** and $b1$ is a **behavior**. The **task** $t1$ requires $(f1 \vee f2) \wedge b1$. The fail of the **functionality** is marked with a red cross.

the diagnosis model is necessary to keep the model up to date. As already mentioned before, diagnosis model is the internal representation of all running components. The diagnosis engine is only able to interpret the observations, if observed components are present in the diagnosis model. For more details see Section 5.2.3 which describes the diagnosis and how its model is continuously updated.

Failing functionalities or behavior

It is not possible to ensure that the execution of low-level functions always finishes successfully. The handling with faulty low-level components is different for **functionalities** and **behavior**.

The sequence diagram of Figure 5.10 shows a scenario where a **task** requires a system configuration, that can be prepared by two different **functionalities**. Therefore, the **task executer** can decide between $f1$ and $f2$. If one **functionality** fails, the **task executer** can use the other **functionality** as an alternative. The example shows the fail of a **functionality**, which can be caused by different reasons. One reason may be, that the **functionality** itself raises an exception and stops execution. Another reason may be that

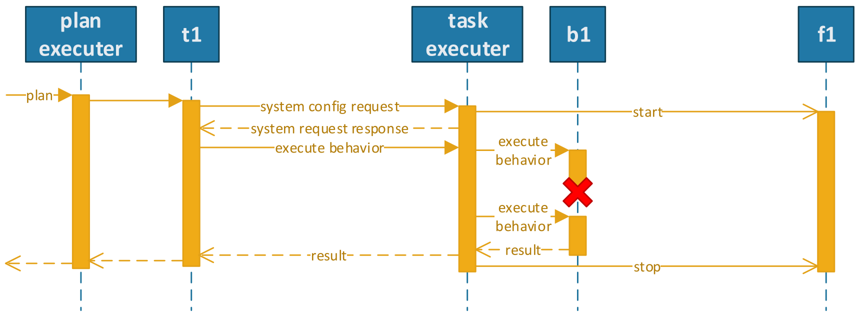


Figure 5.11: Sequence diagram of plan execution with faulty **behavior**. This sequence diagram shows a simple scenario with a plan that has only one **task** $t1$. $f1$ is a **functionality** and $b1$ is a **behavior**. The **task** requires $f1 \wedge b1$. The fail of the **behavior** is marked with a red cross.

the observers detects faulty situations of the **functionality**, the diagnosis reasons that the **functionality** is faulty, and the repair engine arranges the execution stop handled by the **task executer**. However, the execution stops, this **functionality** is required for **behaviors**, which may already running. The **task executer** tries to inform the **behaviors** to wait, because the system need to be reconfigured. If the **behavior** supports this request, they have to wait till the new system configuration is prepared. Otherwise, if the request is not supported, the execution of the **behavior** need to be canceled, which stops the execution of the **task** with an error – replanning is required.

The sequence diagram of Figure 5.11 shows a scenario where a **task** requires the **functionality** $f1$ to execute the **behavior** $b1$. The example shows the fail of a **behavior**, which can be caused by different reasons, similar to the previous example where the **functionalities** fails. One the one hand it is possible that the **behavior** itself raises an exception or the **behavior** is estimated as faulty by the diagnosis. However the execution stops, the **task executer** has to try to restart the **behavior** again. Some **behaviors** may require a "warm start" where already maintained achievements are loaded during restart. Therefore, the **behavior** need to store achievements in the knowledge base. If such information are found in the knowledge base, the **task executer** use them for the initialization of the **behavior**.

If the **behavior** cannot be restarted or it continuously fails after a certain number of restarts, the **task executer** stops the execution by returning an error to the **task**, which effects a failing plan execution and replanning.

5.1.4 Low-Level

The low-level is the lowest level of the architecture, see Figure 5.2. It is used for the communication with the hardware or real-time control loops. Additionally, components of the low-level can have a reactive manner, like stopping the robot immediately if an error occurs to keep the robot and other individuals of the environment save.

Both **functionalities** and **behaviors** have a very similar flowchart. Therefore, both are described by the same flowchart shown in Figure 5.12.

A low-level component is started and configured by the **task executer** of the mid-layer. Before the execution can start, all observers of this component are activated. The execution is started also by the **task executer**. After the execution stops, all observers are also deactivated.

If an interrupt arrives, forwarded from the **task executer**, the component need to pause and wait for being re-enabled again or being terminated by the **task executer**.

The execution of a component is stopped if an exception is raised by the component itself or an observers detects faulty situations, the diagnosis reasons that the component is faulty, and the repair engine arranges the execution stop.

If the execution is stopped the component reports its stop.

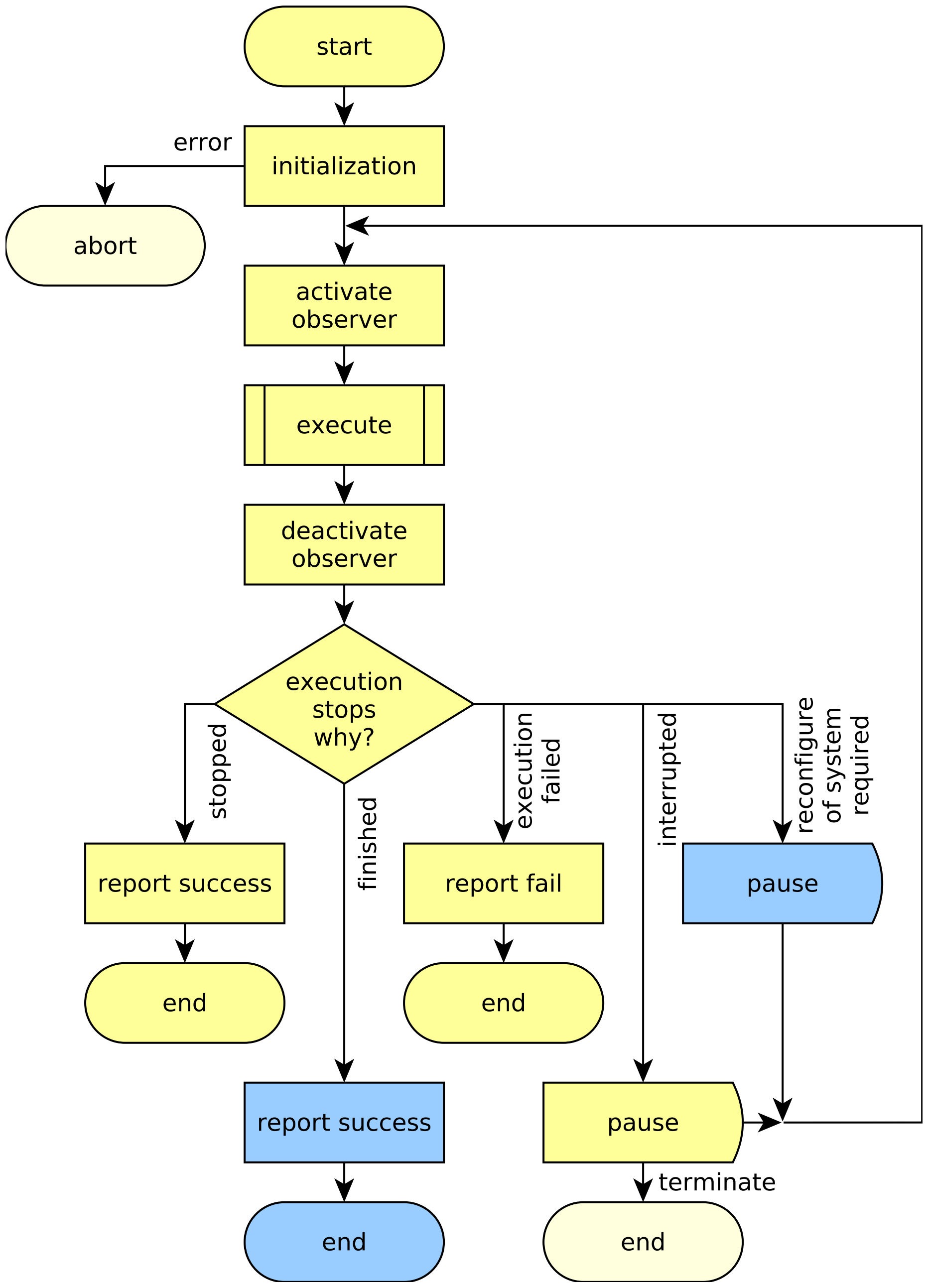


Figure 5.12: Flowchart of a low-level component which is either a **functionality** or a **behavior**. Blue parts are only necessary for a **behavior**.

Another two cases are only used for **behaviors**, colored blue in the flow chart. The first one is very similar to the interrupt. If the component is informed by the **task executer** about a needed reconfiguration of the system, the component has to pause. It may be the case that the **behaviors** does not support a reconfiguration of the system, because of possible lags of the data-flow. In this situation the **task executer** has to stop the execution of the **behavior** and reports an error to the **task** that the execution has failed.

The second case is that the execution finishes without errors. The success is reported and the **behavior** ends.

The major difference of **functionalities** and **behaviors** is that **functionalities** are used by the **task executer** to prepare a system configuration that is necessary for the execution of a **behavior**. Therefore, **functionalities** are started before and stops after the **behavior**. A **functionality** is designed for running continuously after it has been started, hence it does not have the finish state as shown in the flow chart. Some of the **functionality** are shareable and some of them are not. Shareable **functionalities** are primary used for perception, whereas non-shareable **functionalities** are primary used for control the robot and interact with the environment.

5.1.5 Knowledge base

The last of the four parts to describe is the knowledge base. It is a central component and accessible from all three layers. Besides the high-level also the mid-layer and even the low-level have access to it. The knowledge base is used for providing, collecting, and storing all kind of information. As Beetz, et al. already mentioned in [12], it is a challenging **task** to keep the information grounded. Harnad [29] describes in his article the problems of symbol grounding. To give a short example, the **goal planner** only need the name of an object of the environment to create a plan, but the path planner, which plans a path to the object, requires a detailed metrical position.

name	provided by	required by
goals	user, integrity goal generator	**goal planner**
plans	**goal planner**	**plan executer**

Table 5.1: Knowledge base information regarding the high-level.

name	provided by	required by
configuration		**plan executer**
observer configuration		observer manager, **plan executer**
task dependent values (return values)	**task**	high-level, **task**, observer

Table 5.2: Knowledge base information regarding the mid-layer.

The following Tables show a short overview which data the knowledge base has to processes. The first Table 5.1 lists some of the necessary data regarding the high-level. This includes the goals from the user and the integrity goal generator on the one side and the generated plan from the **goal planner** on the other side. Here the knowledge base is used for data exchange between all high-level components.

Table 5.2 lists some of the necessary data regarding the mid-layer, and the last Table 5.3 lists some of the necessary data regarding the low-level. Both include necessary configurations needed for the initialization of **tasks**, **behaviors**, and **functionalities**. Also the configuration of the observers is stored in the knowledge base. This includes several configurations of each observer for each component (**tasks**, **behaviors**, and **functionalities**). All the configurations for observers and components itself are predefined.

These tables only list information regarding the different levels of the architecture. But each component has access to the knowledge base and can use it to store:

- temporal information needed by the component itself,
- state information of itself,

name	**provided by**	**required by**
configuration		**task executer**
observer configuration		observer manager, **task executer**
component dependent values (return values)	**functionality**, **behavior**	observer, **behavior**, **functionality**, **task**

Table 5.3: Knowledge base information regarding the low-level.

- information of achievements, which are loaded if the component is restarted ("warm start"),
- data exchange with other components (limited bandwidth)
- return values.

The exchange of data between component by using the knowledge base is primary for state information in order to perform the execution. For sharing data that requires a high-bandwidth, like video streams, other techniques need to be used.

5.2 Monitoring and Diagnosis

The following section describes the monitoring and the diagnosis concept of the universal architecture.

5.2.1 Overview

The diagnosis system has the following structure as illustrated in Figure 5.13. Each system component, like low-level components as well as **tasks** of the mid-layer, are monitored by different types of observers. Those observers are used to monitor certain aspects of the system. This information of all the observers is processed by a diagnosis engine.

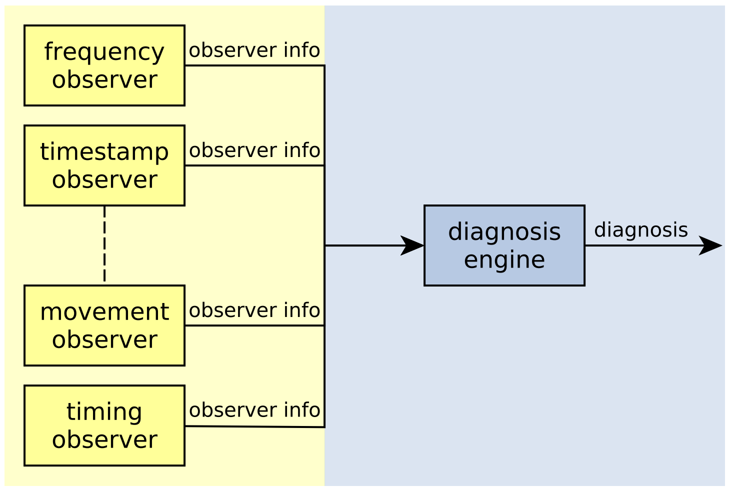


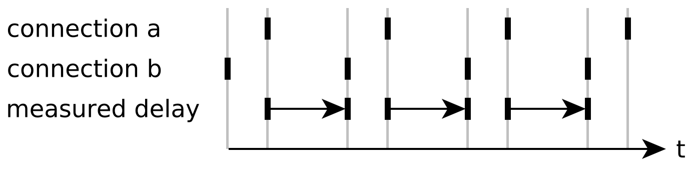
Figure 5.13: Monitoring and diagnosis overview. observer (yellow), diagnosis engine (blue)

5.2.2 Observers

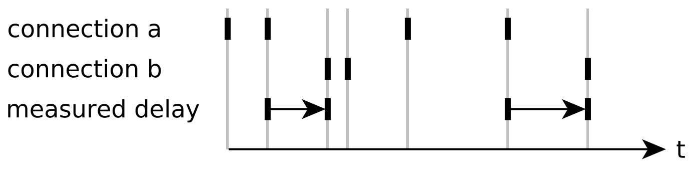
Observers are used to monitor different properties of the system components. Some of them are used to check components directly and some of them only can check the communication between the components. In the following two types are described, that are able to monitor components directly.

Resource This observer is used to check the consumption of system resources of a component. Memory usage and CPU load are monitored by using the system monitor of the operating system.

Component To ensure a component is started, this observer checks for the availability of a component. This does not assure a correct working component, just that it is up and known by the system. As an example, this can be realized by using the system monitor of the operating system for searching for processes of the component.



(a)Delay measurement between connections with same frequency.



(b)Delay measurement between connections with stochastic messages.

Figure 5.14: This figures show two different types of delay measurement. Each message is marked as black bar. The arrow shows the measured time.

The following presented observers are used for monitoring the communication of components. Note that all observers that monitors communication and properties of messages need to add themselves to the receivers lists of the connection. So they are able to receive all send messages.

Frequency The first property of a message, if its send periodically, is the frequency, which can be measured by the Frequency observer.

Time-out If it is necessary that messages are send within a defined period of time, this observer can be used. It will be able to detect lags of messages if they are to big.

Timing The Timing observer basically measures the time difference between two connections. This is done by waiting for the message of the first connection and calculate the time delay to the next message of the second connection.
Figure 5.14 illustrates the measured delay between messages. The upper Figure shows two connections which have the

same frequency. As shown, the measurement starts at the first received message from the first connection. When the message from the second connection is received, the time difference is calculated. With this observer its possible, for instance to monitor the input and the output of a component and if this component sends output messages within a defined time delay depending to input messages.

The lower Figure shows two connections where the messages are send stochastically. As it is shown, the delay is calculated between the closest pair of messages from both connections. With other words, all messages from the first connections except the last are ignored as long as no message from the second connection is received. All following messages of the second connections are ignored as long as no message from the first connection is received.

Time-Stamp This observer reads the content of a message, which message type has to contain a time-stamp. If the time-stamp of the sent message is older that the defined limit, this observer reports it.

Value This observer is able to monitor the content of messages, which requires that the observer can handle the data type of the messages. The observer can compare the contents of the messages with predefined contents or check if values of the messages are within a defined range.

Movement The movement observer is a very special one. Its task is to compare two different connections and both sends movement information. Both messages are compared and if the difference is bigger than a defined threshold the observer reports it. This observer can be used, for instance, to compare measured movements from the IMU[1] and the sensed movement from the odometry which should be the movement measured by the rotation speed of the wheels or tracks. The observer is able to detect if the robot has slippage because of a slippery surface. If so the acceleration and rotations sensors

[1]Inertial measurement unit: A device that is able to measure force, angular rate, and magnetic field for each axis.

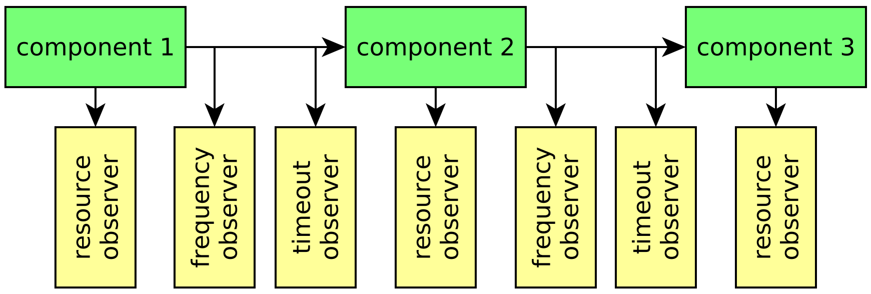


Figure 5.15: This is an example of observers that observe components. Components are observed by resource observers and the connections are observed by frequency and timeout observer.

of the IMU measures less movement than the odometry.

These are only some useful observers which are needed for monitoring an autonomous robot. If other special observers, like the last one, are needed, they can easily be added.

All the different observers need to be known by the diagnosis engine. Otherwise it will not know how to interpret the observer's reports. This rules are described is the Section 5.2.3.

Observer Management for Universal Architecture

All this observes need to be managed by a central module. The tasks of this module, which is the observer manager, are as following:

load and configure observers The observer manager has to load all observers which are needed. Before a component is started different observers need to be started and configured. The **plan executer** and the **task executer** have to care about that. Both get the observer configuration of the component from the knowledge base. The necessary observer configuration is send to the observer manager. This configuration contains information about all observers that are needed. As shown in Figure 5.15 there exist several instances of the same observer. This is necessary because each instance of an observer is configured for one single component or one single connection. Each component and each connection has its own properties, therefore each observer has its own thresholds, limits and parameters. All this information is defined in the configuration request of the **plan executer** or the **task executer**.

activate/deactivate observers Before the execution of a component starts all observers of the component need to be activated. As long as observers are deactivated, they will not monitor the properties of the component or connection. Activating and deactivating of observers is necessary to control if observers should monitor the properties or not. If observers are activated while the component is not executed, the observers may report faults even though the component is not running.

stop observer If the observer manager is informed that a low-level components or a **task** is stopped, also the observations are not necessary any more. The observer manager will stop and unload the corresponding observers.

5.2.3 Diagnosis

The diagnosis engine gets all the observations from all observers. If there is at least one observation that reports a faulty behavior the engine calculates different sets of possible faulty components following the approach of consistency based diagnosis, which is basically described in Section 4.2.

Definitions for Diagnosis Rules

The diagnosis model contains many components, that are currently executed. Most of the components have connections to other components. Hence faults can propagate through the network of components and connections. This behavior need to be formalized with clauses, which are used to define the model of the diagnosis engine. But before some definitions are necessary.

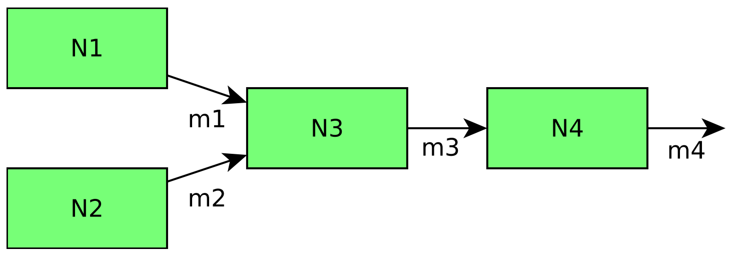


Figure 5.16: Dependencies of components in the diagnosis model. Components are named as $N1$, $N2$, $N3$, and $N4$. Connections of data flows are named $m1$, $m2$, $m3$, and $m4$.

Communication Graph On the one hand, the communication graph contains components $\mathcal{N}$. In the following description of the rules the term n is used to select one component out of $\mathcal{N}$.

On the other hand connections are also contained in the communication graph. These connections are defined with $\mathcal{M}$. In the following description of the rules the term m is used to select one connection out of $\mathcal{M}$.

Additionally, two functions are used. These two are $input : \mathcal{N} \rightarrow 2^{\mathcal{M}}$ and $output : \mathcal{N} \rightarrow 2^{\mathcal{M}}$.

General Rules for Diagnosis Model

Each component can use several data inputs and outputs. This communication between the components defines the dependencies between the components. Figure 5.16 shows an example with four components $N1$, $N2$, $N3$, and $N4$. These connections are named $m1$, $m2$, $m3$ and $m4$. Basically if an observation of a component detects a fault the component is faulty. As a result, the data output of the faulty component can also be faulty. Because of the dependencies of the components, following components that use this faulty data as input are also allowed to have faulty output, but the components themselves are not faulty. On the other side, if an observation of a connection detects a fault, all previously components are possible faulty components.

To define this dependencies two formulas are necessary. Formulas are defined as clauses. The atom $AB()$ stands for the "abnormal" predicate of components or messages as mentioned in 4.2.

The first one is

$$\forall m_o \in output(n) : AB(n) \rightarrow AB(m_o)$$

which implicates that if component n is faulty also the message m_o is faulty. The second formula, which is

$$\forall m_o \in output(n) : AB(m_o) \rightarrow \left(AB(n) \bigvee_{m_i \in input(n)} AB(m_i) \right)$$

defines that if a message $AB(m_o)$ is faulty either its source component itself or some input messages are faulty.

For example, the Clauses 5.1 show all needed clauses for representing the dependencies for the scenario of Figure 5.16.

$$\begin{aligned}
&AB(N_1) \rightarrow AB(m_1)\\
&AB(N_2) \rightarrow AB(m_2)\\
&AB(N_3) \rightarrow AB(m_3)\\
&AB(N_4) \rightarrow AB(m_4)\\
\\
&AB(m_1) \rightarrow AB(N_1)\\
&AB(m_2) \rightarrow AB(N_2)\\
&AB(m_3) \rightarrow (AB(N_3) \vee AB(m_1) \vee AB(m_2))\\
&AB(m_4) \rightarrow (AB(N_4) \vee AB(m_3))
\end{aligned} \tag{5.1}$$

Special Rules of Observers for Diagnosis Model

The basic rules are only describing the dependencies between the components. Each observer type also needs rules, to describe their behavior for the diagnosis model. Some of them are very similar, because they just measure different properties of a connection for instance.

Resource The Resource observer monitors a component n directly. Hence a detected fail can directly assigned to the component. Therefore, the rule looks like

$$\neg obs_{recource}(n) \rightarrow AB(n).$$

Component This observer also checks a component n directly. Hence the rules look similar to the Resource observer rule.

$$\neg obs_{component}(n) \rightarrow AB(n)$$

Frequency Observing the frequency of messages of a connection m requires this rule:

$$\neg obs_{frequency}(m) \rightarrow AB(m)$$

This time not the component is included, but the connections of the component. If the observation is faulty the connection need to be faulty.

Time-out Time-out observer is a watchdog for a connection m and needs a similar rule to the Frequency observer.

$$\neg obs_{timeout}(m) \rightarrow AB(m)$$

Time-Stamp The time-stamp does not check properties of the connection m but meta-information. The needed rule for the diagnosis looks like

$$\neg obs_{timestamp}(m) \rightarrow AB(m).$$

Value The model for the value observer uses

$$\neg obs_{value}(m) \rightarrow AB(m)$$

for describing its behavior. If values of the messages of connection m does not fit to the predefined values, the connection is faulty.

Timing The rule for the timing observer looks like

$$\neg obs_{timing}(m_1, m_2) \rightarrow (AB(m_1) \vee AB(m_2)).$$

If the observation detects an illegal delay between two messages of different connections $m1$ and $m2$, one of this connections need to be faulty.

Movement The last described observer is the movement observer, which compares two different connections whose messages contains movement information. It is described with

$$\neg obs_{movement}(m_1, m_2) \rightarrow (AB(m_1) \vee AB(m_2) \vee AB(movement))$$

which defines, that if the observation of the movement is abnormal, either one of the compared connections ($m1$ or $m2$) is faulty or the movement (*movement*) itself, which is a **behavior**, is faulty. This observer needs a third type, besides components and connections, to describe a faulty situation. For instance, if the robot stands on a slippery surface and it tries to move, the information of the wheel may not match with the information from the IMU. If this is the case the robot slides and the movement fails, but all components and connections are working correctly.

Diagnosis Engine for Universal Architecture

To use the diagnosis engine for the universal architecture it has to provide an interface for updating the diagnosis model. This is necessary because if low-level functions and mid-layer **tasks** are started or stopped, the diagnosis model need to be informed about the changes. Otherwise it will not be able to interpret the new observations of newly started modules or it maybe waits for observations of modules that were already stopped.

The following example is used to describe this procedure. The scenario is shown in Figure 5.16.

1. At the beginning no components are running. Therefore, the diagnosis model is empty.

components	**input**	**output**	**observers**

2. The first component $N1$ is started and its output $m1$ is monitored by a frequency observer $obs_{frequency}(m1)$. After the update the diagnosis model is generated based on the following information.

components	**input**	**output**	**observers**
$N1$		$m1$	$obs_{frequency}(m)$

3. A second component $N2$ is started. It also has no input connection and one output connection $m2$. The connection is also monitored by a frequency observer $obs_{frequency}(m2)$.

components	**input**	**output**	**observers**
$N1$		$m1$	$obs_{frequency}(m1)$
$N2$		$m2$	$obs_{frequency}(m2)$

4. The third component $N3$ is started. It uses the connections $m1$ and $m2$ as input and has the connection $m3$ as output. The connection is monitored by a frequency observer $obs_{frequency}(m3)$. Additionally the $N3$ is monitored by a resource observer $obs_{resource}(N3)$ to.

components	**input**	**output**	**observers**
$N1$		$m1$	$obs_{frequency}(m1)$
$N2$		$m2$	$obs_{frequency}(m2)$
$N3$	$m1$, $m2$	$m3$	$obs_{frequency}(m3)$
			$obs_{resource}(N3)$

5. The last component $N4$ has the connections $m3$ as input and $m4$ as output. This component is monitored by a component observer $obs_{component}(N3)$ to.

components	input	output	observers
$N1$		$m1$	$obs_{frequency}(m1)$
$N2$		$m2$	$obs_{frequency}(m2)$
$N3$	$m1$, $m2$	$m3$	$obs_{frequency}(m3)$
			$obs_{resource}(N3)$
$N4$	$m3$	$m4$	$obs_{component}(N4)$

The diagnosis engine uses the information, shown above, to generate the diagnosis model. The model need to be generated after each update (after each start or stop of components). Updates because of terminating components remove the corresponding entries of the diagnosis engine. Therefore, the afterwards new generated diagnosis model does not include the terminated components.

5.3 Concept for Fault Handling

The following section describes a possible way how to handle faulty software components of the universal architecture.

Rule engine

For repairing faulty component of the robot's software a simple rule system was chosen.

This system has a lot in common with a T-R program [4]. Each rule that is included in the sequence of the T-R program has a condition and an action. This rule engine differs from a T-R program by the execution. The rule engine does not stop searching if a valid rule is found, but it searches all valid rules and executes their actions.

The rule engine collects the information of the diagnosis engine and additionally all the information of the observers. See Figure 5.17, which shows the extended overview of observers, diagnosis engine

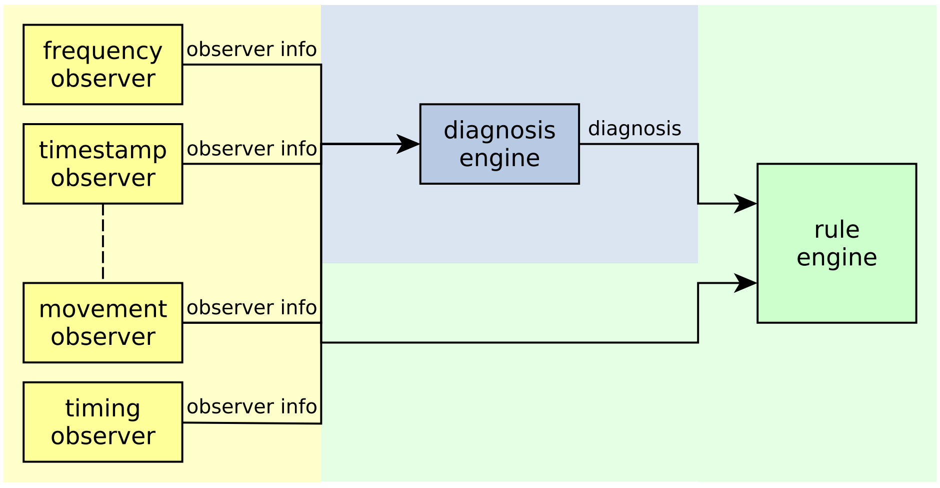


Figure 5.17: Monitoring, diagnosis, and fault handling overview. observer (yellow), diagnosis engine (blue), rule engine (green)

and the added rule engine. If the collected information fit to defined rules, these rules are triggered.

Each rule consisted of two parts, first is the condition and second is the action.

The condition is a logic test and, if it is satisfied, it triggers the corresponding action. Each condition is defined as a conjunctional set of the following sub-condition. Each sub-conditions composes all its elements by a disjunction.

positive observations This particular sub-condition defines a set of observations that has to be positive (not faulty).

negative observations This sub-condition is defines a set ob observations that has to be negative, which is faulty.

positive possible faulty diagnosis Here, a set of positive possible faulty components need to be defined. Possible faulty components are calculated by the diagnosis engine and for this sub-condition, they has to be assumed as faulty.

negative possible faulty diagnosis The last sub-condition defines negative possible fault components, which are components that are not faulty.

rules			actions
1. rule	positive possible faulty diagnosis	$N1$	1. action
	negative possible faulty diagnosis	$N2$	
2. rule	positive possible faulty diagnosis	$N1$, $N2$	2. action
3. rule	positive possible faulty diagnosis	$N1$	3. action
	positive possible faulty diagnosis	$N2$	

Table 5.4: Example rules used by the rule engine.

Table 5.4 shows examples how such rules can look like. The first rule defines that the component $N1$ need to be diagnosed as faulty and $N2$ as not faulty. This is the only situation where the rule engine triggers the action of this rule.

The second rule is triggered if the components $N1$ or $N2$ are diagnosed as faulty. Therefore, it also triggers if the first rule triggers.

The third rule defines that both component $N1$ and $N2$ need to be diagnosed as faulty. If this rule triggers also the second rule triggers but the first rule will not. This example shows also that the rule engine is able to trigger several rule at the same time.

As already mentioned after a condition test satisfies, the corresponding actions are carried out. For this, different types of actions are available. This are only some examples:

execute process This type of action executes a defined command of the operating system, which can kill the process of the faulty component for instance.

send mail Often it is necessary to inform human beings about a faulty situation. With this action, the rule engine of the robot sends a mail with a defined content.

write log Probably the most used action is to write content to a log file if the system seems to be faulty.

send command This action type is able to send a command to components of the robot, for example the high-level.

change parameter The last mentioned type changes parameters of components of the robot.

After the actions have been executed the rule engine waits for an update of the observations and the diagnosis to recheck if some rules can be triggered.

6 Implementation

In this Chapter more details about the implementation and the configurations are given. At first, details about the implementation of the architecture are presented. This includes low-level components, the **task executer**, and parts of the needed knowledge base. Section 6.2 presents the observers that have been implemented with the ROS framework[1] described in Section 4.1. The diagnosis engine is described afterwards. It was adopted from the code of PyMBD[25]. The rule engine is presented as fourth.

All configurations used for the diagnosis and the observers are stored in yaml-files and not in the knowledge base as described in the concept. YAML [30] is a data serialization language for many different programming languages and often used for configuration files.

6.1 Architecture

The implementation of the described architecture from Section 5.1 includes the **task executer**, **functionalities**, and **behaviors**. Additional parts of the knowledge base are included.

The **task executer** is structured as shown in Figure 6.1. The **task executer** gets its commands via several TCP connections and using Protocol Buffers[2], which is a mechanism for data serialization. For each command another connection is established. Therefore,

[1]`http://wiki.ros.org/`

[2]`https://developers.google.com/protocol-buffers/`

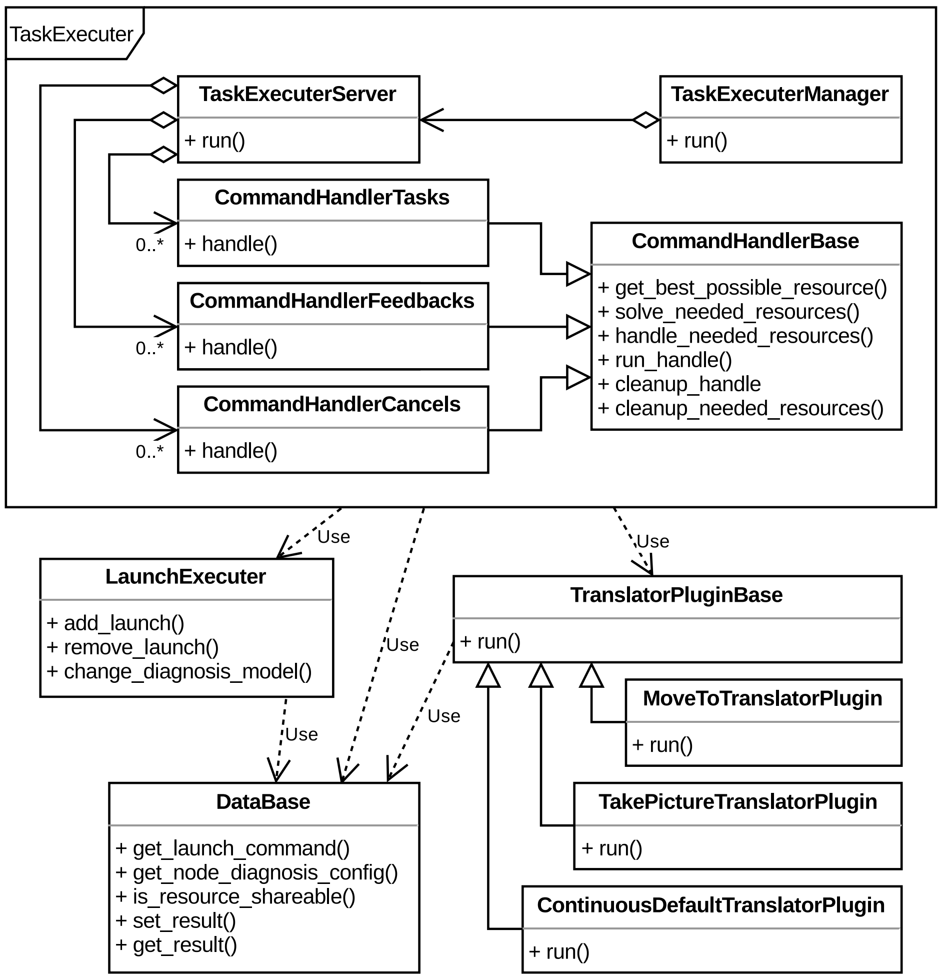


Figure 6.1: Class diagram of the **task executer**.

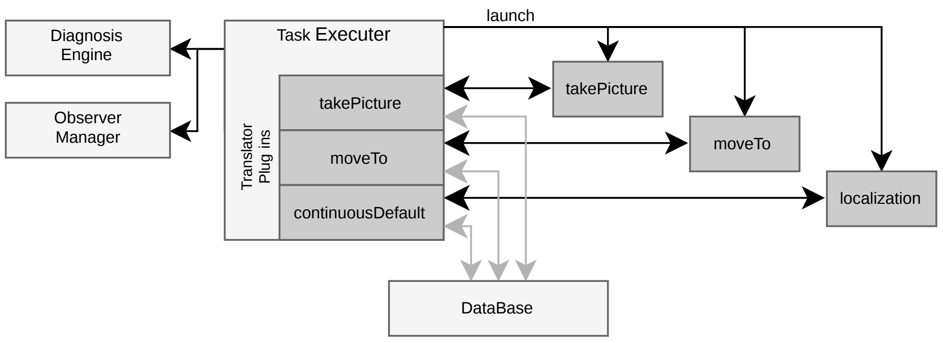


Figure 6.2: Work-flow of the **task executer**.

the **task executer** uses three different command handlers, one for starting a new task, one for send feedback of a running task, and one for cancel a running task. All three command handlers use the same base, which offers all needed methods.

The **task executer** uses the launch executer to start and stop **functionalities** and **behaviors**. The launch executer is designed to start roslaunch-files as new process. Therefore, it is possible to start and stop any ROS-package. The launch executer also cares about the update of the observers and the diagnosis model. All necessary information for updating are stored in the database and need to be predefined.

Each **functionality** and **behavior** can have its own interface, like for the **behavior** *takePicture* uses other instructions than the **behavior** *moveTo*. This came from the fact, that every ROS-node has its own purpose. Therefore, it is necessary to create a plug-in for translating the instructions coming from the **task executer** to instructions understandable by the nodes, see Figure 6.2. These plug-ins are derived from translator plug-in base, which defines required methods. Besides the start and stop of nodes also the translator plug-ins are started and stopped.

This design makes it possible to implement, start, and test **functionalities** and **behaviors** by its own, without the necessity of any running parts of the architecture.

6.2 Observation

This section describes the monitoring of nodes and its topics. At first the necessary message is defined, followed by the observer manager and provide functionalities, usable by the observer plugins. The resource monitor collects information of running processes from the operating system and prepares the information for observers that need them. At last all implemented observers and its configuration are described.

6.2.1 Messages

Each observer generates data about its monitored component. This information need to be send to the diagnosis engine. Hence the information need to be send as structured data as shows in Listing 6.1. The observation (last line of listing) gives information about the current state of the observed node or topic by using a positive integer. It can be used to inform about the current operation mode of the monitored component. Negative state information specifies faults. Therefore, the observers are able to give different types of faults of nodes or topics. Currently the diagnosis engine only distinguishes from correct (positive state) and faulty (negative state) working components.

Listing 6.1: Observer Info message

```
tug_observers_msgs/observation_info[] observation_infos
  std_msgs/Header header                # ros header message
    uint32 seq
    time stamp
    string frame_id
  string type                           # observation type, e.g. hz
  string resource                       # observed topic and/or node name
  tug_observers_msgs/observation[] observation
    int32 GENERAL_OK=0
    int32 GENERAL_ERROR=-1
    int32 NO_STATE_FITS=-2
    int32 NOT_AVAILABLE=-3
    int32 TIMEOUT=-4
    string observation_msg              # a brief message
    string verbose_observation_msg      # description of taken obs.
    int32 observation                   # observation result
```

6.2.2 Observer Manager

All running observers are organized by the observer manager. All observers are designed as plug-ins loadable by the observer manager. Two identical managers were necessary to support observers implemented in C++ and python. These two programming languages offer different capabilities, which can be useful for monitoring different properties. Both managers offer a plug-in base implementation, which is used by all plug-ins. Another part of the managers is to subscribe to all needed topics and forward them to the observers. Hence each topic is only subscribed once even if several different observes need them. This seems to be more efficient than let all observers subscribe to the topic by its own. This was necessary because of the internal organization of the ROS framework, which needs a lot of system resources if many subscribers subscribe to a topic where many messages are send with a high frequency.

At start time of the managers, they get the configurations of the observers, which are instantly loaded and configured. Currently observers need to be configured before the managers are started, hence no later dynamic loading of observers is supported. Also dynamic configuration changes of observers are not supported. This restriction is caused by the used plug-in library, which only supports loading of plug-ins. Unloading plug-ins are not supported.

6.2.3 Observer Functionalities

Each observer plug-in has to validate one or more input channels. In some chases it is necessary to filter this inputs to get more meaningful information. Afterwards the observers have to validate the inputs if they meet certain requirements. Different hypotheses checks are implemented.

Filter

The observers get the messages of a topics. Afterwards they extract the information that are necessary for its type of monitoring. For instance, the frequency observer just measures the time between the arrived messages, while the value observer has to analyze the content of a message. All observers have in common that they may have to filter their measurements. Therefore, this library was implemented for filtering integer and floating values. Its interface offers three functions, which are update (add a new value), get (read the filter value), and reset (reinitialize the filter). Observers can use filters to smooth different values. All filters are available as library or script for C++ and python. This makes it easy to use for different data types.

All different filters are listed below.

No filter Here no filtering is done, the output is equal the input.

```
type: nofilter
```

Mean filter A *window_size* can be defined. Otherwise all measurements ever inserted are part of the mean.

```
type: mean
window_size: 200                        # optional
```

Median filter A *window_size* need to be defined.

```
type: median
window_size: 200
```

Kmeans filter A *window_size* need to be defined for the median. Additional a *k_size* need to be defined for the mean calculation around the median.

```
type: kmeans
window_size: 200
k_size: 10
```

EWMA filter This is the exponentially weighted moving average filter. The *decay_rate* is the blending coefficient to combine a new measurement to the old ones. This filter uses

$$x_{filter} = x_{filter} \cdot (1.0 - decay\_rate) + x_{measurement} \cdot decay\_rate.$$

A *window_size* can also be defined. This forces to calculate the ewma always new from the beginning of the list, because with each new measurement, the list changes.

```
type: ewma
window_size: 200                                    # optional
decay_rate: 0.05
```

Additional filters are available for calculating the deviation of the data. The first one is the standard deviation, which calculates the deviation of the measurements. The second one searches the smallest and the larges value of the measurements. Both uses the same measurements as used for the filters, so the same *window_size* is used.

Hypotheses Checks

Hypotheses checks are necessary to validate that given measurements meets a certain requirement. Two different test methods are available.

The observers get the messages of a topics. Afterwards they extract the information that are necessary for its type of monitoring. The observers can filter the measurements. Afterwards the measurements are checked with one of the following tests.

The first one is the Student T-Test [31]. For this check it is necessary to use the mean and the standard deviation filters. Additionally, several parameters need to be specified: true_mean, standard_deviation, and significance_level. These parameters specifies the values the measurement ideally should have.

```
type: student_t
true_mean: 0.10
std_deviation: 0.0003
significance_level: 0.05
```

The second available test compares the measurements with nominal values. For this procedure different tests are available.

Normal distribution This test checks if the measurement is within the one-sigma uncertainty of a normal distribution.

```
type: gauss
mean: 0.1                           # mean of the distribution
std_deviation: 0.25                 # std dev of the distribution
```

Exact The measurement value need to be exact the specified value. This test supports integers.

```
type: exact
exact: 1                            # the value to meet
```

Different The measurement value need to be different from the specified value. This test supports integers.

```
type: not
exact_not: 1                        # the value to be different from
```

Greater than The measurement value need to be greater than the specified value.

```
type: greater_than
greater_than: 0.1                   # the value to be greater than
```

Less than The measurement value need to be less than the specified value.

```
type: less_than
less_than: 0.1                      # the value to be less than
```

In between The measurement value need to be in between the two specified values.

```
type: in_between
lower_bound: 2.0                    # lower bound of the interval
upper_bound: 5.1                    # upper bound of the interval
```

Not in between The measurement value need not to be in between the two specified values.

```
type: not_in_between
lower_bound: 2.0                    # lower bound of the interval
upper_bound: 5.1                    # upper bound of the interval
```

6.2.4 Resource Monitor

Some observers may need meta information about nodes that are only known by the operating system. The task of the resource monitor is get the CPU and the memory usage of all processes that are used by the robot. Therefore, it requests the names and corresponding process identifier (PID) of all nodes known in by the ROS-master, which is the central management component of ROS. All nodes, topics, and services are registered in the ROS-master. The monitor is written in python and uses the python-library psutil[3]. In combination with the known PIDs the resource monitor is able to get access to the mentioned information. All collected usages are published by using the defined message as listed in Listing 6.2.

This messages includes a list of a structure called NodeInfo which contains information about each individual node. This include not only the CPU and memory usage, but also the PID and the hostname for identifying the host, where the node is running.

Listing 6.2: Resource Info message

```
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
tug_resource_monitor/NodeInfo[] data
  string name
  uint32 pid
  string hostname
  float32 cpu
  uint64 memory
```

6.2.5 Observer Plug-ins

Observers are able to be implemented in C++ or in python, but both have in common to use the provided base for observers and both need to be usable as plug-in. Each observer has its own configuration with its own necessary parameters.

[3]`https://pypi.python.org/pypi/psutil`

The configurations of an observer define valid and also forbidden behaviors of the monitored node(s) or topic(s). If an observer is not able to find an allowed behavior the observation reports a faulty behavior. For the configuration such a behavior is called state and has a name and a number. This number is used in the observer message for the observation result. As already mentioned in the description of the observer message in Section 6.2.1 behaviors with negative numbers are classified as faulty and positive are classified as valid states.

An additional feature of this implementation is that the observers and the diagnosis engine can differ between nodes that are publishing on the same topic. For instance, if node A and node B publish on topic 1, the message can not be assigned to a publisher, but the meta-information refers to a caller_id which is the node name.

Resource Observer

This observer is used to check the CPU and memory consummation of nodes. Therefore, it is necessary, that the resource monitor, described in the previous section, is running. The resource monitor publishes information about all running nodes and this observer has to subscribe this topic. If the resource monitor is not running, this observer will not publish any observations. Each node need to be separately defined with its own filter for CPU usage and an own filter for memory usage. Further, several states per node can be specified.

```
- type: resources
  start_up_time: 10.0                 # wait till first obseration
  resource_topic: /diag/node_infos    # topic of resource information
  nodes:                              # set of nodes to observe
  - name: /nodeA                      # name of the node
    cpu_filter: <filter configuration>  # filter conf. for the CPU
    mem_filter: <filter configuration>  # filter conf. for the memory
    states:                           # set of possible states
    - state: 'nameXY'                 # name of state
      number: 1                       # number of state, negative
                                      # numbers defines faulty states
      cpu: <hypothesis check>         # hypothesis check for the CPU
      memory: <hypothesis check>      # hypothesis check for the memory
```

Component Observer

This observer is used to observe if all predefined nodes are running. Therefore, this observer requires a running resource monitor, because all running nodes are included in resource information message. If a node is not included, it is not registered in the ROS-master. Missing nodes are published as faulty. This observer only monitor if a node exists without checking for valid or faulty behaviors.

```
- type: component
  start_up_time: 10.0                 # wait till first obseration
  resource_topic: /diag/node_infos    # topic of resource information
  nodes:                              # set of nodes to observe
  - name: /nodeA                      # name of the node
```

Frequency Observer

The Hz observer measures the frequency of messages of a topic. This observer also includes caller ids, which names the publishing node of a message. If several nodes publish to the same topic, it will be possible to calculate the frequency for each of them separately.

```
- type: frequency
  main_loop_rate: 1.0                 # observation rate
  start_up_time: 10.0                 # wait till first obseration
  topics:                             # set of topics to observe
  - name: /topicA                     # name of the topic
    callerids:                        # parameter for diff. caller ids
    - callerid: [/nodeA]              # list of caller ids
      filter: <filter configuration>  # filter conf. for frequency
      states:                         # set of possible states
      - state: 'state1'               # name of state
        number: 1                     # number of state, negative
                                      # numbers define faulty states
        frequency: <hypothesis check> # hypothesis check for frequency
```

This example presents different possible configurations for a topic published by several nodes. The first one is that all messages, independent from its publisher, are used for the frequency measurement. The second example configuration restricts the measurement to messages published by the node with the name */nodeA*. The last example shows the configuration for measuring the frequency of messages published from several nodes. Here messages from */nodeA* and */nodeB* are used for the frequency measurement.

```
    ...
    callerids:                          # parameter for diff. caller ids
   - callerid: []                       # list of caller ids
      ...
   - callerid: [/nodeA]                 # list of caller ids
      ...
   - callerid: [/nodeA, /nodeB]         # list of caller ids
      ...
```

Timing Observer

This observer measures the time between messages of two different topics. For a detailed description see Paragraph **Timing Observer** in Section 5.2.2

```
- type: timing
  main_loop_rate: 1.0                   # observation rate
  topics:                               # set of topics to observe
  - topicA: /topicA                     # name of the first topic
    calleridA: []                       # list of caller ids fist topic
    topicB: /topicB                     # name of the second topic
    calleridB: []                       # list of caller ids second topic
    single_shot_mode: true              # true for not continuous msgs
    filter: <filter configuration>      # filter conf. for frequency
    states:                             # set of possible states
      - state: 'state1'                 # name of state
        number: 1                       # number of state, negative
                                        # numbers define faulty states
        delay: <hypothesis check>       # hypothesis check for score
```

Time-Out Observer

If it is necessary, that messages are published in time. The time-out observer is able to monitor this property. This observer is used as watchdog for topics. It measures the time between messages. If the difference is to large or no message is send in time the observer reports an error. This observer includes caller ids, which offers the possibility to define a time-out configuration for a restricted number of nodes.

```
- type: timeout
  start_up_time: 10.0                # wait till first observation
  topics:                            # set of topics to observe
  - name: /topicA                    # name of the topic
    callerids:                       # parameter for diff. caller ids
    - callerid: [/nodeA]             # list of caller ids
      timeout: 1.0                   # maximum time between messages
      max_timeouts_in_a_row: 2       # how often reporting time-outs
```

Time-Stamp Observer

This observer reads the time-stamp of a message. If the time-stamp in the message it is to old, the observer reports a fail.

```
- type: timestamp
  main_loop_rate: 1.0                # observation rate
  topics:                            # set of topics to observe
  - name: /topicA                    # name of the topic
    callerids:                       # parameter for diff. caller ids
    - callerid: []                   # list of caller ids
      filter: <filter configuration> # filter conf. for frequency
      states:                        # set of possible states
      - state: 'state1'              # name of state
        number: 1                    # number of state, negative
                                     # numbers define faulty states
        age: <hypothesis check>      # hypothesis check for age
```

Score Observer (Value Observer)

The score observer is one implementation of a value observer. It is used to monitor floating-point value of message. Observed messages need to be of type float. Structured data is not supported.

```
- type: scores
  main_loop_rate: 1.0                # observation rate
  start_up_time: 10.0                # wait till first observation
  topics:                            # set of topics to observe
  - name: /topicA                    # name of the topic
    filter: <filter configuration>   # filter conf. for frequency
    states:                          # set of possible states
      - state: 'state1'              # name of state
        number: 1                    # number of state, negative
                                     # numbers define faulty states
        score: <hypothesis check>    # hypothesis check for score
```

6.3 Diagnosis and Repair

This section describes the diagnosis to estimate possible faulty nodes, if observers report a faulty behavior. With this knowledge about possible faulty nodes the rule engine searches for actions to be execute in order to repair the system or inform human beings.

6.3.1 PyMBD

The diagnosis engine is based on the framework PyMBD, which is actual a Python library for testing and experimenting with different established model-based diagnosis algorithms. This library is described in detail in [25].

For this work some changes of the framework were necessary, but not for the central functionality. For example, all but one implemented algorithms are removed because only one algorithm is required and to reduce the complexity of the engine. Furthermore, the framework was designed to diagnose logic circuits rather than a robot using observations from different observers. Hence, to receive the messages from the observers, it was necessary to add the support for ROS and for continuous changing observations. The diagnosis now is calculated recurrently, using the last known observations. This is different to the original implementation, where this information was set hard-coded and the diagnosis was calculated once for each algorithm. The input file, which was required before as system description to generate the diagnosis model, contained the information about input and output variables and how they are connected by using logic gates. This had to be changed because it was necessary to update the system description with each started or stopped node. At startup time no nodes are running, therefore the system description is empty.

6.3.2 Configuration and System Description Validation

At start-time of the diagnosis the system description is empty. With the first started node a configuration update is send to the diagnosis engine.

This includes two parts:

node configuration This configuration names each node and list its published and subscribed topics. With the information about the topics, the dependencies between the nodes are defined, as described in Section 5.2.3. Listing 6.3 show a part of the later described ROS-service which is used for updating the system description.

Listing 6.3: Node configuration

```
tug_diagnosis_msgs/node_configuration[] nodes
  string name                # name a node
  string[] sub_topic         # name of topics sub. by the node
  string[] pub_topic         # name of topics pub. by the node
```

observer configuration This configuration names each used observation type and which topic or node they are observing. For example, a frequency observation is added by declare the type as "frequency" and give one topic which frequency is observed. To add another topic, which frequency is also observed, it need to be added with another observer configuration. Listing 6.4 show a part of the later described ROS-service which is used for updating the system description.

Listing 6.4: Observer configuration

```
tug_diagnosis_msgs/observer_configuration[] observers
  string type                # type of observation
  string[] resource          # resource that is observed
```

For a better explanation the same example as in Section 5.2.3 is used. For this example, the nodes $N1$ and $N2$ are already running and node $N3$ is started. Therefore, the node and observer configuration for node $N3$ looks like shown in Listing 6.5.

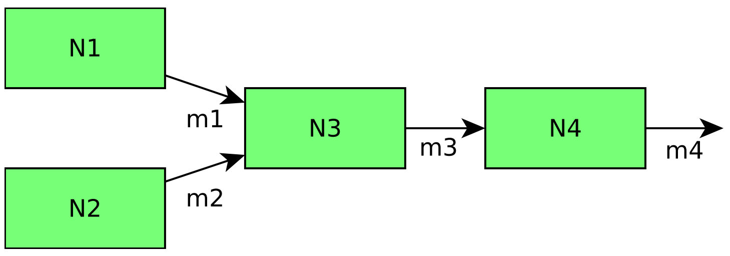


Figure 6.3: Example for updating the system description. Nodes are named as *N*1, *N*2, *N*3, and *N*4. Topics are named *m*1, *m*2, *m*3, and *m*4.

The Listings 6.6 and 6.6 show the system description before and after the update.

Listing 6.5: Configuration that should be added to the system description

```
nodes:
  - name: N3
    sub_topic: [m1, m2]
    pub_topic: [m3]
observers
  - type: frequency
    resource: [m3]
  - type: resource
    resource: [N3]
```

Listing 6.6: System description before the update

```
nodes:
  - name: N1
    sub_topic: []
    pub_topic: [m1]
  - name: N2
    sub_topic: []
    pub_topic: [m2]
observers
  - type: frequency
    resource: [m1]
  - type: frequency
    resource: [m2]
```

Listing 6.7: System description after the update

```
nodes:
  - name: N1
    sub_topic: []
    pub_topic: [m1]
  - name: N2
    sub_topic: []
    pub_topic: [m2]
  - name: N3
    sub_topic: [m1, m2]
    pub_topic: [m3]
observers
  - type: frequency
    resource: [m1]
  - type: frequency
    resource: [m2]
  - type: frequency
    resource: [m3]
  - type: resource
    resource: [N3]
```

If a node is stopped the diagnosis engine gets the request to remove parts of its system description. Continue the example by stopping node $N3$. The configuration for removing node $N3$ looks like shown in Listing 6.5, which is the same configuration that was used for adding node $N3$. This time the diagnosis engine has to search for parts of its system description that confirms with the request and removes it. The resulting system description looks like the system description shown in Listing 6.6 which is the same configuration before adding node $N3$.

For changing the system description of the diagnosis engine a ROS-service is used. The service also includes the information if the configuration should be added or removed form the system description. The service-message is described in Section 6.3.3.

System Description Validation

Changing the system description may cause an inconsistent generated diagnosis model. A diagnosis calculation is only possible with a consistent model. Therefore, it is necessary to validate the description after each change. This validation includes several tests, the most important once are listed below.

- All set node and topic names used in the observer configuration need to be already known by the system description. Hence it is necessary to define the node in the node configuration and topics need to be defined as "published topic" in the node configuration. Otherwise the diagnosis will not be able to judge the node as faulty, even if a topic is faulty, which can only caused by a publishing node.
- There is also a restriction for node and topic names. For instance, the names of topics and nodes are not allowed to start with the characters "AB" or "/AB" because these are used by the diagnosis engine itself. Using such names may cause unpredictable diagnosis results.

After the validation has finished without errors the system description is released for the generation of diagnosis model which is afterwards usable by the diagnosis engine. If the system description is not consistent, no diagnosis model is generated. Therefore, as long as the system description is inconsistent no diagnosis is calculated.

6.3.3 Messages

Two different messages need to be defined for the diagnosis. The first one is shown in Listing 6.8 and defines the structured data used by the diagnosis to publish its results. The diagnosis may have more than one possible sets of node names, because different combinations of faulty nodes can explain the same observed situation. Therefore, for example, the diagnosis may came up with a set including $node_A, node_B$ and with another set $node_F, node_A$ where both sets are plausible explanations for a current faulty situation.

Listing 6.8: Diagnosis message

```
std_msgs/Header header                    # ros header message
  uint32 seq
  time stamp
  string frame_id
string type                               # name of used solver
tug_diagnosis_msgs/diagnosis[] diagnoses
  tug_diagnosis_msgs/resource_mode_assignement[] diagnosis
    int32 GENERAL_OK=0
    int32 GENERAL_ERROR=-1
    string resource                       # estimatet faulty node name
    string mode_msg                       # a brief message
    int32 mode                            # diagnosed mode
```

The second definitions is required for the used ROS-service for updating the diagnosis model when nodes are started or stopped as already described in Section 5.2.3. This service message is shown in Listing 6.9. The service includes a request message part (upper section) and a response part (lower section).

With each update the system description is changed and with each change of the system description a new diagnosis model is generated.

The request includes the configuration of nodes and observers that should be changed in the system description of the diagnosis engine. In addition to the configuration it needs to be specified how the configuration should be interpreted –should it be added or removed from the system description.

The response part of the service is used to inform about the success of the system description update and the consistency of the model, which is validated after each change of the system description to guarantee a consistent model.

Listing 6.9: Model Configuration service

```
int32 ADD=1
int32 REMOVE=2
tug_diagnosis_msgs/configuration config
  tug_diagnosis_msgs/node_configuration[] nodes
  tug_diagnosis_msgs/observer_configuration[] observers
int32 action
---
int32 NO_ERROR=0
int32 GENERAL_ERROR=-1
int32 CONFIG_INVALID=-2
int32 errorcode
string error_msg
```

6.3.4 Rule Based Repair

For processing observations or diagnosis a simple rule system for triggering actions is implemented. For that, rules need to be defined at start time of the rule engine by using the following yaml-configuration:

```
rules:
  - type: <type of the rule>          # type of rule to trigger
    <rule type specific configuration> # type-depending configuration
    <query>                            # rules for trigger
    recall_duration: 10.0              # delay between trigger again
    single_shot: true                  # only allow trigger once
```

This configuration defines a type, the type specific configuration, a query, and further parameters. The rule engine is able to trigger different actions like write a message into a log file, send an email, or change a parameter. The parameter type of the configuration defines, which action should be used for this rule. If the type need type specific parameters, they also need to be defined.

The query defines the conditions, when the rule has to trigger. Therefore, there are four possible fields, two for observation trigger rules and two for diagnosis trigger rules. The first two are **positive_observations**, which defines observations to trigger, and **negative_observations**, which defines observations not to trigger. Listing 6.10 shows the configuration of queries for observation conditions. The parameters *occurrences* and *window_size* are optional. It is not necessary to define both *observation* and *observation_msg* at the same time.

Listing 6.10: Queries for observation conditions

```
positive_observations:
  - type: 'frequency'          # observation type
    resource: 'm3'             # observed resource
    observation: -1            # observation number
    observation_msg: 'error'   # observation message
    occurrences: 3             # number of valid observation in a row
    window_size: 3             # window to count occurences
negative_observations:
  - type: 'frequency'          # observation type
    resource: 'm3'             # observed resource
    observation: -1            # observation number
    observation_msg: 'error'   # observation message
    occurrences: 3             # number of valid observation in a row
    window_size: 3             # window to count occurences
```

The second two are **positive_possible_faulty_resources** and **negative_possible_faulty_resources**, where the first one defines conditions for nodes that need to be faulty and the second defines conditions for nodes that need to not faulty. Both need parameters like *resource*, which can be the node name, to define those conditions. Listing 6.11 shows the configuration of queries for the diagnosis conditions.

Listing 6.11: Queries for diagnosis conditions

```
positive_possible_faulty_resources:
  - resource: 'N3'               # faulty estimanted resource name
    occurrences: 3               # number of faulty estimation in a row
    window_size: 3               # window to count occurences
negative_possible_faulty_resources:
  - resource: 'N3'               # faulty estimanted resource name
    occurrences: 3               # number of faulty estimation in a row
    window_size: 3               # window to count occurences
```

The rule engine is structured as shown in Figure 6.4. With all rules

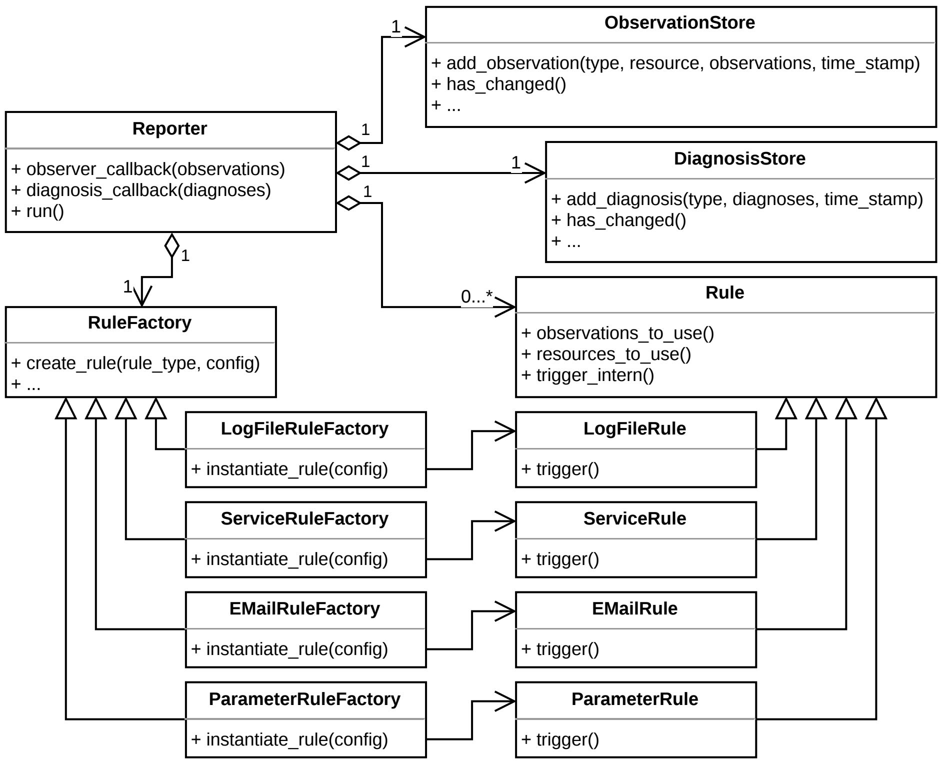


Figure 6.4: Class diagram of the reporter.

defined the reporter uses the rule factory to create objects for each defined rule. All rules are stored in a list by the reporter. Each new observation message is stored in the observation store. The same applies to new diagnosis messages, which are stored in the diagnosis store. With each changing in one of the two stores all rules in the list are asked if they trigger. If a rule triggers it executes its defined action.

New actions can easily be added to the rule engine. Therefore, two parts need to be implemented for each new rule type. The first part is a class that need to be derived from *Rule* as shown in Listing 6.12. This class has to include the function *trigger* which is executed if this rule is executed.

Listing 6.12: Derived class for new rule

```python
class NewRule(Rule):
  def __init__(self, positive_observations, negative_observations,
               positive_possible_faulty_resources,
               negative_possible_faulty_resources, recall_duration,
               is_single_shot, new_rule_data):
    super(NewRule, self).__init__(positive_observations,
                                  negative_observations,
                                  positive_possible_faulty_resources,
                                  negative_possible_faulty_resources,
                                  recall_duration, is_single_shot)
    # store rule specific data
    self._new_rule_data = new_rule_data

  def trigger(self):
    super(NewRule, self).trigger_intern()

    # process rule specific action
    pass
```

The second part is a class that need to be derived from *RuleFactory* as shown in Listing 6.13. This class is necessary to read the configuration for the new rule and to create an instance of the rule.

Listing 6.13: Derived class for new rule factory

```python
class NewRuleFactory(RuleFactory):
  @staticmethod
  def instantiate_rule(config):
    # read configuration for:
    # pos_observations = RuleFactory.pars_positive_observations(config)
    # neg_observations = RuleFactory.pars_negative_observations(config)
    # pos_possible_faulty_resources =
    #         RuleFactory.pars_positive_possible_faulty_resources(config)
    # neg_possible_faulty_resources =
    #         RuleFactory.pars_negative_possible_faulty_resources(config)
    # is_single_shot, recall_duration, new_rule_data

    return NewRule(pos_observations, neg_observations,
             pos_possible_faulty_resources, neg_possible_faulty_resources,
             recall_duration, is_single_shot, new_rule_data)
```

The last step is to add the new rule factory class to the basic rule factory class, shown in Listing 6.14. Hence the new rule is known by the rule engine.

Listing 6.14: Code snippet for adding the new rule factory

```
class RuleFactory(object):
  _factory_map =
  {
    ...
    'newrule': lambda config: NewRuleFactory.instantiate_rule(config),
    ...
  }
```

7 Evaluation

This chapter contains a qualitative use case with four different scenarios. It should show hot the system tries to keep the robot alive. The first scenario demonstrates the stop of a **task** if its low-level components are stopped because of a fault. After that another scenario demonstrates how a low-level component is replaced with an equivalent one. The third and the fourth scenarios show the behavior of the architecture with shareable and non-shareable low-level components. But before the use case is described in detail.

7.1 General

7.1.1 Description

For this use case a simple architecture was chosen with five nodes used as **functionalities** and two nodes as **behaviors**. Figure 7.1 shows all used nodes and how they are connected by topics. There connections define the dependencies between the nodes.

The whole use case is simulated by using dummy nodes. Hence no real robot was used. All nodes are only able to subscribe and/or publish messages on topics. They do not offer other functionalities. The nodes used for the **behaviors** are finishing after a certain runtime and return. The remaining nodes that are used for the **functionalities** are not terminating and are running continuously till an external termination request.

The following nodes are used as functionalities:

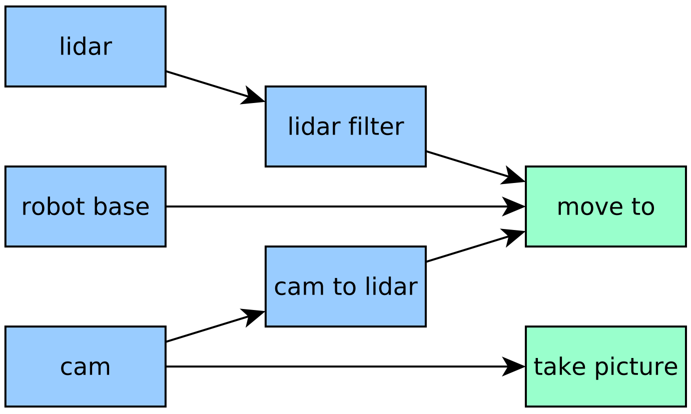


Figure 7.1: Dependencies of **functionalities** and **behaviors** of the use case. **Functionalities** are shown in blue, **behaviors** are shown in green.

lidar This node emulates a Lidar and publish Lidar-like information.

lidar filter The Lidar filter is emulating a node that is able to filter the raw-data from the Lidar.

robot base The robot base publishes dummy messages to emulates the sensors, like odometry, of the robot.

cam The node cam is used to simulate a camera.

cam to lidar The last used functionality is the cam to lidar converter. It subscribes to the topic of the cam and publishes emulated Lidar-like information on the same topic as the Lidar filter.

Two further nodes, used as behaviors, need to be described. Both nodes start a ROS-action server and waits for new action-goals. If a new goal arrives the node waits some seconds to emulate a behavior the robot movement. Afterwards an action-result is send and the **task executer** stops. Generally, nodes for **behaviors** would terminate after it execution finishes. But this two nodes need to be terminated by the **task executer** because the action servers need to be stopped. The termination of a behavior is done automatically by the **task executer** if the node is still running even it has send its results.

move to This node emulates the behavior of moving the robot to a defined goal position.
take picture For emulating the behavior of taking a picture this node is implemented.

For this use case the knowledge base is a storage class and stores

- information which **behaviors** is used by which **task**,
- which **functionalities** are needed for the **behaviors**,
- the configuration of the diagnosis,
- the rating of nodes,
- the fault counters of nodes,
- and the information of shareable and non-shareable nodes.

7.1.2 Setup

For this use case several components are used:

- the resource monitor for monitoring the resources consummation of running nodes,
- the observer managers for C++ and Python observers,
- the diagnosis engine for calculating diagnoses if properties are observed as faulty,
- the **task executer** as central part for starting and stopping **functionalities** and **behaviors**,
- and a text-based user interface for starting, stopping, and restarting **tasks**.

All configurations for the observers are predefines as yaml-configurations. The observers are started and initialized after the observer managers are started.

The configuration of the diagnosis for all nodes is also predefined and is stored in the knowledge base.

With the text-based user interface an user can execute a **task** without the need of a **plan executer**. It uses the same interfaces for starting and stopping **task** the **plan executer** would use. The used interface for restarting a **task** would also be used the rule engine. Therefore, the user is able to simulate the request a restart of a node that is estimated as faulty.

7.1.3 Test procedure

After all components of the use case are started the user can select between two **tasks**. The user selects a **task** it wants to be executed. The name of the **task** is send to the **task executer**. The **task executer** sends a request to the knowledge base to get a suitable **behavior**, which has the highest rating for this **task**. The rating of **behavior** is done by the knowledge base. At startup all nodes have a predefined rating. If a node is estimated as faulty its rating is decreased. Additionally, the fault-counter of the node is increased. Before the **behavior** is executed the **task executer** has to prepare the system. Therefore, it sends a request to the knowledge base to get all needed **functionalities** for the **task**. The returned list of **functionalities** are executed. After system is prepared the the **task executer** executes the **behavior**.

With each started execution of nodes the **task executer** has to update diagnosis engine.

While a **task** is executed the user is able to start further **tasks**, stop running **tasks**, or mark a **task** as faulty and restart it. The different scenarios of this use case show the behavior of this architecture.

7.2 Scenario 1

For the first scenario only a subset of the node are used as shown in Figure 7.2. This scenario demonstrates how the **task executer** deals with not replaceable **functionalities** that are estimated as faulty. Not replaceable means that there is no alternative **functionality** that can replace the faulty one.

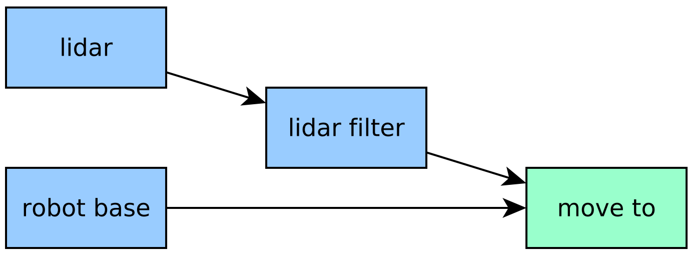


Figure 7.2: Used nodes for the first scenario.

Some additional information for this scenario:

- The maximal number a node is allowed to be faulty is three.
- The **behavior** of *move to* needs to be stopped and restarted if the system configuration needs to be repaired.

The following steps describe the test procedure of this scenario:

1. The user selects the **task** *move to*.
2. The **task executer** prepares the system by starting the *lidar*, *robot base*, and *lidar filter*.
3. Afterwards the **behavior** of the **task** is executed.
4. To simulate a continuous faulty Lidar, the user continuously requests to restart the node *lidar*.

At the first time the *lidar* was estimated as faulty the **task executer** stops the execution of *move to*. The faulty **functionality** is marked as faulty for the first time (increasing the fault-counter). In the interests of simplification, the **task executer** stops also other **functionalities** used by this **task**. If no component is running any more, the **task** is restarted, which requires a prepared system configuration. After that the **behavior** is executed again.

But the *lidar* stays faulty. The whole procedure from above is repeated. This time the counter of the faulty **functionality** is increased to two.

After the third time the *lidar* is faulty the counter increases to three and now the **functionality** is marked as permanent faulty because it reached the maximal number of allowed faults. It cannot be used any more.

The **task executer** is not able to prepare the system configuration anymore, because it gets no possible **functionality** because no node is defined which can replace the defective *lidar*. Therefore, the **task executer** has to cancel the execution of the **task**.

7.3 Scenario 2

This scenario is similar to the first use case, with the difference that this time there is also a second **functionality** that can offer **lidar**-like information. Therefore, *cam to lidar*, which needs *cam*, can be used in case the *lidar* is marked as permanent faulty. See Figure 7.3, which shows the used architecture with all used nodes. This scenario demonstrates how the **task executer** deals with replaceable **functionalities** that are estimated as faulty.

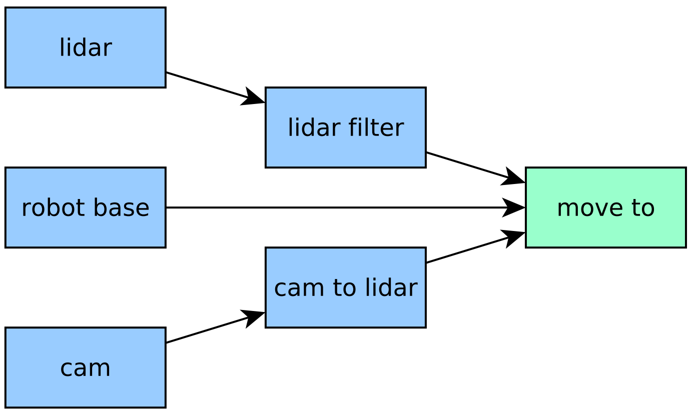


Figure 7.3: Used nodes for the second scenario.

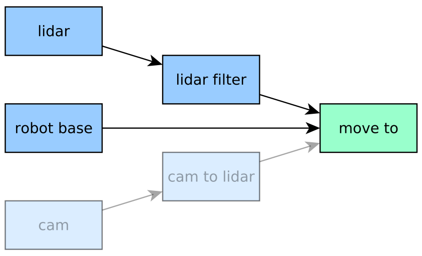
Some additional information for the use case setup:

- The maximal number a node is allowed to be faulty is three.

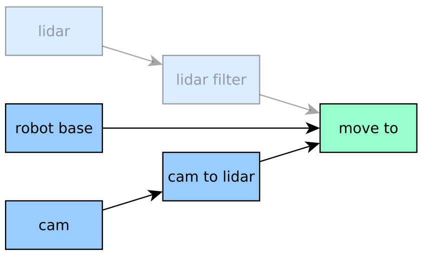
- The **behavior** of *move to* needs to be stopped and restarted if the system configuration needs to be repaired.
- The **task executer** always gets the **functionalities** *lidar filter* and *lidar* first, because they have a higher rating than *cam* and *cam to lidar*.

The test procedure of this scenario is the same as described in the first scenario. Also the behavior of the architecture for the first two times, where the *lidar* is faulty, is the same as in the first scenario. The system configuration looks like shown in Figure 7.4a. After the third time the *lidar* is estimated as faulty it is marked as permanent faulty and this node can not be used any more. Because *lidar filter* required *lidar* also this node is marked as permanent faulty.

This time the **task executer** gets the **functionalities** named *cam to lidar* and *cam*, which are able to replace the *lidar filter* and the *lidar*. After starting **functionality** *cam* and *cam to lidar* another valid system configuration is prepared to execute the **behavior** again. Now the system configuration look like shown in Figure 7.4b.



(a)System configuration before *lidar* is marked as permanent faulty.



(b)System configuration after *lidar* is marked as permanent faulty.

Figure 7.4: System configuration change if *lidar* is marked as permanent faulty. Transparent components are inactive and not used by the **task executer**.

7.4 Scenario 3

The third scenario demonstrates how the **task executer** handles several executions of **tasks** that are not executable at the same time. For this, the system is shows in Figure 7.5.

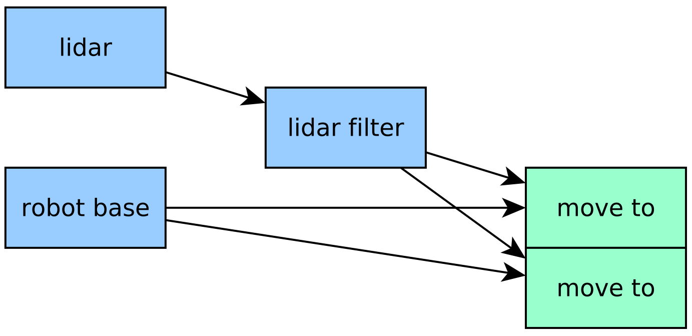


Figure 7.5: Used nodes for the third scenario.

Some additional information for the use case setup:

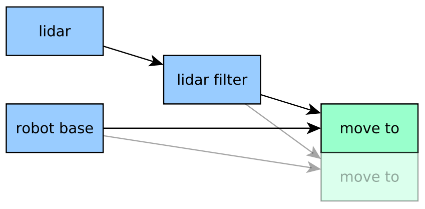
- The **functionalities** *lidar* and *lidar filter* are shareable.
- The **functionality** *robot base* is non-shareable.

The following steps describe the test procedure of this scenario:

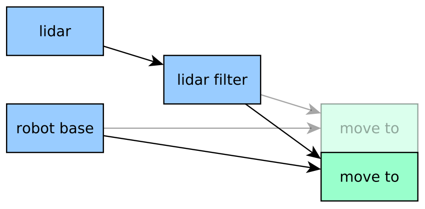
1. The user selects the **task** *move to*.
2. The **task executer** prepares the system by starting the *lidar*, *robot base*, and *lidar filter*.
3. Afterwards the **behavior** of the **task** is executed.
4. The user selects the **task** *move to* another time while it is still running.

The **task executer** tries to prepare the system configuration for the second **task**. But because the *robot base* is not shareable, the execution of the second **task** need to be suspended, see Figure 7.6a. The robot is not able to execute both **tasks** simultaneously, hence they are scheduled and executed sequentially.

After the first *move to* finishes the second is executed, Figure 7.6b. In the interests of simplification, all **functionalities** are stopped after the first **task** and are restarted for the second **task**.



(a)System configuration for first *move to*.



(b)System configuration for second *move to*.

Figure 7.6: System configuration for sequential execution of *move to*s that use non-shareable nodes. Transparent nodes are inactive and not executed/suspended.

7.5 Scenario 4

For the forth scenario the whole architecture, which was described at the beginning of this chapter, is used. Figure 7.7 shows the architecture again. This scenario shows how the **task executer** switches from a system configuration with a faulty *lidar* to a configuration with the **functionality** *cam to lidar* even if the *cam* is already used by the **task** *take picture*.

Some additional information for the use case setup:

- The maximal number a node is allowed to be faulty is three.
- The **behavior** of *move to* and *take picture* need to be stopped and restarted if the system configuration needs to be repaired.
- The **task executer** always gets the **functionalities** *lidar filter* and *lidar* first, because they have a higher rating than *cam* and *cam to lidar*.
- The **functionalities** *lidar*, *lidar filter*, *cam*, and *cam to lidar* are shareable.

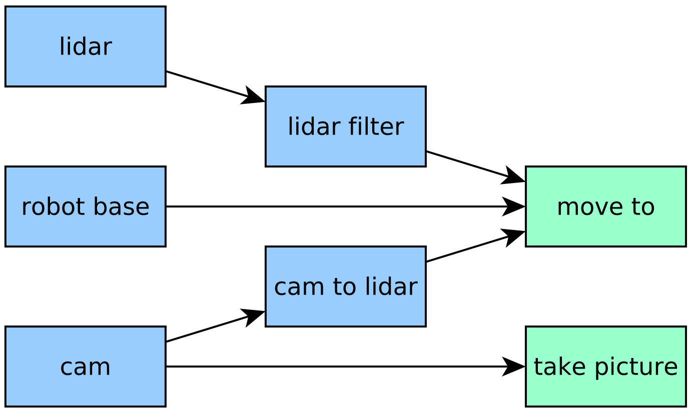


Figure 7.7: Used nodes for the fourth scenario.

- The **functionality** *robot base* is non-shareable.

The following steps describe the test procedure of this scenario:

1. The user selects the **task** *move to*.
2. The **task executer** prepares the system by starting the *lidar*, *robot base*, and *lidar filter*.
3. Afterwards the **behavior** of the **task** is executed.
4. The user selects the **task** *take picture* frequently while the node *move to* is running.

   4.1. The **task executer** starts the *cam* to prepares the system.
   4.2. Afterwards the **behavior** of the **task** is executed.

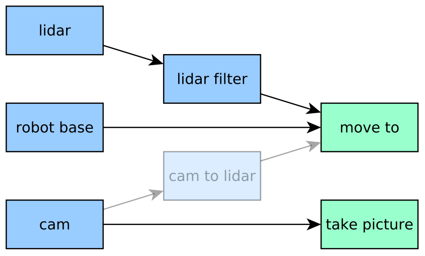
5. To simulate a continuous faulty Lidar, the user also continuously requests to restart the node *lidar*.

This scenario is an extension of the second scenario described in Section 7.3. As long as the *lidar* is not marked as permanent faulty the system configuration looks like shown in Figure 7.8a.

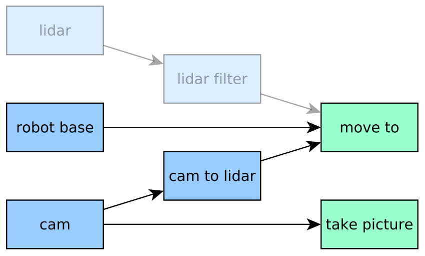
If the request for restarting the node *lidar* arrives the third time, *lidar* and *lidar filter* are marked as permanent faulty, as it is described in the first two scenarios.

This time the **task executer** gets the **functionalities** named *cam to lidar* and *cam*. After starting **functionality** *cam* and *cam to lidar* another valid system configuration is prepared to execute the **behavior** *move to* again.

Now, if the **task** *take picture* is started by the user, the **functionality** *cam* is already running because it us used for the system configuration of *move to*. However the **task executer** starts with the execution of the **behavior** of *take picture* because *cam* is a shareable **functionality**, see Figure 7.8b.



(a)System configuration before *lidar* is marked as permanent faulty.



(b)System configuration after *lidar* is marked as permanent faulty.

Figure 7.8: System configuration change if nodes are marked as permanent faulty while other **tasks** are running. Transparent components are inactive and not used by the **task executer**.

The **functionality** *cam* is running as long as at least one of the two **tasks**, which requires it for their system configuration, are executed by the **task executer**.

8 Conclusion and Future Work

This chapter summarizes the work presented in this thesis and in addition possible improvements and future work are discussed.

This thesis presented an architecture for autonomous robots with the ability to observe, diagnose, and repair components. Further software components can easily be added and are usable by the architecture. The architecture also supports redundant software modules. Therefore, not repairable modules can be replaced.

First all necessary parts of the architecture are defined by giving a formal problem formulation, which mainly depends on the definitions used by Ghallab, et al. [3]. This definition and the description about consistency-based diagnosis are used to describe the interactions and work flow of the universal architecture. This includes also the interactions between the different components of the different layers. Furthermore, the procedure of monitoring components and the diagnosis engine, which collects all the results of the observers and calculates a hitting set of possible faulty components, was described. Finally, a repair engine was presented that is used to collect the observation results and the results of the diagnosis engine and sends commands for the repairing procedure. Further chapters describe the implementation including the implementation of mid-layer and low-level parts of the architecture, the observers, the diagnosis engine, and a rule-based repair engine.

The qualitative evaluation of this thesis includes an use case with four different scenarios that show the potential of this architecture, even if the implementation only includes the **task executer** of the mid-layer and low-level components. Attention was also given to the observers and the diagnosis engine. During the test of the different scenarios is always observes faulty working components and the diagnosis calculated correct estimations about faulty components. The use case shows that the architecture is able to handle **tasks** that are estimated as faulty and restart them. Tests also showed that the **task executer** is able to switch between different system configurations for the same **task**. Some **tasks** are not executable at the same time because of intersections of their system configurations. The last two use cases show that the **task executer** is able to manage shareable and non-shareable components of the low-level. Therefore, **tasks** are executed simultaneously or in a row.

Some drawbacks of ROS are recognized during the implementation for this thesis. Some core functionalities of ROS are not well-engineered. For example, the implementation used for the communication between nodes is inefficient and needs a lot of system resources independent from the content of the transferred data.

8.1 Future Work

Several observers for different properties of low-level components are presented in this work, but hardware components are not included. As mentioned in the work of Zaman, et al. [21], a diagnosis board would offer the possibility to measure the voltage and also the current of several devices like laser scanner and motor drivers. With such a diagnosis board, observers would be able to detect a faulty device and the the diagnosis engine would be able to make a more precise estimation of faulty components of the robot. Another feature of the diagnosis board would be possibility to switch devices on and off. One the one hand the repair engine would be able to restart the device, e.g. if the laser is in a fault state and need to be disconnected from and reconnected to the power source. On the other hand, it would be possible to switch off unused devices.

In some situations, this would save some energy, which can be an important point for mobile robots that are sourced by batteries.

The implementation, which was done for this thesis, includes only the parts of the mid-layer and low-level. As further parts, the high-level components **goal planner**, **plan executer**, and **integrity goal generator** need to be implemented. The **task executer** uses simple TCP connections with simple structured data (Protocol Buffer) these high-level components can be ROS independent. Furthermore the **task executer** can be extended to support non-ROS related software modules. Due to the translator plug-ins used by the **task executer**, the realization of this extension should be simple.

Bibliography

[1] Reid Simmons and David Apfelbaum. A task description language for robot control. In *Intelligent Robots and Systems, 1998. Proceedings., 1998 IEEE/RSJ International Conference on*, volume 3, pages 1931–1937. IEEE, 1998.

[2] Stefan Loigge, Clemens Mühlbacher, and Gerald Steinbauer. Supervision of hardware, software and behavior of autonomous industrial transport robots. In *IEEE QRE Workshop on Verification and Validation of Adaptive Systems (VVASS 2016)*. IEEE, 2016.

[3] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated planning: theory & practice*. Elsevier, 2004.

[4] Nils J. Nilsson. Teleo-reactive programs for agent control. *JAIR*, 1:139–158, 1994.

[5] Rachid Alami, Raja Chatila, Sara Fleury, Malik Ghallab, and Félix Ingrand. An architecture for autonomy. *The International Journal of Robotics Research*, 17(4):315–337, 1998.

[6] Sara Fleury, Matthieu Herrb, and Raja Chatila. G en om: A tool for the specification and the implementation of operating modules in a distributed robot architecture. In *Intelligent Robots and Systems, 1997. IROS'97., Proceedings of the 1997 IEEE/RSJ International Conference on*, volume 2, pages 842–849. IEEE, 1997.

[7] François Félix Ingrand, Raja Chatila, Rachid Alami, and Frédéric Robert. Prs: A high level supervision and control

language for autonomous mobile robots. In *Robotics and Automation, 1996. Proceedings., 1996 IEEE International Conference on*, volume 1, pages 43–49. IEEE, 1996.

[8] Solange Lemai and Félix Ingrand. Interleaving temporal planning and execution in robotics domains. In *AAAI*, volume 4, pages 617–622, 2004.

[9] Didier Crestani, Karen Godary-Dejean, and Lionel Lapierre. Enhancing fault tolerance of autonomous mobile robots. *Robotics and Autonomous Systems*, 68:140–155, 2015.

[10] BSI BS5760. Reliability of systems, equipment and components, part 5. guide to failure modes, effects and criticality analysis (fmea and fmeca). *British Standards Institute*, 1991.

[11] Michael Beetz, Lorenz Mösenlechner, and Moritz Tenorth. Cram—a cognitive robot abstract machine for everyday manipulation in human environments. In *Intelligent Robots and Systems (IROS), 2010 IEEE/RSJ International Conference on*, pages 1012–1017. IEEE, 2010.

[12] Moritz Tenorth and Michael Beetz. Knowrob: A knowledge processing infrastructure for cognition-enabled robots. *The International Journal of Robotics Research*, 32(5):566–590, 2013.

[13] Moritz Tenorth, Dominik Jain, and Michael Beetz. Knowledge processing for cognitive robots. *KI-Künstliche Intelligenz*, 24(3):233–240, 2010.

[14] Moritz Tenorth, Daniel Nyga, and Michael Beetz. Understanding and executing instructions for everyday manipulation tasks from the world wide web. In *Robotics and Automation (ICRA), 2010 IEEE International Conference on*, pages 1486–1491. IEEE, 2010.

[15] Michael Beetz, Moritz Tenorth, Dominik Jain, and Jan Bandouch. Towards automated models of activities of daily life. *Technology and disability*, 22(1, 2):27–40, 2010.

[16] L Riazuelo, M Tenorth, DD Marco, M Salas, L Mosenlechner, L Kunze, M Beetz, JD Tardos, L Montano, and J Montiel. Roboearth web-enabled and knowledge-based active perception. In *IROS Workshop on AI-based Robotics*, 2013.

[17] Giuseppe De Giacomo, Yves Lespérance, Hector J Levesque, and Sebastian Sardina. Indigolog: A high-level programming language for embedded reasoning agents. In *Multi-Agent Programming:*, pages 31–72. Springer, 2009.

[18] Hector J Levesque, Raymond Reiter, Yves Lesperance, Fangzhen Lin, and Richard B Scherl. Golog: A logic programming language for dynamic domains. *The Journal of Logic Programming*, 31(1):59–83, 1997.

[19] John McCarthy and Patrick J Hayes. Some philosophical problems from the standpoint of artificial intelligence. *Readings in artificial intelligence*, pages 431–450, 1969.

[20] Anand S Rao, Michael P Georgeff, et al. Bdi agents: From theory to practice. In *ICMAS*, volume 95, pages 312–319, 1995.

[21] Safdar Zaman, Gerald Steinbauer, Johannes Maurer, Peter Lepej, and Suzana Uran. An integrated model-based diagnosis and repair architecture for ros-based robot systems. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 482–489. IEEE, 2013.

[22] Raymond Reiter. A theory of diagnosis from first principles. *Artificial intelligence*, 32(1):57–95, 1987.

[23] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5, 2009.

[24] Eitan Marder-Eppstein and Vijay Pradeep. Ros actionlib package documentation (2009). *http://www.ros.org/wiki/actionlib*, 30.

[25] Thomas Quaritsch and Ingo Pill. Pymbd: A library of mbd algorithms and a light-weight evaluation platform. *Proceedings of 2014 international workshop on the principles of diagnosis*, 2014.

[26] Daniel S Weld. Recent advances in ai planning. *AI magazine*, 20(2):93, 1999.

[27] Avrim L Blum and Merrick L Furst. Fast planning through planning graph analysis. *Artificial intelligence*, 90(1):281–300, 1997.

[28] Rina Dechter. *Constraint processing*. Morgan Kaufmann, 2003.

[29] Stevan Harnad. The symbol grounding problem. *Physica D: Nonlinear Phenomena*, 42(1-3):335–346, 1990.

[30] Oren Ben-Kiki, Clark Evans, and Brian Ingerson. Yaml ain't markup language (yaml^{TM}) version 1.1. *yaml. org, Tech. Rep*, 2005.

[31] William H Press, Saul A Teukolsky, William T Vetterling, and Brian P Flannery. *Numerical recipes in C*, volume 2. Cambridge university press Cambridge, 1996.