Intelligent Model-Based Diagnosis and Repair for ROS-Based Autonomous Robots

Safdar Zaman

Doctoral (Ph.D) Dissertation

Supervisor : Univ.-Prof. Dipl.-Ing. Dr. techn. Wolfgang SLANY
Co-Supervisor : Dipl.-Ing. Dr. techn. Gerald STEINBAUER
External Examiner: Univ. Prof. Dipl.-Ing. Dr. techn. Gerhard FRIEDRICH

Autonomous Intelligent Systems (AIS) Lab.
Institute for Software Technology
Graz University of Technology
Graz, April, 2014.

Declaration

I declare that I have authored this dissertation independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources. Moreover, I happily make this dissertation publicly open so that anyone can use its contents for study, research, and development for the betterment of human future.

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 entnommene Stellen als solche kenntlich gemacht habe. Weiters mache ich diese Arbeit öffentlich, sodass jeder den Inhalt fr Lehre, Forschung und Weiterentwichlung zum Besseren der menschlichen Zukunft nutzen kann.

Safdar Zaman
Graz, 24.4.2014

Dedication

To all the poor people, widows, orphans, and the helpless people of this world.
For the past, present, and future.

Abstract

Autonomous robots comprise a significant number of hardware (e.g., actuators and sensors) and software components (e.g., drivers and processing entities) that are quite heterogeneous in their structure and functionality. Moreover, these components closely interact with dynamic real-world environments. Hence, because of various problems like wear, damage, design, software crashes, and implementation flaws or shortage of testing the involved modules are always subject to faults, unexpected behaviour, and non-deterministic interaction with the environment. This dissertation presents a novel intelligent model-based diagnosis and repair system for autonomous mobile robots running under the ROS (Robot Operating System) framework. The system's architecture has been developed to detect and repair both software and hardware faults in robotic systems at run-time.

The proposed system offers four major contributions. First it offers a model-based diagnosis architecture for a robotics system in order to detect the root cause of the fault both in software and hardware components. Secondly, it provides a planner-based repair engine to repair the faults at run-time without any external intervention. Thirdly, the architecture provides integration into the popular ROS framework. Finally, the proposed work provides an online learning mechanism in order to capture the correct behaviour of the robotics system automatically.

The proposed architecture utilizes five modules; an observer-based monitoring system, a diagnosis model server, a model-based diagnosis engine, a planner-based repair engine, and a hardware diagnosis and repair board. The monitoring system comprises a number of observers in order to monitor the different properties of the robotics system's components at run-time. The diagnosis model server provides a logic-based model of the system description of the robotics system for fault detection and localization. The diagnosis engine performs the task of deriving the root cause of the fault that has occurred. Finally, the repair engine generates a plan of the repair actions and executes them in order to repair the faults. All the modules are based on the Robot Operating System (ROS).

The proposed framework is flexible and generic such that it can be applicable to any ROS-based system. An evaluation of the architecture in real-world experiments showed that the architecture is able to detect and repair a number of software and hardware faults automatically. The shortcomings of the proposed framework and future research challenges also discussed at

the end of the dissertation.

Abstrakt

Autonome Roboter bestehen aus eine Vielzahl an Hardware (z.B. Aktuatoren und Sensoren) und Software Komponenten (z.B. Treiber und Berechnungssoftware) mit unterschiedlichster Struktur und Funktionalität. Zudem interagieren diese Komponenten stark mit ihrer dynamischen Umgebung. Verschiedenste Probleme wie Verschleiss, Schäden, Design, Softwareabstürze, Implementierungsfehler oder mangelhafte Tests, könenn bei diesen komponenten zu Fehlen, unerwartetes Verhalten und nicht vorhersehbarer Interaktion mit der Umgebung führen. Diese Dissertation präsentiert ein neues intelligentes und modellbasiertes Diagnose-und-Reparatur System für autonome mobile Roboter, die mit dem Robot Operating System (ROS) laufen. Die System Architektur wurde dazu entwickelt sowohl Software als auch Hardware-Probleme während des Betriebs zu detektieren und reparieren.

Das entwickelte System bietet vier wesentliche Errungenschaften. Erstens bietet das Systeme eine modellbasierte Diagnose-Architektur für Roboter Systeme, um die ursachen von Fehlern in Software und Hardware-Komponenten zu identifizieren. Zweitens enthält das system eine Planer-basierte Reparatureinheit, um Fehler ohne externe intervention zur Laufzeit zu reparieren. Drittens, ist das System in das populäre Roboter Framework ROS integriert. Viertens bietet das System Online Lern-Mechanismen, um das korrekte Verhalten eines Systems automatisch zu erlernen.

Die vorgeschlagene Architektur besteht aus fünf Modulen: einem Beobachter-basiertem Monitorringsystem, einem Server für das Diagnosemodell, einer Modell-basierter Diagnose einheit, einer Planer-basierte Reparatureinheit und einer Hardware Diagnose Einheit. Das Monitorringsystem besteht aus verschiedenen Beobachtern, die zur Laufzeit unterschiedliche Eigenschaften der Komponenten des Robotersystems überwachen. Der Diagnosemodell Server stellt ein logikbasiertes Modell des Roboters fr die Fehler lokalisierung zur Verfügung. Die Diagnose einheit ermittelt die Hauptursache des detektierten Fehlers. Nach dem Ermitteln des Fehlers, erstellt die Reparatureinheit einen Plan, um diesen zu reparieren. Alle Komponenten laufen mit dem Robot Operating System (ROS).

Das vorgestellte Framework ist flexibel und generisch und kann auf jedem ROS basiertem Roboter System eingesetzt werden. Eine Experimentelle Evaluierung auf realen Roboter systemen hat gezeigt, dass die Architektur verschiedenste Software und Hardware Fehler detek-

tieren und automatisch reparieren kann. Die Einschränkungen des gezeigten Frameworks und zukünftige Verbesserungsmöglichkeiten werden am Ende dieser Arbeit Diskutiert.

Acknowledgement

Praise be to Almighty Allah, the most Beneficent, the most Merciful, who helped me getting this great achievement of my life.

Getting Ph.D without any supervision, support, help, co-operation, and encouragement is although not impossible but is extremely difficult.

First of all I would like to acknowledge and thank the Government of my beloved country Pakistan for supporting me financially during my Ph.D studies through HEC (Higher Education Commission). I also thank ÖAD-Graz for its co-operation during the whole tenure of my studies.

My very special thanks go to my supervisor Professor Dr. Wolfgang Slany for his kind and scientific supervision. He always encouraged and motivated me during my studies. Furthermore, he always open heartedly supported me by providing funds in order to attend research conferences and events which not only helped me scientifically but also broadened my exposure internationally.

I like to greatly thank my co-supervisor Dr. Gerald Steinbauer without whose technical and scientific oriented guidance this milestone would have been extremely difficult for me in the field of autonomous mobile robots. I am specially thankful to him for helping me in building scientific mind and guiding me in every circle of my research work from creating ideas, developing software, preparing experiments to writing research papers.

I also like to thank my external supervisor Professor Dr. Gerhard Friedrich from University of Klagenfurt, for his kind feedback to my Ph.D dissertation.

Furthermore, I like to thank all the teachers, researchers, professors, the deans, the rectors, the administrative staff, the people from the workshops and all other employees of the Graz University of Technology which supported me during my studies.

My love and special thanks go to my parents, my brothers and sisters who always prayed for my success and encouraged me during the endless and tough days of my stay out of my home.

Last but not the least I am grateful to Petra Pichler and Christina Duess for their administrative support at Institute for Software Technology and at ÖAD office Graz respectively.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

This opening chapter of the dissertation presents a brief introduction of what a robot is, why it has become so important to human life, and what it is made up of. Moreover, the chapter also discusses the problem statement and the contribution of the presented work.

1.1 Overview and Motivation

The word "Robot" in a common mind means something that imitates a human being, something that is "smart", "autonomous", "powerful", and/or "fascinating". This word is originated from the Czech word "robota" meaning "work" and "forced labor" [FGL87]. It was first used by a Czeck novelist and journalist called Karel Čapek [1]. He introduced it in his story called "Opilek" published in 1917, later based on this story he wrote a famous play named "Rossum's Universal Robots" in 1920. However, nowadays the "robot" is much more than just a word, it is now an autonomous machine with specific design [Tak07, BGI$^+$09] that acts like an intelligent human being acts. There are many definitions, however here robot can be (probably) defined as:

Definition 1.1 (**robot**). *A robot is a mechanical machine that may resemble a human being, and can autonomously perform the task assigned to it.*

Autonomously performing robot inaction might be dangerous to its environment. Isaac Asimov introduced *robot laws* called "Three Laws of Robotics" [Asi42] as; *Law1:* A robot must not injure a human being nor allow a human being to harm himself. *Law2:* A robot must follow instructions given by human beings without violating the first law. *Law3:* A robot must protect itself without conflicting with the first two laws. During the Middle ages, the robots were primarily used for entertainment purposes. However, the development of the industrial robots got focus in the 20th century. Robot manipulators took over industrial jobs and enhanced productivity by

[1] http://en.wikipedia.org/wiki/Karel_Čapek

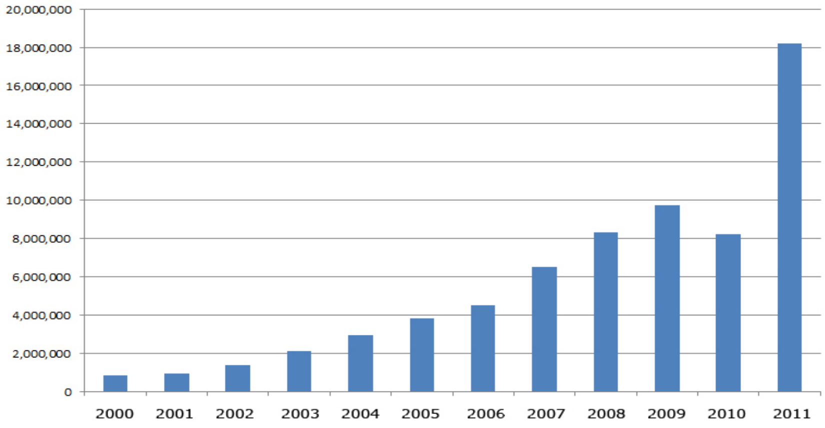


Figure 1.1: Increasing growth of Robots in last decade by 2011 [Pop12].

several times resulting increase in robots demand to get industrial processes partially or fully automatized, e.g., IBM keyboard factory in Texas. As a matter of fact the robots manufacturing companies started establishing and booming the sales of the robots around the world. For example, *KUKA*[2] - a worldwide industrial robot manufacturer. This growing need of industrial robots in past couple of decades exponentially increased robot population around the world. According to International Federation of Robotics (IRF), the world robot population is continuously increasing over the last decade by 2011 which can be clearly seen in Figure 1.1 [Dru12]. If the growth continues with the same pace, its most likely that within next 20 years robot population will exceed (Figure 1.2) the human population on earth[3].

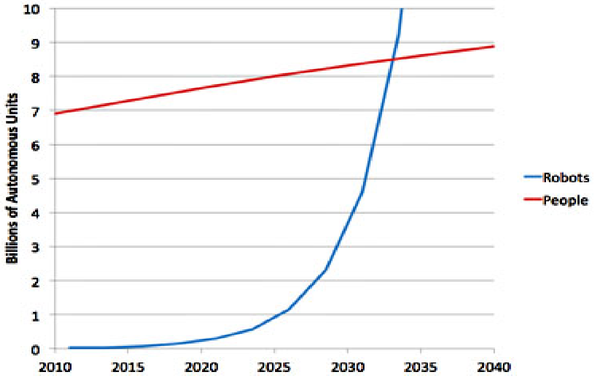


Figure 1.2: Chart showing rapidly increasing population of robots than people [Dru12].

[2] http://www.kuka-robotics.com/

[3] http://earlywarn.blogspot.co.at/2012/04/global-robot-population.html

The extension of robotics application's domain from industries to human environment is rapidly growing due to the desire of automatizing daily life tasks. Next section briefly discusses a number of domains where robots have started performing. Introduction of robots with human related environment have posed a number of questions like how safe and dependable a robotics system should be for a human environment. Safety and dependability (Sec. 1.1.3 discusses in detail) have to be the foremost and the highest priority as human co-existing in an autonomous robot operational environment can be full of risk for unavoidable danger or injury. Robots at industries with current mechanical structure raise crucial physical issues, because unexpected operation of a robot and/or unintended behavior of human can severely harm only the human being. During interaction between a human and a robot manipulator, different kind of injuries can potentially occur, e.g., cuts due to sharp edges, bone fracture or even death due to the manipulator direct crush loads [UAW04, Ogo09]. However, in order to minimize the risk of danger and the severity of injury a safeguarding system for the robots is of utmost necessity. The standards like ANSI/RIA R15.06-1999 and ANSI/UL 1740 (American National Standard for Industrial Robots and Robot Systems - Safety Requirements) are for safety in the factories where robots are employed, address safeguarding requirements for personnel safety. However, these standards are not very useful when a domain is too unstructured because these standards do not specify when people can share workspace with robots [AASB$^+$06].

Although one of the most revolutionary features of robots is going to be the physical human-robot interaction (pHRI), complete safety and dependability issues remain unsolved problems. The dependability of a robotics system is the ability to avoid service failures that are more frequent and severe than those acceptable to the users. This ability encompasses features like reliability, availability, safety, confidentiality, integrity, and maintainability [AA04]. In order to attain dependability features, necessary means of fault prevention, fault tolerance, fault removal, and fault forecasting have to be developed [CGD12]. The means of the fault prevention and the fault tolerance provide trust on the robot service delivery while fault removal and forecasting give confidence in the ability to deliver service successfully.

The motivation for the work presented in this thesis origins from two fundamental observations. First, autonomous robots are artifacts that comprise a number of hardware and software components that are quite heterogeneous in their structure and functionality. Moreover, these components closely interact with highly uncertain and dynamic real-world environments. Therefore, because of various problems like wear, damage, design and implementation flaws or shortage of testing, the involved modules are always subject to faults and unexpected behavior. Moreover, the potentially non- deterministic nature of the interaction of robot with its environment leads to additional faults and unexpected situations. Obviously, such problems negatively affect the performance and the autonomy of the robot. Therefore, a truly autonomous and dependable robotics system has to have the capability to actively cope with such phenomena in an automated way.

1.1.1 Autonomous Robots

Robotics is one of the most interesting and modern fields of study and research. It talks about how to make machines behave intelligently more or less the same way human beings do, and calls such machines the intelligent robots [SNS04]. Recalling the definition from the previous section, a robot does not necessarily have to be anthropomorphic or even animal-like as it is merely a mechanical device to get its jobs autonomously done. Figure 1.3[4] shows autonomous robots with different shapes in their operational fields covering almost every sphere of human life from cleaning to life-saving services. As a source of entertainment, the robots are playing their roles like dancers [Fuj11], lifelike friendly toys for instance iCat[5] and AIBO [Fuj00], and as elderly care robots [HKEW10]. In the medical sector, robots are being vastly used for rehabilitation, conventional, computer-integrated, and minimally invasive surgeries [DGA94, TS03]. Besides that, the robots are also autonomously performing heavily repetitive, monotonous, and dangerous jobs tirelessly 7 days a week and 24 hours a day. Industries are now being equipped with fully automatic robotic manipulators to get works done in hours instead of months. Extra

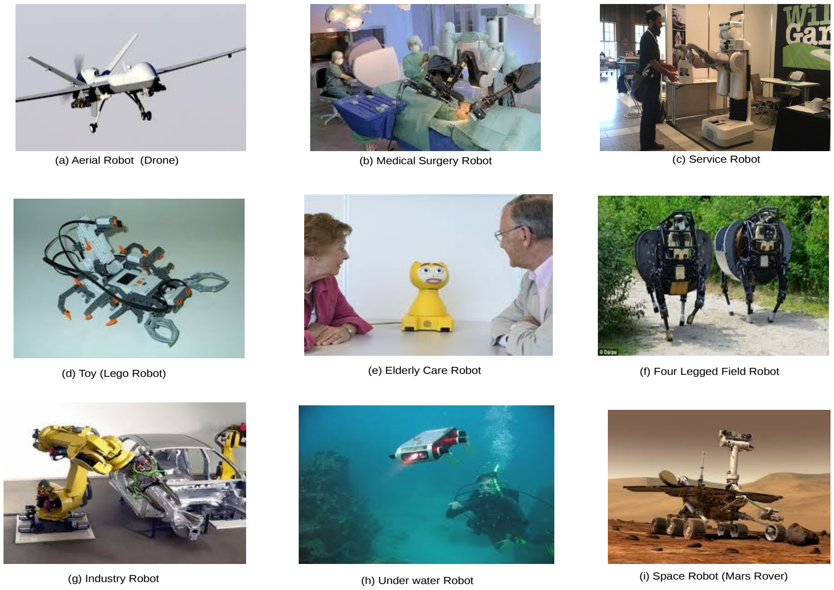

(a) Aerial Robot (Drone)
(b) Medical Surgery Robot
(c) Service Robot
(d) Toy (Lego Robot)
(e) Elderly Care Robot
(f) Four Legged Field Robot
(g) Industry Robot
(h) Under water Robot
(i) Space Robot (Mars Rover)

Figure 1.3: Robots performing different tasks in different fields.

[4] (a) MQ-9 Reaper UAV, USAF Photographic Archives, (b) Da Vinci robot lawsuit, Flagstaff Medical Center, (c) PR2 robot, Willo Garage (d) MrRobot LEGO Mindstorms, (e) UVA-Universiteit Van Amsterdam, (f) LS3, Boston Dynamics, (g) Emerald Insight, (h) AQUA project, Mcqill university Canada, (i) NASA, public domain.

[5] http://www.hitech-projects.com/icat/

shifts have been added to factories because robots never get tired unlike human beings. Robots make us learn about the places that are too dangerous for us to go, e.g., underwater in ocean. The robot *REMUS100-AUV* helps study temperature and other water properties at the depth of 10-$120m$ beneath ice in the area of 10-$20km$ between Pacific and Arctic Ocean [PKSF12]. Robots in space are also helpful, e.g., satellite robot ETS-VII [Mit99] orbiting at altitude of $550km$ with $35\,^{\circ}$ inclination for monitoring international space station and inspecting other orbiting satellites. Exploring dusty lunar surface by rover Apollo-LRV robot [HABJ08], and around 173kg Mars Exploration Rover (MER) to explore $600m$ distance on Mars in 90days in order to acquire knowledge about life, ancient water and climate on this red planet. Moreover, sensing the sun has also been one of the interesting objectives on Mars [ELP02, ISH^{+}03]. Advanced robots, e.g., four legged LS3 robots shown in Figure 1.3, have been built with biology inspired features, e.g., mobility. Hence, moral of the story is that existence of robots has now become a necessity of human existence.

An autonomous robot may or may not be intelligent. For example, an industrial robotic manipulator for assembly line is an autonomous robot because it performs its task without any external help. However, it is not an intelligent robot as it always makes the same moves after a certain period of time even if no part appears on the line in front of it. In order to make it an intelligent robot it has to be provided with some capability of sensing (sensors) its assembly line, and enabling it to make the decision (programmed controller) whether or not to perform the activity.

Definition 1.2 (intelligent robot). *An autonomous intelligent robot is capable of sensing its world, planning future steps using sensed data and its world's model, and executing the plan.*

An automatically navigating robot that can localize itself in its environment, avoid the obstacles (objects and humans), and find alternative paths (if currently followed path is blocked) to its destination, is an autonomous intelligent mobile robot. An intelligent robot perceives its world using its sensors, makes the plan using its internal model, and then executes that plan for the next step. It can have other sensing capabilities like object recognition, speech recognition, face recognition, touch sensing, position and orientation sensing, etc. For instance, *PR2* robot platform from Willowgarage[6] is an intelligent autonomous service robot.

Future intelligent humanoid robotic systems taking part in everyday life are the ones that are supplied with an adequate artificial intelligence and cognition. Cognitive robotics deals with such artificial intelligent functions that include perception, recognition, storing, memorizing, thinking, problem solving, etc. It strives to supply the robots with the ability to learn previously unknown tasks, new motions, new concepts and new objects, and to remember, reason,

[6] https://www.willowgarage.com

and communicate with humans and with each other. Moreover, it uses machine learning Artificial Intelligence (AI) techniques inspired by animal biological systems, e.g., human biological nervous system. These AI techniques are to implement cognitive science theories or models in a robotics system to make them "really" intelligent. Cognition in robotics is mainly required when robot's world is completely uncontrolled and task complexity is large, e.g., nuclear plant operations, space robotics, or unmanned autonomous vehicles. Cognitive capability of grasping the structure of an environment is named as Perception. This involves hardware sensors (e.g., cameras, laser, GPS) to acquire raw data. Learning is another cognitive capability through which a robotics system adapts itself to a new state of the environment by learning from changes. Planning is the process of computing the most appropriate next steps using its knowledge (model) and perceived data. Infering some conclusions from available information in the knowledge, is known as Reasoning. Figure 1.4 shows two cognitive humanoid robots *ARMAR-II* [DBS04] and *iCube* [SMV07]. The robot *iCube* is an openly-available robotics system. It is as tall as a three-year old child. It can crawl, sit, pick up things, move its head, and has fully articulated eyes so that it can show emotions. Its cognitive architecture comprises three layers namely Phylogenetic Sensorimotor, Ontogenetic Action, and Prospective Action Primitives. It enables the robot with the visual, vestibular, auditory, and haptic sensory capabilities. It has as many as 53 degree-of-freedom (DOF) in its whole body which makes it flexible and smoothly moveable. Its cognitive architecture is based on the analysis of the phylogeny and ontogeny of human neonates and hybrid cognitive architectures. An analogous cognitive robot *ARMAR-II* possesses a spine type central body, an active sensor head, 5-fingered dexterous two hands, and 7 DOF each of two arms. Its skills include multimodal man-machine interfaces, augmented reality for modeling, and simulation of robots, environment and user, and cognitive abilities. Its cognitive architecture includes functions for perception, attention control, communication elements, dialogue, management, memorizing, learning, complex task planning and motor control [BMS^{+}05].

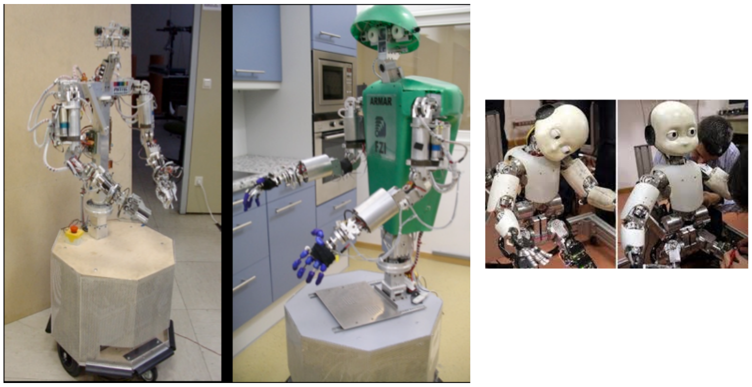


Figure 1.4: Cognitive Humanoid Robots ARMAR II (left [BMS^{+}05]) and iCube (right [SMV07])

Autonomous robots are artifacts that comprise hardware and software components that are quite heterogeneous in their structure and functionality. They are composed of movable physical structures, a power system (electrical/hydraulic/pneumatic), a sensor system, and a software control system. For example, a robot may have a robotic arm called end-effector in order to grasp objects. The sensor system enables robots to receive information about what is happening in their surroundings. Most importantly robots have a reprogrammable "brain" that guides them on each step they take. Designing an autonomous robotics system is a complex and a challenging task [BGI$^+$09] because of its heterogeneous components, complex communicational control software system, and its interaction with highly dynamic, unstructured and uncertain environment. In order to achieve its mission intelligently and autonomously, the robot uses its hardware components (e.g., sensors, actuators) to perceive the environment, and software (e.g., navigation, planning modules) components to make plans for further actions. The hardware components include motors, sensors, actuators, for example, some robot hardware components and sensors are shown in Figure 1.5. The robot perceives its environment with the help of its hardware components by interacting with the environment. Trackers, flippers, wheels, joints, motors, and actuators give the robot a capability to mobilize itself in its world. The sensors like Sonar, Ultrasonic, Infrared (IR), and Laser sensors measure distance of objects in front of the robot, Thermal cameras measure heat energy, Tactile sensors measure pressure force and so on. The software components include operating system for robot, all hardware drivers (e.g., Kinect camera driver), and software entities (e.g., navigation stack) that are altogether involved in completing a mission. A robotics system contains a special mechanism called *control paradigm* that uses hardware and software components in order to make it possible for the robot to achieve its task.

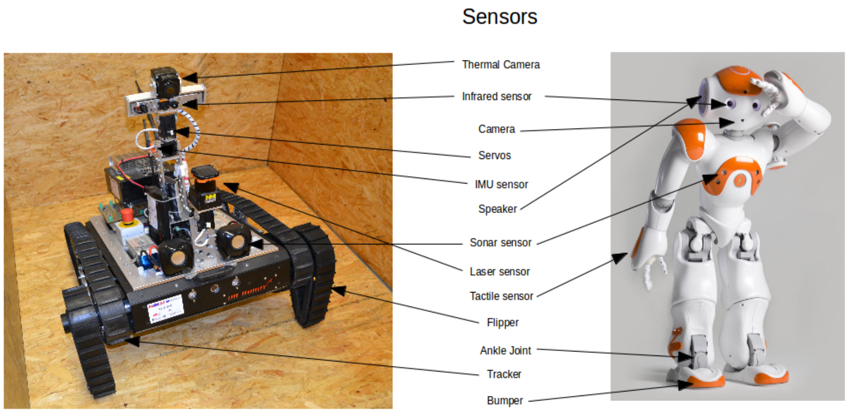


Figure 1.5: Robot's sensors; TEDUSAR search and rescue robot (left) and Nao robot (right)

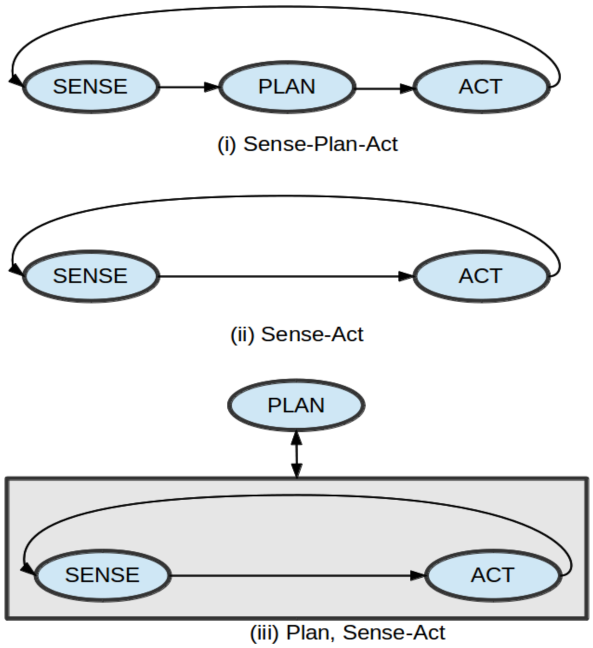


Figure 1.6: Robot control paradigms [Mur00]: Deliberative (i), Reactive (ii), and Hybrid (iii).

1.1.2 Robot Control Paradigm

No matter how robots look like and what they are built for, they all basically carry out three common activities called the primitives [Mur00], namely *Sense*, *Plan*, and *Act*. **Sense:** Every robotics system has to have some way of sensing the world around it. To acquire this ability, a robot possesses number of sensors. The information from sensors lets robot know how far it has moved, and if there are other moving objects in its world, how does an object look like, how far an obstacle is in front of it, etc. **Plan:** This is an activity where intelligence jumps in. The *robot control system* smartly prepares instruction(s) called *plan* by using sensed information in combination with the *belief* (internal representation or model of its world) a robot already has. A plan may contain instructions like stop, speak, turn 180° , move, etc. **Act:** This activity carries out physical movement in actuators as per instructions from the plan generated by previous activity. Examples may include; rotating wheels with certain velocity, opening gripper for grasping, getting robotic arm up, etc.

Every robotics system in action continuously performs cycle of these three basic primitives. On the basis of these primitives there are three control paradigms namely *Deliberative, Reactive,* and *Hybrid* as depicted in Figure 1.6.

- **Deliberative:** Under this paradigm, a robot first senses the world around it, then makes a plan for the next action, and finally acts by executing the planned actions. This paradigm is also called *sense-plan-act* paradigm. For example, a robot takes laser data telling that a door is opened, then it plans to move (not to turn back) one meter straight as the door is sensed opened, and finally action transfers to the actuators to move straight to reach the goal.

- **Reactive:** This paradigm does not use planning. The robot senses the world and then acts by calculating the best action through *sense-act* coupling. This paradigm uses multiple *sense-act* behaviors. For example one *sense-act* behavior directs the robot to move straight by sensing the goal, and another concurrent *sense-act* behavior senses a person appeared in front of it, and steers the motors to make robot turn or stop.

- **Hybrid:** It combines the *hierarchical* and the *reactive* paradigms making pair (*plan, sense-act*). The planner first decomposes a task into subtasks then executes them using *Reactive* paradigm. When these subtasks are completed the planner generates a new set of subtasks. For example, the planner decomposes the task of constructing map and instantiates different *sense-act* behaviors for auto exploration, obstacle avoidance, and map building.

1.1.3 Robotic System Dependability

Autonomous robots have gained place almost in all domains of human life, some of the domains are significantly critical, e.g., human surgeries, space exploration, or nuclear domain. Moreover, it is an acknowledged fact that threats to a system can never be 100 percent removed, and that they may occur at any time. As a matter of fact, a robotics systems failure in a critical field such as above, will of course lead to catastrophes. Therefore, due to advanced decisional capabilities of autonomous robots in highly uncertain and critical environments, these systems raise concerns regarding their dependability. The authors of the contribution [AA04] define dependability both from qualitative and quantitative perspectives as:

Definition 1.3. *(***dependability***) Qualitatively the $dependability$ is a justifiable trust in a robotics system to perform its task correctly. Quantitatively it is the ability of robotics system to avoid service failures that are more severe and frequent than are acceptable to the users.*

A service is said to be successfully and correctly delivered by a robotics system, when during the service delivery it functions almost the same way as it was intended to. The more dependable a robotics system is, the more trustworthy it becomes. Dependability offers following attributes:

1. **Availability :** It defines the capability of readiness for delivering a service correctly.

2. **Reliability :** It is continuity of correct service deliveries in the presence of threats.

3. **Safety :** It ensures avoidance of catastrophic consequences to the environment.

4. **Confidentiality :** It ensures absence of disclosure of information to unauthorized users.

5. **Integrity :** It provides absence of improper system functionality alterations.

6. **Maintainability :** It enables the system for repair and modification process.

The threats to the dependability are the causes that can prevent a robotics system to deliver its service correctly. These threats include failures, errors, and faults as briefly explained below:

- **Failure** is an event which occurs when a robotics system delivers a service that deviates from correct service. The service failure can be the cause of poor implementation of service function. A failure can also be regarded as permanent interruption in delivering correct service.

- **Error** is a deviation between computed and true values. In the context of dependability, it is deviation of delivered service from correct service. It is the error which leads to the service failure.

- **Fault** is in fact a violation of a standard condition. The hypothesized cause of an error is a fault.

There are two basic measures namely *Fault avoidance* and *Fault acceptance* that can help system develop as a dependable system [LCI+04], [LLC+05].
Fault Avoidance : This concept completely prohibits a system to get faults by any means. In order to nearly develop such dependable system, there are costs to pay, e.g., developing a strong monitoring system. Fault avoidance is further divided into two sub groups:

- Fault Prevention : This is how to prevent the occurrence or introduction of the faults to a system.

- Fault Removal: This defines how to remove the faults in order to reduce the number or severity of faults on a system.

Fault Acceptance : It allows a system to face fault when it occurs. The system then estimates and learns in order to reduce future fault occurrences and their consequences. Like fault avoidance, it can also be divided into two sub categories:

- Fault Tolerance : This deals with how to keep delivering correct services in the presence of faults in a system.

- Fault Forecasting : This is how to estimate the present number, the future incidence, and the likely consequences of faults in a system.

These four general attempts namely *Fault Prevention*, *Fault Removal*, *Fault Tolerance*, and *Fault Forecasting* are altogether considered as $means$ in order to attain a system's $dependability$. Autonomous robots actively interact with real-dynamic and unstructured world where any unforeseen event is always likely to happen, e.g., a route is completely blocked because of people, or the robot slipped because of some slippery material on the surface. During mission completion robotic system may encounter such unforeseen events; therefore a dependable robotics system should deal with them properly. A robotics system can be made more dependable by resolving concerns raised by introducing appropriate fault tolerance mechanisms.

1.2 Problem Statement

As autonomous robots are gaining more attention in almost every domain, and robotic applications for non-trivial tasks in everyday life environments are significantly increasing. This comes up with the need of a robust and dependable robotics system. Robustness is the delivery of a correct service in adverse situations due to uncertainties [LLC$^+$05] in real world scenarios. Therefore, in order to make robotics systems more dependable and robust for various tasks in different environments, a number of scientific questions are raised. This dissertation is focused on fault detection and repair in autonomous robots in order to resolve the following issues:

Firstly, a robotics system comprises a significant number of hardware and software components that are quite different with respect to their structure and functionality. Moreover, these components closely interact with dynamic real-world environments. Hence, because of various problems like wear, damage, design and implementation flaws or shortage of testing the involved hardware and software modules are always subject to unexpected behavior and faults. This fact leads to the need for online fault detection and localization in hardware as well as in software.

Secondly, during accomplishing a task it is quite likely that a component of a robotics system shows an undesired behavior that can be caused by a wide range of faults such as defective hardware or software deadlocks. This phenomenon is caused by the complex interactions within the robotics system and the non-deterministic interaction with the dynamic environment. In order to be able to automatically cope with such problems it is necessary to have a monitoring system that is not only able to detect the faults but is also able to repair the faults at run-time in order to bring the robotics system back to its normal form.

Thirdly, in order to cope with these issues, if a model-based diagnosis approach is used, one needs a model of the correct system behavior. Such a diagnosis model can be acquired using three basic approaches. The first approach is to reuse requirements or engineering models that are already available if for instance a model-driven development process is used [BGVB10]. If no reusable models are available diagnosis models can be created by hand. While this second approach is quite widespread it is cumbersome and error-prune in particular for complex systems.

This requires the need of learning a diagnosis model automatically during a controlled learning phase.

Fourthly, the robot operating system (ROS) [QCG$^+$09] already provides a simple fault diagnosis system[7]. However, it is mainly limited to only monitoring hardware modules and code execution. Moreover, it is essential for an autonomous robot that once a problem has been identified the system is able to derive and execute appropriate repair actions automatically but ROS does not have any such capabilities for autonomous repair. An increasing number of research groups around the world use ROS as a standard framework for the development of robot systems, these facts also demand for a diagnosis and repair system based on ROS platform.

In order to tackle with the above stated issues a required autonomous fault detection and repair system should posses the following features:

- ◇ **Scalability :** It is the well-behavior of the system performance under change in resources [Dav94, Hil90]. The more scalable the system the better it is.
- ◇ **Reusability :** Reusability is one of the key features of a system to keep it alive. The desired system should be easily reused over the time when required. This feature should not only apply to the whole system but its components should also be individually reusable.
- ◇ **Generalizability:** The robots and their control systems are evolving rapidly because of new technologies both in hardware and software directions. Therefore, the system should be completely generic rather than being task specific. It should follow standards to make it more general so that the systems following the same standards, can use it or its components.
- ◇ **Reliability :** The system should keep performing its correct services for a period of time [CGD12, LCI$^+$04], be reliable enough in order to cope with "every" situation in daily life. It will make the robot interact with dynamic world reliably in order to meet its goal.

The purpose of this dissertation is to present a diagnosis and repair system architecture for ROS-based robotics systems, that can offer as many features of above as possible to the scientific community. The contribution behind the presented work is described in the following section:

1.3 Contribution

The presented work contributes to the solution of the challenges stated in Section 1.2. It offers a diagnosis and repair system with the following features:

[7] For further information on the ROS diagnostics stack please refer to http://www.ros.org/wiki/diagnostics.

- **Software and Hardware diagnosis :** The diagnosis system presents a number of observing entities in order to monitor the robotics system's behavior at run-time. In addition to this it also presents a diagnostic board and a hardware observer to cope with detecting and localizing the faults related to the hardware components of a robotics system.

  The basic architecture of our diagnosis and repair system for software and hardware is presented in our contribution [LMS$^+$12]. For the purpose of diagnosing and localizing hardware faults a diagnostic board that follows a standard based on TCP/IP protocol has been manufactured and successfully tested. The particulars of the diagnostic board with evaluation are presented in our contributions [ZL13, ZL14].

- **Repairing diagnosed Faults :** The system is capable of not only diagnosing the faults in software and hardware components but also can repair them. The system can deal with only transient faults not the permanent ones; i.e., it is not possible to automatically repair a permanent fault (e.g., a broken gear) at run-time. It is important to bring robotics system into its normal state again.

  The work presented in our contribution [ZSM$^+$13] presents the architecture of the diagnosis and repair system for both software and hardware faults. The contribution provides a number of monitoring units in order to observe the robotics system for detecting the faults and repairing them.

- **Support for ROS framework:** ROS is becoming more popular and almost every robotics system is now adapting this framework. Therefore, the diagnosis and repair system is based on the ROS framework to be used by all ROS based robotic systems. Moreover, the system is compatible with already existing ROS-based packages, i.e., it does not require the already existing packages to be edited and recompiled for being compatible.

  The utilization of the ROS framework has been presented in our contribution [ZSS11], where a scenario for mapping and automatic robot navigation based on ROS framework is presented.

- **Integration of Existing Diagnostics :** The diagnosis system accommodates already existing ROS diagnostics by providing a monitoring unit for the diagnostics messages. Moreover, the system is also capable of publishing the ROS diagnostics compatible messages.

- **Automatic model generation :** A model of correct behavior of a robotics system is very vital to detect a fault precisely. For a complex robotics system, the generation of the model by hand, can be a very tedious job and subject to errors. Therefore, the diagnosis system is capable of generating a model of correct behavior of a robotics system automatically.

> A preliminary and simple system for the model generation for the robotics system is presented in our contribution [ZS13a]. While a fully comprehensive automatic model generation for the ROS-based robotic systems is presented [ZS13b]. The model generation process generates a logical behavioral model of the robotics system.

The work explains the overview of the architecture, its five important modules namely, (1) *Observers*, (2) *Diagnosis Model Server*, (3) *Diagnosis Engine*, (4) *Action Servers*, and (5) *Diagnosis Repair Engine*. The work also explains integration of the architecture into ROS, and integrates the already existing ROS diagnostic stack. In the work all components of the architecture are discussed and an evaluation of the system is presented showing feasibility of the work.

The work presents a complete methodology from extracting training set data to generating observers and a model for diagnosis and repair system. The system is publicly available online and can be downloaded through git system[8].

1.4 Outline of the Thesis

Chapters of this thesis are organized as follows:
Chapter 2 : Research work related to the presented methodology is given in this chapter. It covers a brief survey on the contributions done in the direction of *diagnosis, planning, repair* and *modeling*.
Chapter 3 : Robot Operating System (ROS) and Model-Based Diagnosis (MBD) System are both basic building components for this dissertation. Important and basic terminologies related to ROS and MBD system are covered in this chapter. Moreover, it also presents basic concepts about the logic and the planning used in our diagnosis and repair system.
Chapter 4 : This chapter presents the overview sketch of the methodology of this work. The basic architecture of presented work is described in this section of the thesis. The architecture includes *observers, a model_server, a diagnosis_engine, action_servers,* and a *repair_engine*.
Chapter 5 : This chapter presents a number of observers that monitor different properties of the system components. The observers provide observed behavior of robotics system at run-time.
Chapter 6 : This chapter is dedicated to diagnosis. How a system is monitored and which kind of observers are used for this purpose is discussed here. Moreover, it also discusses ROS-Based diagnosis engine which derives diagnosis on the basis of the observations.
Chapter 7 : This chapter discusses how *repair_engine* uses diagnosis and observations to make a plan for repair actions. The plan is then executed through *action_servers* in order to repair faulty component(s).
Chapter 8 : The hardware diagnostic board is discussed in this chapter. The diagnostic board is

[8] git@robotics.ist.tugraz.at:tug_ist_model_based_diagnosis.git

built particularly for dealing with hardware related faults. Moreover, this chapter also presents hardware insights, a protocol, and control software of the diagnosis board.
Chapter 9 : Model-based diagnosis depends upon the model of the correct behavior of the system. The way how the model is learned, is presented in this chapter. The model is learnt from a fault free run of a robotics system to collect correct behavior.
Chapter 10 : This chapter is dedicated to the results of the evaluation process and the experiments. It covers the evaluation of diagnosis (detecting and locating faults), repair (recovering faulty components), and also modeling (creating and influence of different models).

Chapter 2

Related Research

This chapter discusses some of already existing research contributions in the area of fault detection and repair. The related research in the direction of diagnosis, planning, repair, and modeling is presented in the relevant sections.

2.1 Diagnosis

The diagnosis is a problem of finding what is wrong with a system, based on (1) knowledge about the structure of the system, (2) possible malfunctions, and (3) observations (symptoms, evidences) coming from the behavior of the system [Poo89]. Fault diagnosis is the field that focuses on detecting faults and localizing them by locating the root causes of these faults. Increasing research in the direction of *Fault Diagnosis* reflects the significance and the need towards this field [LK08, BCNB07, IB97, BBdK82, Sim99]. Furthermore, two well known distinct and parallel communities namely FDI (working in control field) , and DX (working in Artificial Intelligence) [CPVG05] are contributing in fault detection and isolation using model-based reasoning. The development in this field began many decades before in the last century. In 1971s, the work on observer-based fault detection in linear systems was reported [Bea71], and in early 1976 local approaches to fault diagnosis used fault models to localize failure in faulty components [dK76]. According to contributions [VRYK03, VRK03, VRKY03] diagnostic methods can be categorized on the bases of the types of knowledge and diagnostic search strategy (Figure 2.1 from [VRYK03]). The analysis of diagnosis and diagnosability in the context of model-based diagnosis notions by using Performance Evaluation Process Algebra (PEPA) is covered in the contribution [CPR00].

A diagnosis system generally takes as an input a set of symptoms which are measurements or observations acquired from candidate system, and encoded in machine readable format. In order to identify root cause of these symptoms, the diagnosis system has to have necessary knowledge

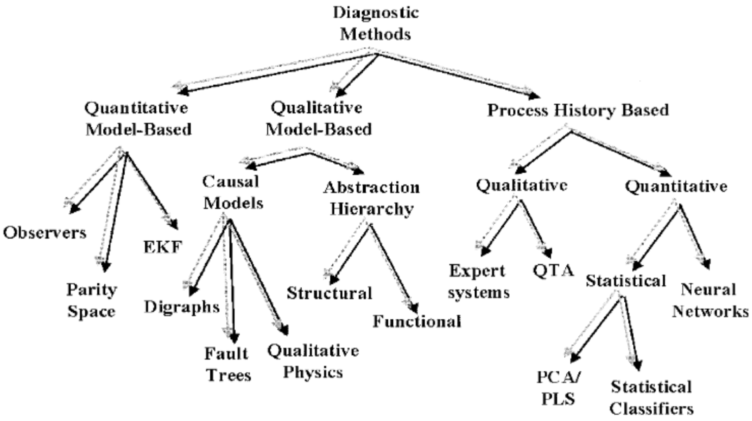


Figure 2.1: Classification of diagnostic algorithms [VRYK03].

of the domain. This necessary domain knowledge is engineered into the system in some form (e.g., rules, semantic networks, frames, logic, etc.), or the system is made capable of acquiring the domain knowledge on its own. The system following the former way of knowledge engineering is called a knowledge-based system [BH98]. Typically, the knowledge is engineered in such systems in the form of rules; this is the reason why such systems are also called rule-based systems. The complete knowledge base of such systems contains rules of *IF-THEN* form. *Expert System* is one of the knowledge-based systems where domain knowledge is maintained in the form of such rules. An example of domain knowledge coded in IF-THEN rules:

RULE1 : IF Headlights do not work
THEN Battery and/or Bulbs are faulty

RULE2 : IF Battery and/or Bulbs are faulty
AND Engine does not start
THEN Battery is faulty

RULE3 : IF faulty Bulbs and/or Battery
AND Engine starts
THEN Bulbs are faulty

The rules described above represent a sub part of domain knowledge from an auto mobile diagnosis system described in the contribution [BH98]. Knowledge-based systems generally use an inference engine to draw inferences by matching known facts with *IF* part of the rules in the domain knowledge. Inference engine achieves this by searching through the domain knowledge. One of the major issues with knowledge-based systems is to acquire domain knowledge form the expert of domain. There are a number of issues with such systems [Web08]: Firstly, the domain knowledge is hard to create by hand. Secondly, the quality of the diagnosis depends on expert's experience of the domain. Thirdly, these systems cannot correctly locate multiple faults because they rely on single fault assumption.

Unlike knowledge-based systems, the model-based diagnosis systems [DH88a] use a generic abstract model of a domain instead of hand-crafting knowledge bases. The model in the model-based diagnosis can be either constraint-based or logic-based [PW03]. The model-based diagnosis uses a logical reasoning by using a model in order to derive the root causes if any deviation in the behavior is observed. It offers two basic approaches namely *abductive* and *consistency-based*. Both of these approaches differ in representing a diagnostic problem. The abductive approach [Poo94, CTng] revolves around the concept of *causes* and *effect*. The possible causes (faults, diseases) parameterized by the values on which they depend, are the possible hypotheses. The axioms are developed on how symptoms follow from the causes. These axioms should be facts if the symptom is always present given the cause, otherwise these should be possible hypotheses. The approach utilizes knowledge about the faults and their symptoms in order to monitor abnormalities. It requires different knowledge than a consistency-based approach in order to get the same diagnoses [Poo88, Poo89, CDTnn]. The consistency-based approach [dKW87, Gen84] uses correct behavior of a system without having knowledge of abnormalities or faults. On deviation, it isolates abnormal system components form the normal ones using logical reasoning.

The presented work follows the concept of consistency-based diagnosis system first proposed in 1987 by Reiter's contribution in this field [Rei87]. It is a widely used approach to the model-based diagnosis within the community of artificial intelligence. This is a process of an iterative cycle consisting of steps, namely behavior prediction, conflict detection, candidate generation, and candidate refinement [PG04]. Consistency-based reasoning provides a logical foundation for diagnostic reasoning and clarifies fundamental assumptions, such as single fault and exoneration. It can work with Horn clause logic or predicate logic like programming language (Prolog) [GMC04]. It is a specialized model-based diagnosis approach that uses propositional logic to express and analyze the model of systems. It generates diagnosis which is a set of faulty components in the system. This set explains the observations using the notion of logical consistency [Pal01]. Apparently diverse model-based diagnosis systems have been built for troubleshooting and diagnosis [IB97, Ise97, CP99, GMC04]. This diversity basically lies in the varying kinds of knowledge in each stage of their process [DH88b]. Every model-based system (e.g., Consistency-based) follows the same fundamental paradigm as the interaction of observation (from actual system) and prediction (from system's model) as shown in Figure 2.2. This fundamental paradigm depicts two behaviors, namely $observed$ and $predicted$. It states that for the correct model of a system (the presumption behind every model-based system) all discrepancies are due to the defects coming only from the observed behavior of the system.

Throughout the development of this dissertation, we utilized the experience in the diagnosis

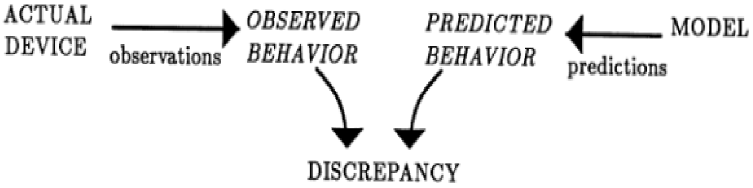


Figure 2.2: Model-Based Diagnosis in terms of predictions and observations [DH88b].

systems and the results from different previous similar works. The authors of the contribution [SMW06] presented a model-based approach for diagnosing faults in a robot control software which uses a model of the communication between components. The approach was based on a CORBA-based communication framework; we developed an architecture under ROS platform which is a latest and widely used robotic software framework nowadays. Moreover, the approach is only meant for the software fault diagnosis. Instead we integrate hardware fault diagnosis as well. The research contribution [KSW09] introduced the concept of utilizing qualitative property (increasing, decreasing, and constant) of data for sensor validation. The approach uses qualitative symbols `[-,0,+]` for the abstraction of sensor data based on qualitative reasoning techniques. In order to reason about the root cause of any unexpected behavior, it uses expected qualitative relations between sensor streams. In our work we exploit the same idea to develop one of the observers for monitoring such qualitative trends in the data. Moreover, our system learns the qualitative relations between sensors data online during the learning phase. In [PW03] the model-based reasoning from first principles in the domain of circuit designs is presented. The approach uses a logical model to represent correct behavior of circuits, and uses observations to reason about deviation using first principles. We use a similar way to this approach for representing logical behavioral model. Moreover, we use a diagnosis engine (open-source) which is a java-based implementation of the approach.

The authors of [MAVL06, MAVL07] presented an FDI approach to detect faults of sensors in a mobile robot. The approach uses a set of constraints about known and unknown values in a system (e.g., one value is proportional to another one) to generate residuals which can be used to detect sensor faults like offsets in accelerometers. Moreover, the system proposes to use filter techniques (Kalman, particles) to cope with noisy observations. The work is very much related to the proposed approach in the sense that faults in components are identified based on observed signals. But while the constraints are manually given in that approach, our method identifies such relations automatically in a learning phase. Moreover, the method presented in this paper uses a qualitative approach to detect deviations rather than a quantitative approach. In [KKR13] an approach for on-line diagnosis of components of autonomous system is presented. The approach compares pairs of sensor signals in order to detect faulty components. The approach is similar to the presented approach but needs to know the structural model in advance. Moreover, the signals are only linearly correlated. In contrast our approach learns the structural model automatically and correlates the signals qualitatively over a period of time. The contributions [REW06, RW05] describe a model-based system that provides the ability to increase the robustness of complex systems. It is based upon model-based programming tools that enable the specification of self-deprecating and self-reconfiguring methods along with model-based executives that reasons from the component service models to continuously monitor, diagnose, reconfigure a function in a complex system. The author of the contribution [Kal12] presents a model-based approach to coordination failures in a multi-agent system due. The agents with their states are represented by a matrix-based notation which defines a coordination design of the multi-agent system. The approach uses a logical multi-agent system description (*MSAD*), and a diagnoser for observing the agents. It reports a coordination failure when an agent's observed coordination mismatches the expected coordination in the matrix.

In the context of hardware diagnosis and repair, the Livingstone architecture proposed by [MNPW98a] was used by NASA's first New Millennium mission named *Deep Space One* (DS1) in order to diagnose the space probe's *Remote Agent* for its failures in hardware and to recover from them. The process of fault identification and reconfiguration uses compositional, declarative, concurrent transitions system model together with probabilistic and deterministic transitions, while the planning and scheduling part is constraint-based, operating on a declarative domain model in order to generate a plan from first principles. A model-based diagnosis approach in order to identify faults in the hardware design descriptions is presented in the research work [FSW99]. The work uses a communication structure from hardware design source program coded in a hardware description language for an hardware (e.g., D75) in order to debug the code for a possible misbehavior of the hardware. We adapt this concept of extracting diagnosis model from communication structure between the components, and comparing observerd and predicted behavior of the components to detect discrepancies and localize the faults using the model-based diagnosis and reasoning. The contribution [CM03] presented an evaluation of failures on as many as 13 different mobile robots. The most common failures encountered were hardware related failures, e.g., effector failures, namely, tracker, gear, motor, and wheel problems. The analysis shows that hardware failures are important to deal with in order to have trustworthy and dependable autonomous robotics system.

The work [BHSW07] presented a mechanism for fault diagnosis and reconfiguration of robot wheel drives at run-time. It provides a control framework which is capable of reconfiguring the control functions of the drive based on the detected faults. The approach reacts to the faults in robot drives in three different ways: (1) "slightly faulty" behavior adaptation if the fault is not serious, (2) system degradation (omni to differential drive) if fault is not adaptable, and (3) safe state or informing human operator if fault cannot be reconfigured at all. Although the approach is a good initiative in the direction of hardware diagnosis and repair, it is specific to particular hardware and reconfiguration. The approach cannot deal with the situation where robot drives need to be switched ON/OFF depending on the behavior of its software driver, which is a more common failure than a broken drive. Our diagnosis system uses a hardware diagnostic board which can automatically control different hardware components attached to it. The authors of [ZKH$^+$01] presented a hybrid diagnosis approach for a printer system (Xerox DC265) where mode estimation for continuous sensor measurements was combined with a decision tree using discrete components modes. The diagnosis approach presented in this paper is related to this work because it uses correlations of signals to generate input for a discrete model-based approach. A different approach using particle filters for estimation the mode of robot hardware was presented in [VVGG$^+$04].

2.2 Planning

In order to enable repair actions we use in our work the widely recognized Planning Domain Definition Language (*PDDL*) [KBC$^+$98] which is one of the extensions to research in the direction of planning, originated with a basic planning language called *STRIPS* (STandford Research Institute Problem Solver) [FN71b]. *STRIPS* is a member of planning problem solvers that searches a space of ”world models” to reach a given goal. It represents a world model by a set of well-formed formulas (wff) of the first-order predicate calculus. Due to some limitations in STRIPS, its first advancement in the form of *ADL* (Action Description Language) was introduced [Ped89]. *ADL* supports many additional features not supported by STRIPS, e.g., negative literals, quantifiers and disjunction in goal state, equality predicate ($x = y$), and support for types, etc. The extension to ADL is PDDL which not only supports STRIPS and ADL but also offers a number of extra features that include specification of hierarchical actions, subactions, subgoals, etc. Since its development, PDDL is gaining popularity and getting extended to integrate new features [FL03]. In our system architecture we use a planner based repair engine. It converts observations and diagnosis into a PDDL planning problem definition, and get a plan for the actions. In order to get a valid plan from PDDL representations of a system, we use a java-based GraphPlan [BF97].

A propositional planning system called *LAMA* is presented by [RWH11]. It uses a heuristic derived from landmarks in conjunction with the well-known FF heuristic. LAMA builds on the Fast Downward Planning System using non-binary state variables and multi-heuristic search. It uses $A*$ search so that the planner continues to search for the plans of better quality until the search is terminated.

One of the most exciting development in AI-Planning is *GRAPHPLAN* [BF97]. The *GRAPHPLAN* is widely used planner because of two reasons: Firstly, it is an elegant and simple algorithm which is an extremely speedy in many cases, especially faster than previous systems like *PRODIGY* [MCK$^+$89], and *SNLP* [MR91]. A general purpose problem solver STRIPS is developed specifically for robot tasks planning problems [FN71a]. It represents a world model in the form of a set of well-formed formulas (*wffs*) for instance, a world model where a robot is at location a and boxes B and C are at locations b and c. STRIPS *wffs* would be: ***ATR(a), AT(B, b), AT(C,c)***. Problem space for STRIPS is defined by (1) an initial world model, which is a set of *wffs* describing present state, (2) a set of operators with description of their effects and preconditions, and (3) a goal condition stated as a *wff*. STRIPS needs model of each action in order to generate a plan. These model actions are *operators* which transform one model into another. Each STRIPS operator is characterized by three entities: a *delete_list*, an *add_list*, and a precondition *wff*. A general STRIPS planning problem comprises of an initial state S, a goal state G, and a set of STRIPS actions. A STRIPS action definition specifies three sets of facts: a set of preconditions facts (*PRE*), a set of add effect facts (*ADD*), and a set of delete effect facts (*DEL*). For instance, a world model with two objects A, and B, a table, and a robotic arm. Initial state $S = \{holding(A), clear(B), onTable(B)\}$ specifying that the only object B is on the table, and the object A is held in robotic hand. The action $putDown(A, B)$ can be specified in terms of ADD, DEL, and PRE lists as $\langle PRE : \{holding(A), clear(B)\}$, $ADD : \{on(A, B), handEmpty, clear(A)\}$, $DEL : \{holding(A), clear(B)\}\rangle$ The fact for goal state can be for example $on(A, B)$. An operator is applicable if its precondition *wff* is satisfied. Add and delete lists describe how an operator transforms from one state into another. Authors in contribution [FHN72] explain STRIPS planning process in detail.

2.3 Repair

Faults in a robotics systems are always unavoidable. If a robotic system in a critical situation faces a failure (e.g., NASA Mars rover gets fault in a motor [WHC$^+$06]) then it needs to be recovered efficiently and automatically. This raises the need of an automatic repair system for the faults in both hardware and software components. Repairing faults in hardware component, e.g., robot drives, is presented in contribution [BHSW07]. It offers a reconfiguration engine (Figure 2.3) along with a diagnosis engine in order to detect, localize, and repair faults in omni-directional drives of a specific robot. It deals with three scenarios: Firstly, if a fault which is not too serious is detected, then after localizing the fault the reconfiguration engine adapts its behavior to the new situation by reconfiguring the robot drive without changing the functionality of the robot. Secondly, if a fault cannot be fully compensated by the reconfiguration engine then it performs a controlled degradation of the functionality. For instance, an omni-directional drive degrades to a differential drive. In this scenario the higher level control system of the robot (planner) is also adapted according to the new functionality. Thirdly, it considers a serious fault where either the robot is switched to a safe state or alarmed for external intervention to recover from the fault. For evaluation it injected online faults simulating broken wheels, gear,

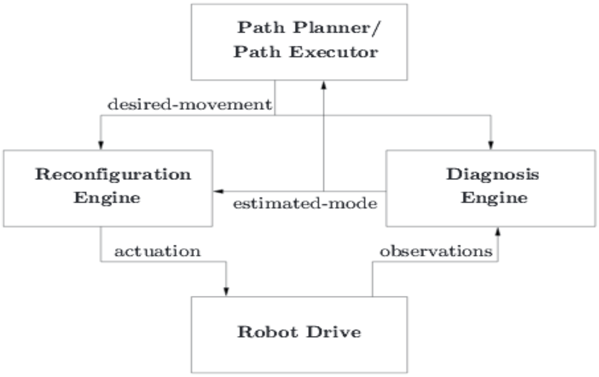


Figure 2.3: Framework with reconfiguration engine for repair [BHSW07].

fault in motor, or in its control electronics. The detailed generalized framework for the same reconfiguration repair system has been described in the work [HKSW07].

Robotic control software consists of different services that communicate with each other and with hardware components. The control software is responsible of overall control of robot in achieving its mission. A malfunctioning service can disturb the system to a complete robot mission. The detection and the localization of the faults in a control software of a robot are presented in [SMW06]. The contribution uses a framework with three modules: (1) a monitoring module, (2) a diagnosing module, (3) and a repair module. The monitoring module observes the behavior of software services, and reports $\neg ok(s)$ if service s deviates from its correct behavior, otherwise $ok(s)$. The diagnosing module locates the faulty software component if a deviation is reported. It uses abstract model of the correct behavior of the system. The repair module consists of two actions namely *stop* and *restart* in order to stop functioning or again starting the faulty service of robot control software [MNPW98a].

The authors of [GWHH10a] and [GWHH10b] presented statistical learning techniques for models of the communication within robot systems. The presented techniques allows for estimating a probability distribution of the internal data exchange and communication in a robot system. This approach is similar to the presented because it uses statistical information about the occurrence of messages. We extend this approach by additionally integrating the content of the messages.

A Sensor Fusion Architecture Exception Handling (SFX-EH) has been presented by [CM03]. The approach recovers autonomous mobile robots from errors. Figure 2.4 depicts the steps involved in SFX-EH approach. It achieves exception handling strategy with two steps *error classification* and *error recovery*. The former step generates all possible hypotheses and tests them in order to classify sensor failures. The underlying cause of the classified failure is passed to the *error recovery* step for the repair. The recovery process uses a lookup table to search for alternatives sensor and recovers from the error. If no alternate is found the approach simply declares

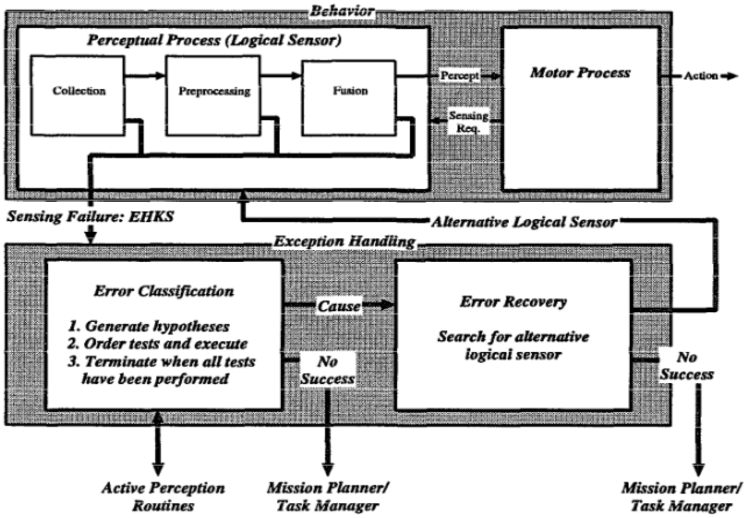

Behavior
Perceptual Process (Logical Sensor)
Collection
Preprocessing
Fusion
Percept
Sensing Req.
Motor Process
Action
Sensing Failure: EHKS
Alternative Logical Sensor
Exception Handling
Error Classification
1. Generate hypotheses
2. Order tests and execute
3. Terminate when all tests have been performed
Cause
No Success
Error Recovery
Search for alternative logical sensor
No Success
Active Perception Routines
Mission Planner/ Task Manager
Mission Planner/ Task Manager

Figure 2.4: Error Classification and Error Recovery steps of SFX-EH [CM03].

mission failure and gives control to the planner.

2.4 Modeling

A lot of research on diagnosis and repair for robots has been conducted during the last decades. The approaches basically differ in the application domain (software, hardware, behavior, or an integration of all), in the methodology used, and if diagnosis and repair are combined. In [SMW06] the authors presented a model-based approach for diagnosing faults of robot control software which uses a model of the communication between components. The approach was based on a CORBA-based communication framework while our system learns model for ROS-based communication framework presented in [QCG$^+$09].

The concrete behavior of a robotics system at run-time is determined by the behavior of its software and hardware components. In order to formalize the concrete behavior of the robotics system we need to analyze the computation and interaction between its components. The idea behind the abstract behavior model is similar to the models described by the contribution [FSW99] which automatically generates a diagnosis model from one of the most widely used hardware description languages *VHDL* (Very High Speed Integrated Circuit Hardware Description Language). The approach adapts model-based system description to derive model from VHDL program. Our model learning follows the model representation strategy specifically from the communication between the software and hardware components. The LAAS contribution [IGI11] presents an approach in order to learn the global behavior of a robotics system from its observation data. The work models the behavior of the robotics system at run-time using dynamic

Bayesian and decision networks. The learning methodology used in the work is similar to ours. It records the raw data from robot sensors, defines state variables, constructs a Bayesian graph, and then finds conditional probability distribution that fits the training raw data. We also have a learning phase and record raw data from robotic systems and constructs a logical model instead of using probabilistic model.

The research work [CS01] learns the diagnosis model for the dynamic system by generating system parameters bounds. Likewise the contribution [LC05, CS01] presents a model-based system approach First Priority Diagnostic Engine (FPDE) for fault diagnosis. In order to generate diagnosis model for the diagnosis process the EPDE engine offers a module called *bound generator* which calculates maximum and minimum bounds of the robot parameters by using kinematics equations while computation. The set of the bound parameters is used as a model in order to find discrepancy in the observed behavior.

Automated learning of the communication models for robot navigation software is presented in [KSW08]. The approach identifies different types of communication patterns which are used in the diagnosis model. We reuse this idea, however, by using an intelligent automated selection we avoid the manual setting of many parameters which is necessary in that approach. In the context of hardware diagnosis and repair the authors of [BHSW07] presented a mechanism for fault diagnosis and repair of robot drives at runtime. It provides a control framework which is capable of reconfiguring the control functions of the drive based on detected faults. In [ZSM$^+$13] an integrated diagnosis and repair approach for ROS-based robot system is presented. The approach uses a model-based approach and is able to deal with faults in software and hardware. Within this paper we present an extension to this approach. The authors of [ZKH$^+$01] presented an hybrid diagnosis approach for a printer system where mode estimation for continuous sensor measurements was combined with a decision tree using discrete components modes. The diagnosis approach presented in this paper is related to this work because it uses correlations of signals to generate input for a discrete model-based approach.

In [KKR13] an approach for on-line diagnosis of components of autonomous system is presented. The approach compares pairs of sensor signals in order to detect faulty components. The approach is similar to the presented approach but it needs to know the structural model in advance. Moreover, the signals are only linear correlated. In contrast our approach learns the structural model automatically and correlates the signals qualitatively over a period of time. A different approach using particle filters for estimation the mode of robot hardware was presented in [VVGG$^+$04]. In the domain of diagnosis in multi robot systems the work in [MTT06] uses communicating automatons where the work in [PK06] uses causal models and a hypotheses generation and testing strategy. The latter moreover provides pre-defined repair action and the learning of new fault types.

The methodology for learning model presented in this dissertation follows the approach for abstract logical model defined in the consistency-based diagnosis [Rei87]. It uses abstraction of real and correct behavior of the system, such abstract models have also been used in different areas [MSS95, MS96] . There are a number of approaches for modeling highly dynamical systems for diagnosis including probabilistic automatons or finite state machines [Ste06].

Chapter 3

Prerequisites

The presented work uses the concepts of Robot Operating System (ROS)[1], First Order Logic (FOL), and Model-Based Diagnosis (MBD) system. This chapter gives a brief understanding about the basic concepts and terminologies related to *ROS, Logic,* and *MBD* system.

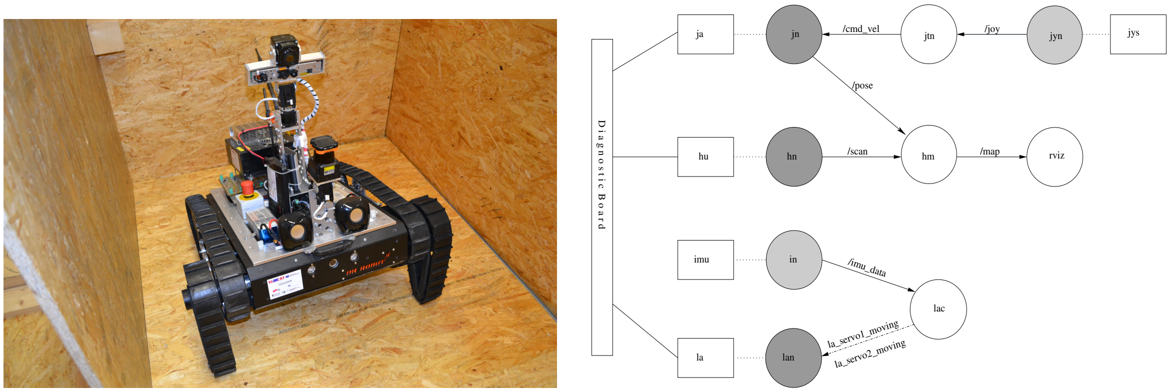
3.1 Overview

Our work is based on ROS because this robotic framework is gaining popularity among the robotics labs around the world. All modules of the presented work function as ROS units and communicate with each other under this platform. The scope of the work is also the ROS-based robotics systems, therefore, it is important to introduce what the ROS platform is, what are its basic building units, and how these units work together in order to accomplish a task.

The second necessary important system which our diagnosis system is based on, is the logic. The diagnosis engine module used in this work is specifically based upon the propositional and predicate logic. It uses propositions and Horn-clauses in order to deduce diagnosis. Therefore, it is also necessary to get the logic briefly introduced here. We give a brief introduction of what is propositional logic and how does it provide a basis to the predicate logic.

As the presented work is a model-based diagnosis system, the basic concepts of the model-based diagnosis are also briefly given at the end. The diagnosis concepts are explained through definitions and examples. In the following section we give an example which will be used in this and the following chapters as a running example. The definitions and concepts used in this work will be clarified through this running example.

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



(a) (b)

Figure 3.1: Simple control architecture (b) for the search and rescue robot (a). Rectangles represent hardware modules. Gray circles represent hardware nodes. Dark gray circles represent hardware driver nodes with switchable hardware devices. White circles represent normal software nodes. Solid arrows represent publisher/subscriber communication. Dot-dashed arrows represent service calls. Dotted lines represent hardware connections.

3.1.1 Running Example

We consider a teleoperated mapping scenario using the TEDUSAR search and rescue robot as shown in Figure 3.1 part (a), while a simple robotics system is depicted in part (b). The robotics system comprises five hardware components and nine software components. The hardware components include *jaguar* (ja) base of robot, *IMU* sensor, *Hokuyo* (hu) laser sensor, laser alignment (la), and *joystick* (jys). Software components are to control hardware and perform required task for building a map. *Jaguar_teleop_node* (jtn) provides command velocities on topic */cmd_vel* to *jaguar_node*(jn) which moves jaguar base (ja) and publishes odometry (robot's position) information on the topic */pose*, *hokuyo_node* (hn) is driver for Hokuyo laser sensor and provides laser data on the topic */scan*, IMU measures orientations and its node *imu_node* (in) publishes the data on the topic */imu_data*, and laser alignment system contains two servo motors controlled by *laser_alignment_node* (lan) in order to keep the Hokuyo sensor parallel to the surface in both pitch and roll orientation by using data from IMU sensor through *laser_alignment_control* (lac). Hector mapping (hm) builds map of the environment using robot position and laser data. The software component *rviz* visualizes the map, and *joy_node* (jyn) provides joystick commands on topic */joy*. Except *Joystick* and *IMU* sensors all other three hardware components are switchable meaning they can be automatically switched `ON/OFF` through diagnostic board. The task of the robot is to build a map of an unknown environment on uneven surface, while being teleoperated through a joystick.

3.2 Robot Operating System (ROS)

Robot Operating System (ROS) is an open-source, meta-operating system that primarily runs on Unix-based platforms. Basically ROS is not an operating system like its name suggests [QCG$^+$09], rather it is a framework that works under the umbrella of an operating system, e.g., Linux. It provides functionality for hardware abstraction, low-level device control, and message passing between nodes. It provides commonly used operations for high-level applications, a number of tools for the organization and execution of the development process, a number of means of inter-software communication, basic data types, and a number of off-the-shelf third-party modules. It is a common framework to the community of robotics research for sharing and utilizing already built and existing libraries and working modules. Every robotic system built under ROS contains a number of running processes performing specific tasks, communicate with each other, and share processed information using strictly typed messages. ROS provides a *peer-to-peer* topological mechanism in order to deal with such concurrently running processes connected with each other at run-time. ROS support for an operating system is wide, e.g., Linux, Windows, OSX, Debian, etc. It is not one programming-specific framework, rather it supports programs written in *C++, Java, Python, Lisp*, etc. This amazing feature enables every software programmer to use ROS. Its build system uses *CMake* for a modular build inside a code tree. A ROS executable entity called *node* with a list of parameters which makes it possible to re-use it outside of its original context. This enables its users to easily execute these nodes on a different platform. ROS also offers a comprehensive documentation for tutoring its new users in a very easy way by providing easy-to-implement examples in different languages, e.g., English (mostly), German, French, Italian, Japanese, Simplified Chinese, etc. Moreover, it offers visualization tools (e.g., *Rviz*), and a number of simulators (e.g., *Gazebo*) for different robotics systems where new users can evaluate their programs. ROS is continuously evolving framework over the time in order to cope with new requirements of the community.

The robotics system in the running example (Sec. 3.1.1) functions under ROS platform. It uses a number of software components based on ROS, namely jaguar_node (jn) , hokuyo_node (hn), imu_node (in), laser_alignment_node (lan), jaguar_teleop_node (jtn), hector_mapping (hm) node, laser_alignment_control (lac) node, joy_node for Joystick (jys), and $rviz$ node for visualization.

3.2.1 ROS framework architecture

The main idea behind ROS framework implementation is a distributed system of running processes that are individually designed and loosely coupled at run-time. Communication between these processes is established either by a send/receive mechanism on an appropriate channel. Another mechanism similar to remote method invocation (RMI) that ROS implements is request/reply interactions between processes. As stated earlier ROS provides a structured communicational layer above the hosting operating system (e.g., Linux). At this communicational layer ROS employs *XML-RPC* (Extensible Markup Language Remote Procedure Call) for connection negotiation and configuration, which is supported by all major computer programming languages.

3.2.1.1 ROS Master

ROS provides a structured communication layer for robotics systems containing a number of processes. The number of the processes grows larger if the robotics system is getting more complex and performing many functions at a time or the system is a multi-robot system. In such a case there is a need for a mechanism to monitor communicating entities, to allow them to find each other, and to lookup message passing between these entities. This lookup mechanism and necessary services are provided by a ROS central core and named service called *master*.

Definition 3.1 (ros master). *The ROS Master is the core of the ROS, which provides a lookup mechanism in order to keep the standards and the goals provided by ROS.*

Some of the important services that *master* provides are:

* to offer APIs and libraries based on XML-RPC protocol.
* to enforce communicational standards.
* to provide registration services to new coming processes.
* to use naming services for executable entities to uniquely identify a process.
* to help the entities find each other for exchanging messages.
* to establish a peer-to-peer way of communication among multiple processes.

In order to facilitate running processes in a robotic system under ROS framework, it provides a lookup mechanism that helps these processes to find, request and reply to each other at runtime. This lookup mechanism is provided by ROS *master*. ROS architecture uses workspaces in the form of *stacks* and *packages* to ensure reuse without any naming ambiguity. Executable processes in ROS are *nodes* which communicate with each other using *topics or services* by exchanging typed *messages*. ROS frame architecture also provides a *parameter server* which handles node parameters and plays an important role in reconfiguring them at runtime.

3.2.1.2 ROS basic building units

The fundamental building units and concepts of the ROS implementation are *messages, topics, nodes, packages, stacks* and *services*, etc.

A message is a simple named data structure that possesses information about some component or component's property. It contains fields of standard primitive types, e.g., float, integer, etc.

Definition 3.2 (**message**). *A message $\mathcal{M}$ is a strictly typed data structure.*

Example 3.1. *(Continued) Command velocity message contains information about linear and angular velocities.*

A message can be nested meaning that a message within a message is allowed. A message is always sent and received through *topics*.

Definition 3.3 (**topic**). *A* topic τ *is a named channel which carries messages.*

The communication over the *topics* is of streaming, many-to-many, and unidirectional nature. Each topic is strongly typed by its message type. The transport of messages over the topics is TCP/IP-based and UDP-based.

Example 3.2. *(Continued) Running example uses different topics, e.g., $/imu\_data$ transports position of the robot in x-y-z-w quaternions, and the topic $/scan$ carries laser sensor messages about distances of the obstacles in front of the sensor.*

A node uses topics to send and receive messages to/from other nodes.

Definition 3.4 (**node**). *A node η is an executable software program that performs computations. It can have a number of topics for sending (publisher) messages and/or a number of topics for receiving (subscriber) messages. It can be represented as tuple $\langle \eta, Pb, Sb \rangle$ where:*

- *η is the name of the node.*
- *$Pb = \{\tau_{p1}, \tau_{p2}, ..., \tau_{pn}\}$; set of n publishing topics which the node uses to send messages.*
- *$Sb = \{\tau_{s1}, \tau_{s2}, ..., \tau_{sm}\}$; set of m subscribing topics which the node uses to receive messages.*

Definition 3.5 (**publisher, subscriber**). *A publisher to a topic τ is a node η which sends messages on τ. Likewise, a subscriber to a topic τ is a node $\mathcal{N}$ which receives messages from τ.*

A ROS node is the basic executable entity or process which performs computation and communicate with other nodes using (1) *topic*, (2) *services*, and/or (3) parameters via *Parameter*

Server. The node which sends message is called *publisher* and the recipient node is called *subscriber*. A node can be either *publisher* or *subscriber* or both. Typically a robotic system comprises many such nodes, each (*publisher/subscriber*) for a specific task, e.g., rotating the robot wheels. A node can be developed in any ROS supported computer language like *C++, Java, Python,* etc. Each active node has a unique name through which it is identified in the system.

Example 3.3. *(Continued) The node $jaguar\_node$ (jn) is subscriber to the topic $/cmde\_vel$ and is used to drive the robot base (jaguar), the node $hector\_mapping$ is publisher to topic $/map$ to send map to $rviz$ node which is subscriber to $/map$ topic.*

All nodes of related functionality are combined in a package. A node cannot exist without a ROS package which is simply a directory that contains all related executable nodes, their source files, an XML file describing the package and stating its dependencies. In general a package is a container for the nodes; i.e., $\mathcal{P} = \{node_1, node_2, ..., node_n\}$ for $n \geq 1$. The *package* is a directory under a ROS path, and can depend upon other packages. Every package must contain two necessary files namely *manifest.xml* and *CMakeList.txt*. The file *manifest.xml* provides meta data about the package whereas the *CMakeList.txt* uses *CMake* to build specified nodes of the package. In addition to node source files, a package may also include configuration files, dataset, ROS-independent library, third-party software, or anything useful for developing a module. However, it should not contain too much software material to avoid heavy weight, porting and usage difficulty.

Example 3.4. *(Continued) The node $hector\_mapping$ (hm) is from a package named $hector\_mapping$[2]. The name of the package is same as its node's name.*

Like *package*, a stack is simply a directory which is a primary mechanism in ROS system for distributing software for code sharing purposes. A stack is a collection of one or more packages that collectively provides a functionality. It can depend upon other stacks which become its dependencies. Each stack has a manifext (stack.xml) file which declares meta data of the stack and dependencies on other stacks. A stack only contains packages; it cannot be nested within another stack. A stack is a container for all packages that provide related functionalities. It can be represented as a set of packages; i.e., $\mathcal{S} = \{pkg_1, pkg_2, ..., pkg_n\}$ for $n \geq 1$.

Example 3.5. *(continued) A ROS stack named* Navigation[3] *contains all packages necessary for automatic navigation.*

A *service* offers a communication of request/response mechanism. Unlike *topic*, the services need two kinds of messages; one for the request and second for the reply. A node can send

[2] http://wiki.ros.org/hector_mapping

[3] http://wiki.ros.org/navigation

a request and the service returns a response to the node after fulfilling the request. The topic-based communication is a flexible model, but its many-to-many broadcasting nature of publishing information is not appropriate. A service has a name and a pair of messages one for the *request* and one for the *response*. Another difference between topic and service communication is that more than one node can publish data on a single topic whereas this is not in the case of service. Only one node can advertise a service of a specific name.

Definition 3.6 (service). *A service is a named mechanism that offers remote procedure call (RPC) request/reply type interactions. A service receives a request and sends a reply after completing the request.*

Example 3.6. *(Continued) The node $hector\_mapping$ (hm) provides a service named $dynamic\_map$ which provides map data in response if it receives a request for the map data.*

A node can also use, store and/or receive parameters at runtime. A parameter is of a certain primitive type that can have a default value if not provided during instantiating the node. Using this facility a node can reconfigure some behavior by changing value of its parameter at runtime. A parameter server runs inside *ROS Master*. It uses the XML-RPC mechanism to send parameter values to the nodes.

Example 3.7. *(Continued) The node $hector\_mapping$ (hm) uses a number of parameters, e.g., $map\_resolution$ parameter is the length of grid cell. Its value 0.05 means that one cell in the grid (map) represents a distance of $5cm$.*

3.2.1.3 Action Server

ROS services provide request/reply interaction, however, if a service takes longer then the request cannot be cancelled once it is made. An Action server is an executing entity that executes long-running goals. It also provides periodic feedback during execution, and it can be preempted when necessary. After completing the execution it sends back the result. An action server and client uses four kind of messages, namely goal, result, feedback, and cancel.

3.2.2 ROS framework distributions

ROS comes in the form of distributions. Early distribution of ROS are *Box Turtle, C Turtle,* and *Diamondback* while currently circulating distributions are *Electric, Fuerte,* and *Groovy*. The most recent and uptodate distribution is *ROS Hydro Medusa* which is now available for three versions of *Ubuntu* 12.04 (Precise), 12.10 (Quantal), and 13.04 (Raring). A recent distribution can be easily downloaded and installed[4].

The main idea behind ROS is to have a distributed system of programs called $nodes$ that are individually designed and loosely coupled at run-time. Communication between nodes is established either by a publisher/subscriber mechanism on appropriate topics or by calling services on responsible nodes. $Topics$ are strings that uniquely identify communication channels. Services implement request/reply interactions between nodes and are similar to remote method invocation (RMI). $Messages$ are simple named data structures that are passed between nodes.

3.3 Logic Preliminaries

The history of *Logic* is thousands of the years long, but the propositional logic was presented more than one and a half century ago by G. Boole [Boo54]. This contribution presents logical concepts in a comprehensive simple way using "+" and "." symbols for disjunction and conjunction logical operation. The basic but modern logical concepts are thoroughly covered by [MH04]. Logic is possibly related to how to evaluate reasoning and arguments in order to separate truth from falsehood. It can be considered as a science of correctness and incorrectness of the reasoning about a situation which means constructing arguments about the situation. It aims to develop formal languages to model such situations in order to formally reason about them. The presented diagnosis system is based upon "first principles theory" which uses a language representation based on first-order logic for logical formalization of system description, and computing the diagnosis through logical reasoning.

3.3.1 Propositional Logic

Propositional logic is a branch of logic that studies about the *propositions*, and logical relationship and properties between them. It is also called *sentential logic* or *statement logic* for the fact that it is based on indivisible sentences or statements called *declarative sentences* or *propositions*. A proposition can always be argued as being *true* or *false* and not both.

Definition 3.7 (**proposition, declarative sentence**). *A proposition or declarative sentence is an indivisible unit statement that can be argued as either true or false but not both.*

Example 3.8. *These are some propositions: (1) "Bulb is ON", (2) "Camera is not faulty", (3) "Voltage is high in the wire", (4) "It is raining", etc. There are some sentences that cannot be argued as being true or false so they are not propositions like: (1) "May you succeed", (2) "one, two, three, go!", (3) "Could anybody please bring me a pen?", etc.*

[4] http://wiki.ros.org/ROS/Installation

Propositional logic uses *propositional terms* (e.g., **p,q,r**,..) also called propositional atoms to represent the propositions, and propositional connectives in order to construct more complicated propositions.

Definition 3.8 (**logical connectives**). *Propositional logical connectives are negation ($\neg$), conjunction ($\wedge$), disjunction ($\vee$), and implication ($\rightarrow$). The negation logical operator ($\neg$) is unary (e.g., $\neg p$) while $\wedge, \vee$, and $\rightarrow$ are binary operators (e.g., $p \rightarrow q$). The $\neg$ operator toggles its operand's truth value, the $\wedge$ operator produces true when its both the operands are true, the $\vee$ operator returns true if any of its operands is true, and the $\rightarrow$ operator produces always true except when its first operand (assumption) is true but second operand (conclusion) is false.*

The logical operators $\neg, \wedge, \vee$, and $\rightarrow$ represent English words “not”, “and”, “or”, and “if..then..” respectively.

Example 3.9. *Considering three declarative sentences “Bulb is ON”, “Voltage is high in the wire”, and “Camera is not faulty” with the propositional atoms “b” and “v”, and “c” respectively. The following compound propositional logical sentences are as:*

$\neg b \wedge v$ **: Bulb is OFF and Voltage is high in the wire.**
$\neg c \vee \neg v$ **: Camera is faulty or Voltage is low in the wire.**
$b \rightarrow v$ **: If Bulb is ON then Voltage is high in the wire.**

All possible strings made up of propositional terms and propositional logical connectives make a language called propositional language L. A string may contain brackets ”()” to specify the priority of a logical operator. The logical truth-value of a propositional string depends upon the logical operator used and the truth-values of the propositional terms.

Definition 3.9 (**propositional language**). *A propositional language L is a set of strings over alphabet comprising propositional terms and connectives; i.e., $\sum = \{p, q, r, ..., \neg, \wedge, \vee, \rightarrow, (,)\}$. Each string $s \in L$ is called a propositional word in language L.*

A possible string of L over alphabet $\sum$ defined above, is ”$(\neg \wedge q) \rightarrow p$” but inspecting this string closely it turns out into no logical meaning, because the negation logical connective ($\neg$) is without an operand term. Hence, it is not always the case that every string $s \in L$ is meaningful for propositional logic. A string needs to follow certain rules to qualify for being propositional formula properly called *well-formed formula (wff).*

Definition 3.10 (**well-formed formula**). *A string $s \in L$ is well-formed formula (wff) if it satisfies following rules:*

1. *Every propositional atom p is a well-formed formula.*

2. *If F is a well-formed formula then $(\neg F)$ is also a well-formed formula.*

3. *If F_1 and F_2 are well-formed formulas then $(F_1 \wedge F_2)$, $(F_1 \vee F_2)$, and $(F_1 \to F_2)$ are also well-formed formulas.*

The above definition of the well-formed formula can be compactly defined in *Backus Naur form (BNF)* as:

F ::= p | (¬ F) | (F∧ F) | (F ∨ F) | (F→ F)

F is a well-formed formula which can be either a propositional atom p or any already recursively constructed well-formed formula.

Example 3.10. *"$(((\neg q) \vee p) \to r) \to ((p \vee q) \vee r)$" is a valid string s of L, i.e., $s \in L$, and it is well-formed formula because it satisfies the above defined rules whereas the string "$(\neg \wedge q) \to p$" is not a wff.*

A propositional formula can be proved from other propositional formulas using the rules of deduction; i.e., inferring a conclusion from other formulas called primeses. It uses rules of deduction e.g., $\neg\neg q$ can be replaced with q or vice versa. Such rules can be applied in succession to reach to the conclusion from the primeses. The symbol used for this provability is $\vdash$ which is read as "yields" or "proves".

Definition 3.11 (primes, sequent). *If a formula β can be obtained from set of formulas $\Gamma = \{\alpha_1, \alpha_2, \alpha_3, ..., \alpha_n\}$ by applying proof rules in succession, then $\Gamma \vdash \beta$ holds otherwise not. The formula β is called conclusion, $\alpha_1, \alpha_2, \alpha_3, ..., \alpha_n$ are called primeses, and the expression $\Gamma \vdash \beta$ is called* sequent. *The primeses are always assumed to be already concluded.*

Definition 3.12 (deduction). *A deduction is a logical process of reasoning in which conclusion is drawn by previously known facts called premises. It provides absolute proof of the conclusions given that the premises are assumed to be true.*

Example 3.11. *To judge if $p \wedge q, r \vdash q \wedge r$ holds or not? we start with primeses $p \wedge q$ and r which are already assumed to be true. If $p \wedge q$ is true then it can be replaced with either p or q, let replace is with q. Now we are left with q and r which both are true giving $q \wedge r$ as true. It concludes that $q \wedge r$ can be obtained from formulae $p \wedge q$ and r resulting $p \wedge q, r \vdash q \wedge r$ holds.*

Proving a formula can lead to contradiction if there are contradictory primeses in a sequent. Assume if a sequent contains primeses like $\alpha_i = p$ and $\alpha_k = \neg p$, that means both p and $\neg p$ have already been concluded which is logically not possible; i.e., $p \wedge \neg p$ always yield $false$. This introduces contradiction and sequent concludes always $false$; i.e., $\Gamma \vdash \perp$ for $\Gamma = \{p, \neg p\}$.

Definition 3.13 (contradiction). *Contradiction is an expression of the form $\alpha \wedge \neg\alpha$ or $\neg\alpha \wedge \alpha$ where α is any logical formula. It always results false and can be represented by bottom symbol $\perp$.*

Definition 3.14 (tautology). *Tautology ($\top$) is a logical expression whose value is always $true$. It is of the form $\alpha \vee \neg\alpha$ or $\neg\alpha \vee \alpha$ or $\alpha \to \alpha$ where α is any logical formula.*

If a conclusion β can be obtained from primeses $\alpha_1, \alpha_2, \alpha_3, ..., \alpha_n$ then $\alpha_1, \alpha_2, \alpha_3, ..., \alpha_n \vdash \beta$ is said to be valid. It is not difficult to show that for all valuations in which all propositions $\alpha_1, \alpha_2, \alpha_3, ..., \alpha_n$ evaluate to true the β also evaluates to true.

Definition 3.15 (semantic entailment). *If for all valuations in which all $\alpha_1, \alpha_2, \alpha_3, ..., \alpha_n$ evaluate to true, β also evaluates to true as well, then*

$$\alpha_1, \alpha_2, \alpha_3, ..., \alpha_n \vDash \beta$$

holds. $\vDash$ is called semantic entailment *relation.*

Example 3.12. *The semantic entailment $p \vee q \vDash p$ does not hold because $p \vee q$ can be true if p is false and q is true. However, $\neg q, p \vee q \vDash p$ holds because when p is true then $p \vee q \vDash p$ is also true because q is always false due the fact that $\neg q$ is true. Another example is $p \wedge q \vDash p$ holds because p has to be true keeping $p \wedge q$ true.*

Definition 3.16 (literal). *A literal L is a propositional atom or the negation of propositional atom. L is positive literal while $\neg L$ is a negative literal. A clause D is a finite disjunction of literals; i.e., $D = \{L_1, L_2, , ..., L_n\}$ for $n \geq 0$.*

Definition 3.17 (conjunctive normal form). *A logical formula C in conjunctive normal form (CNF), is a conjunction of the clauses D of literals L as:*

L ::= p | ¬ p
D ::= L | L ∨ D
C ::= D | D ∧ C

Example 3.13. *$q \wedge (p \vee \neg r) \wedge (p \vee \neg q \vee r)$ and $(\neg r \vee \neg p) \wedge (\neg p \vee q)$ are CNF formulae whereas $(\neg(r \vee q) \wedge p$ is not CNF formula because $\neg(r \vee q)$ is not a literal due to $r \vee q$.*

Definition 3.18 (horn clause). *A Horn clause is a clause with zero or one positive literal. If positive literal is exactly one then it is a definite Horn clause.*

Example 3.14. *$(\neg p_1 \vee \neg p_2 \vee ... \neg p_{n-1} \vee p_n)$ is definite Horn clause but $(\neg p_1 \vee \neg p_2 \vee ... \neg p_n)$ is not. A definite Horn clause can be written in the form of implication like $p_1 \wedge p_2 \wedge .... \wedge p_{n-1} \to p_n$.*

Definition 3.19 (horn formula). *A Horn formula H is a conjunction of definite Horn clauses. Following Backus Naur form (BNF) defines H:*

$$\begin{aligned}&\text{P} ::= \top \mid \bot \mid \text{p}\\ &\text{A} ::= \text{P} \mid \text{P} \land \text{A}\\ &\text{C} ::= \text{A} \rightarrow \text{P}\\ &\text{H} ::= \text{C} \mid \text{C} \land \text{H}\end{aligned}$$

Example 3.15. *The examples of Horn formulae are:*

$$\begin{aligned}&(q \rightarrow r) \land (p \land r \rightarrow q) \land (p \land s \rightarrow p)\\ &(p \land r \rightarrow \bot) \land (q \rightarrow r) \land (q \rightarrow r)\\ &(q \land r \land s \land \rightarrow r) \land (q \rightarrow s) \land (r \rightarrow p)\end{aligned}$$

and the following examples are not Horn formulae:

$$\begin{aligned}&(r \rightarrow q) \land (r \land s \rightarrow p) \land (s \rightarrow \neg p)\\ &(\neg q \land s \rightarrow r) \land (q \rightarrow r) \land (q \rightarrow r)\\ &(s \land p \land \rightarrow p \land q) \land (q \rightarrow s) \land (r \land p)\end{aligned}$$

First logical expressions is not Horn formulae due to $s \rightarrow \neg p$ as it is not a Horn clause due to $\neg p$. Second expression also contains a non-Horn clause because of $\neg q \land s \rightarrow r$, and the third expression contains two non-Horn clauses $(s \land p \land \rightarrow p \land q)$ and $(r \land p)$.

3.3.2 Predicate Logic

Propositional logic uses propositional atoms, e.g., $p, q, ..$, which are just symbols and not very expressive to convey the meaning of the sentences. The predicate logic is an extension to the propositional logic but is not a replacement. It extends the propositional logic with more concepts, e.g., introducing *quantifiers* to range over the individual elements of a set. Moreover, to make a logical sentence more expressive and understandable, it introduces predicate symbols that act like functions that take zero or more arguments. It uses the concepts of sets and variables for accessing individual element of those sets, can quantify the set's element, therefore, predicate logic is also called *first-order logic*.

Definition 3.20 (first-order logic). *The first-order logic (predicate logic) is a logical system that ranges (quantifies) over individual elements (atomic entities) of a set or class.*

Unlike propositional logics it does not show limitation to express the logical aspects of natural language; i.e., it is able to access the individual element of a set or class. For instance, consider the sentence in natural language:

$$\text{“There exists a bulb which is not ON.”} \tag{3.1}$$

The propositional logic can only identify this sentence with a propositional atom *p*, but it cannot provide the way to express an individual item (bulb) from a set (e.g., many bulbs). These limitations lead to the need for a richer logic representation than propositional logic. These limitations led to the design of *Predicate Logic* (*first-order logic*).

3.3.2.1 Basics of Predicate Logic

The basic elements of the predicate logic are originally entities (called objects) which correspond to the things in the world and their properties (called predicates). In order to discuss these basic concepts in detail consider the natural language sentence (3.1) given in the preceding section. The sentence gives the information about something called “bulb” which can be either switched ON or not. This is some kind of property of the bulb. Therefore, in order to express properties and their relationship the *Predicate Logic* offers the use of *predicates*. We could write a *Bulb(doorBulb)* and *On(doorBulb)* to denote that ”doorBulb” (a bulb fixed on a door) is a bulb and is switched ON. The symbols *bulb* and *on* are known as predicates, and ”doorBulb” is their argument. A predicate can have logical value either *True* or *False*, e.g., *On(doorBulb)* is *True* if the bulb ”doorBulb” is powered ON, otherwise is *False*. Similarly a predicate “*equal*” with its two arguments, i.e, *equal(2,3)* is *False* but *equal(5,5)* is *True*.

Definition 3.21 (**predicate**). *A predicate is a name with zero or more arguments, that describes a property or relation on its arguments. A predicate with one argument is called an unary predicate, and with two arguments it is called a binary predicate. Predicates with any finite number of arguments are possible in the predicate logic.*

The predicate logic denotes properties and relations by using predicates, e.g., *red(a)* where the predicate *red* denotes the property of its argument a, and *brother(x,y)* where the predicate *brother* relates its argument x and y with each other.

As in above the predicates *Bulb* and *On* are not yet enough to express the sentence 3.1. The sentence states that there exists atleast one bulb which is switched OFF. That means the logical value of the sentence will be *False* only when all the bulbs are ON, and *True* if at least one bulb is OFF. Moreover, the predicate does not prefer to write the name of every bulb and its property because the sentence does not provide how many bulbs are there and what are their identities. Therefore, the predicate logic employs the concept of a variable which are written as lower case alphabet, e.g., *a1,b1,w,x,y,* or *z*. The variable can be considered as place holder for concrete values, e.g., doorBulb, mainBulb, etc. Using the variable ’x’ we can now specify that:

Bulb(x) : x is a bulb.
On(x) : x is powered on.

A variable is just a concrete value holder but is still not sufficient for capturing the essence of the sentence . It is required to convey the meaning of ”There exists a” which leads to the introduction of the quantifiers ∀ and ∃”.

Definition 3.22 (quantifier). *A quantifier quantifies a variable representing an element from a set. There are two quantifiers namely universal quantifier (∀) and existential quantifier (∃) which are only attached with variables. The quantifier ∀ is as ”for all” and ∃ is as ”there exists”.*

These quantifiers are always attached with the variables, i.e., $\forall x$ means “for all x” and $\exists x$ means “there exists x”. Now it is possible to express the sentence in 3.1 as a predicate formula:

$$\exists x(bulb(x) \wedge \neg on(x)) \tag{3.2}$$

which states that *“there exists an x, the x is a bulb and it is not switched ON”*. The logical formula 3.2 can also be alternatively written as:

$$\neg(\forall x(Bulb(x) \rightarrow On(x))) \tag{3.3}$$

which states that *“It is not the case that all things which are bulbs are powered ON”*. Both the logical formulas 3.2 and 3.3 are indeed equivalent semantically because these both evaluate to the same logical value *True/False* for the sentence in 3.1.

In contrary to the propositional logic the predicate logic also provides *function symbols*. Considering a predicate *mother* with two argument, i.e., *Mother(x,y)* could mean x is y’s mother. As every individual has one and only one mother so the predicate logic allows us to represent y’s mother in more direct way. Instead of writing *Mother(x,y)* we can simply write *mother(y)* to mean y’s mother. In *mother(y)* the symbol *mother* is used as a function which takes one argument and returns the mother of its argument. This is expressed by the following example:

$$\forall x(Child(x) \rightarrow Younger(x, mother(x)))$$

for the assertion *“Every child is younger than its mother”* by using unary predicate *Child* and binary predicate *Younger* with the unary function *“mother”*.

There can be different functions which different number of arguments. A *constant* in predicate logic is a concrete value or a function with zero arguments. For example, in *Brother(Hasi, Carina)* the arguments ”Hasi” and ”Carina” are the constants of the binary predicate *Brother*. As concrete values can also be regarded as functions with no argument, therefore, *constants* can only be thought of as functions which don't take arguments and we can drop the argument brackets. An *object* in the predicate logic are individual values, e.g., *Hasi*, the variable that refer to these individual values, e.g., *a*, or the function symbols that refer to some objects, e.g., *mother*(x) function refers to the object mother of x. Expressions in the predicate logic which denote *objects* are called *terms*.

Definition 3.23 (term). *A term can be defined as:*

- *A variable is a term.*
- *A zero-argument function (hence constant) is a term.*
- *If* $t_1, t_2, .., t_m; m > 0$ *are terms then a function* $f(t_1, t_2, ..., t_m)$ *is a term.*

Example 3.16. *Consider* $0, 1, ...$ *are nullary, float unary, and* $+, -$ *are binary functions. Then "*$+(1 - (float(x), 3))$*" is a term, "*1*" is also a term but alone "*$+$*" or "*$-$*" is not a term.*

A predicate formula or predicate sentence (can be simply referred as formula or sentence) can be defined over the *predicates* and the functions of the predicate logic as follows:

Definition 3.24 (formula). *A predicate formula follows the following rules:*

- *If* P *is a predicate symbol of arity* $n \geq 1$, *and if* $t_1, t_2, ..., t_n$ *are terms, then* $p(t_1, t_2, .., t_n)$ *is a formula.*
- *If* ϕ *is a formula, then* $\neg\phi$ *is also a formula.*
- *If* ϕ *and* ψ *are formulas, then* $\phi \wedge \psi$, $\phi \vee \psi$, *and* $\phi \rightarrow \psi$ *are also formulas.*
- *If* ϕ *is a formula and* x *is a variable, then* $\forall x(\phi)$ *and* $\exists x(\phi)$ *are also formulas.*
- *Everything else is not formula.*

In Backus Naur Form (BNF) for the predicate formula (ϕ*) can be described as:*

$$\phi = P(t_1, t_2, .., t_n)|(\neg\phi)|(\phi \wedge \psi)|(\phi \vee \psi)|(\phi \rightarrow \psi)|(\forall x(\phi))|(\exists x(\phi))$$

Example 3.17. *Following are some examples of formulas in the predicate logic:*

1. $Grade(Andy, Math)$ *to mean "Andy's grade in Math".*
2. $\forall x(Human(x))$ *to mean "All humans".*
3. $\exists x(Bird(x) \wedge \neg Fly(x))$ *to mean "Not all birds can fly".*
4. $\forall x(Brother(x, Mary) \wedge likes(Anna, x))$ *to mean "Anna likes Marry's brothers".*
5. $\forall x \exists y(Student(x) \rightarrow Teacher(y, x))$ *to mean "Every student has a teacher".*

3.3.2.2 Syntax for Predicate Logic

Having briefly discussed the basics of the predicate logic we can now give syntax of the predicate logic's basic constructs in BNF form:

$$\begin{aligned}
&Constant ::= \mathit{A} \mid \mathit{Bassi} \mid \mathit{Andy} \mid \ldots\\
&Variable ::= x \mid y \mid z \mid \ldots\\
&Connective ::= \neg \mid \vee \mid \wedge \mid \rightarrow \mid \leftrightarrow\\
&Quantifier ::= \exists \mid \forall\\
&Predicate ::= \mathit{Likes} \mid \mathit{Fly} \mid \mathit{Bird} \mid \ldots\\
&Function ::= father\_of \mid \mathit{plus} \mid \ldots\\
&Term ::= \mathit{Function}(Term, Term, \ldots)\\
&\qquad\quad \mid Constant\\
&\qquad\quad \mid Variable\\
&\mathit{Atomic\,sentence} ::= \mathit{Predicate}(Term, Term, \ldots)\\
&\qquad\qquad\qquad\quad \mid Term\ = \ Term\\
&\mathit{Sentnece} ::= \mathit{Atomic\,Sentence}\\
&\qquad\qquad \mid \mathit{Sentence\,Connective\,Sentence}\\
&\qquad\qquad \mid \mathit{Quantifier\,Variable\,Sentence}\\
&\qquad\qquad \mid \neg \mathit{Sentence}\\
&\qquad\qquad \mid (\mathit{Sentence})
\end{aligned}$$

3.3.2.3 Interpretation

The interpretation in the predicate logic concepts is used for defining truth and falsehood of the predicate formulae [VJ99]. It specifies a set of constants, and a set of predicates which are true. An interpretation is a model $\mathcal{M}$ for a predicate formula ϕ if it makes ϕ true in $\mathcal{M}$, and is denoted as $\mathcal{M} \models \phi$.

Definition 3.25 (model). *Let $\mathcal{F}$ and $\mathcal{P}$ are set of function and predicate symbols with a fixed number of arguments. A model $\mathcal{M}$ over $(\mathcal{F}, \mathcal{P})$ consists the following data:*

- *$A \neq \varnothing$ a set of constants;*
- *a constant value $f^{\mathcal{M}} \in A$ for each zero-argument function symbol $f \in \mathcal{F}$;*
- *a concrete function $f^{\mathcal{M}} : A^n \rightarrow A$ for each $f \in \mathcal{F}$ with arity $n > 0$, A^n being n-tuples over A;*
- *$P^{\mathcal{M}} \subset A^n$ for each $P \in \mathcal{P}$ with arity $n > 0$.*

Let we have:

man : a nullary function.
Parent(x) : x is parent.
Father(x,y) : x is father of y.

So we define a pair $(\mathcal{F}, \mathcal{P})$ such that $\mathcal{F} \overset{\text{def}}{=} \{man\}$ and $\mathcal{P} \overset{\text{def}}{=} \{Parent, Father\}$; where man is a nullary function (constant), $Parent$ is a unary predicate, and $Father$ is a binary predicate. Let consider a first-order logic formula ϕ according to the pair $(\mathcal{F}, \mathcal{P})$:

$$\forall x(Parent(man) \rightarrow \exists x Father(x, y)) \tag{3.4}$$

with a nullary function "*man*", a unary and a binary predicates "*Parent*" and "*Father*" respectively.

Definition 3.26 (ground instance). *A formula without variable is regarded as ground instance of the formula with variables. A ground instance is said to be at ground level of its formula.*

Example 3.18. *$Parent(Andy) \rightarrow Father(Andy, Bassi)$ is a ground instance of the formula ϕ 3.4. The ground instance is on the constants "Andy" and "Bassi".*

For the formula ϕ we have defined $\mathcal{F} \overset{\text{def}}{=} \{man\}$ and $\mathcal{P} \overset{\text{def}}{=} \{Parent, Father\}$; where man is a nullary function (constant), $Parent$ is a unary predicate, and $Father$ is a binary predicate. A model $\mathcal{M}$ will contain a set of concrete elements (constant) A, and the interpretations $man^{\mathcal{M}}$, $Parent^{\mathcal{M}}$, and $Father^{\mathcal{M}}$.

Example 3.19. *Having pair $(\mathcal{F}, \mathcal{P})$ for $\mathcal{F} \overset{def}{=} \{man\}$ and $\mathcal{P} \overset{def}{=} \{Parent, Father\}$ as defined above, let $A \overset{def}{=} \{a, b, c\}$, $man^{\mathcal{M}} \overset{def}{=} a$, $Parent^{\mathcal{M}} \overset{def}{=} \{a, b\}$ for the two true ground instances $Parent(a)$ and $Parent(b)$ of $Parent(x)$, and $Father^{\mathcal{M}} \overset{def}{=} \{(a, b), (b, c)\}$ for the two true ground instances $Father(a, b)$ and $Father(b, c)$. Checking whether $\mathcal{M} \models \phi$ is valid we have to check ϕ (Formula 3.4) against each element $x \in A$, therefore, replacing x with the constants in the formula:*
$Parent(a) \rightarrow Father(a, y)$ is true because $a \in Parent^{\mathcal{M}}$ and $(a, y = b) \in Father^{\mathcal{M}}$
$Parent(b) \rightarrow Father(b, y)$ is true because $b \in Parent^{\mathcal{M}}$ and $(b, y = c) \in Father^{\mathcal{M}}$
$Parent(c) \rightarrow Father(c, y)$ is true because $c \notin Parent^{\mathcal{M}}$ so $\rightarrow$ becomes true anyway
That means $\mathcal{M} \models \phi$ is valid and $\mathcal{M}$ is model of formula ϕ. Now consider a model $\acute{\mathcal{M}}$ identical to $\mathcal{M}$ except that $Parent^{\mathcal{M}} \overset{def}{=} \{a, b, c\}$, so it is not difficult to prove that $\mathcal{M} \not\models \phi$.

3.4 Model-Based Diagnosis

Model-based diagnosis (MBD) is one of the categories of the consistency-based diagnosis (CBD) discussed in Chapter 1. The presented work is basically based on MBD, therefore, it is necessary to revise important basic terminologies associated with MBD. The definitions in this section are similar as in Reiter's contribution which provides a formal framework for consistency-based diagnosis [Rei87]. In order to clarify the basic MBD terminologies for diagnosis, a simple digital circuit example is presented:

Example 3.20. *Consider a digital Full-Adder circuit with two XOR-gates (x1,x2), two AND-gates (a1, a2), and one OR-gate (o1). Each gate has two inputs and generates an output. The circuit has overall three inputs and two outputs out of the box.*

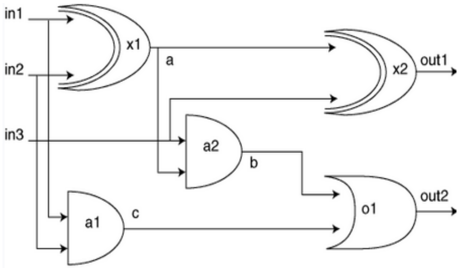


Figure 3.2: Digital Full-Adder circuit [Pal01].

In the theory of diagnosis, a system (e.g., circuit) has components, each with a description in order to describe its behavior. Typically, a system comprises a number of parts called components that collectively accomplish the task for the system.

Definition 3.27 (components). *A finite set of constants; i.e., $COMPS = \{c_1, c_2, c_3, ..., c_n\}$, where each constant c_i represents one part in a system.*

The digital circuit depicted in Figure 3.2 contains five components: two XOR-gates, two AND-gates, and one OR-gate. Therefore the set of components becomes (call it $COMP_{circ}$):

$$COMPS_{circ} = \{x1, x2, a1, a2, o1\}$$

Where x_i, a_i, and o_i respectively represent the ith XOR, AND, and OR gate of the circuit. The term *System Description* (SD) is self explanatory for it describes the system in terms of its behavior. For the sake of generality in diagnosis theory, first-order logic as a language is needed for representing system's flow of information. The unary predicate $AB(c_i)$ represents abnormality of a component c_i.

Definition 3.28 (system description). *A System Description describes how the parts of a system normally behave by appealing to the distinguished predicate AB whose intended meaning is "ABnormal".*

Hence, using first-order logic, the digital circuit in Figure 3.2 may be represented with the following System Description (Call it SD_{circ}):

$$\begin{gathered}
XORGate(c) \wedge \neg AB(c) \supset \mathit{out}(c) = \mathit{xor}(\mathit{input}1(c), \mathit{input}2(c)) \\
ANDGate(c) \wedge \neg AB(c) \supset \mathit{out}(c) = \mathit{and}(\mathit{input}1(c), \mathit{input}2(c)) \\
ORGate(c) \wedge \neg AB(c) \supset \mathit{out}(c) = \mathit{or}(\mathit{input}1(c), \mathit{input}2(c)) \\
ANDGate(a1), ANDGate(a2), ORGate(o1) \\
\mathit{input}1(c) = 0 \vee \mathit{input}1(c) = 1 \\
\mathit{input}2(c) = 0 \vee \mathit{input}2(c) = 1 \\
XORGate(x1), XORGate(x2) \\
\mathit{output}(x1) = \mathit{input}1(x2) \\
\mathit{output}(x1) = \mathit{input}2(a2) \\
\mathit{output}(a2) = \mathit{input}1(o1) \\
\mathit{output}(a1) = \mathit{input}2(o1)
\end{gathered}$$

A system can be considered as a specific part of a task domain, e.g., medicine, digital and analogue circuits, along with a set of its components and its description, taking some inputs and generating some output for its domain. The definition of a *system* in diagnosis theory is adapted as:

Definition 3.29 (system). *A system is a pair $\langle SD, COMPS \rangle$ where SD (system description) is a set of first-order sentences, and $COMPS$ (system components) is a finite set of constants.*

The digital circuit in Figure 3.2 can be represented as system $\langle SD_{circ}, COMPS_{circ} \rangle$. A system cannot be diagnosed for the faults if it is not observed continuously. The Observation at time t means the status of the system at t time. Therefore, observation provides a way to determine if the system is functioning properly or a fault has occurred. In theory of diagnosis *observation* can be defined as:

Definition 3.30 (observation). *An observation of a system is a finite set of first-order logic literals (positive and negative).*

Consider the set of observations $OBS = \{\theta_1, \theta_2.., \theta_n\}$ where each θ_i is first order logic (FOL) literals, e.g., $ok(camera), \neg ok(laser)$. A system with observations can also be represented as $\langle SD, COMPS, OBS \rangle$ which is called a diagnosable system.

Example 3.21. *Suppose that the digital circuit in Figure 3.2 is observed providing $out1 = 1$ and $out2 = 0$ on inputs $in1 = 1, in2 = 0$, and $in3 = 1$. This makes set of observations in as follows (call it OBS_{circ}):*

$$OBS_{circ} = \{input1(x1) = 1, input2(x1) = 0, input1(a2) = 1, output(x2) = 1, output(o1) = 0\}$$

With above observations the predicted (correct) behavior of the circuit (Figure 3.2) produces $output(x2) = 0$ and $output(o1) = 1$ on above inputs but the observed outputs are different, that means the circuit is faulty because the observed behavior of the circuit is different than its predicted (correct) behavior. Hence the next step is how to determine which component is faulty, thereby generating a diagnosis. From the observation OBS_{circ}, it is clear that the system $\langle SD_{circ}, \{x1, x2, a1, a2, o1\}\rangle$ (defined above) is faulty. From system description SD_{circ}, the assumption was that all system components are behaving correctly; i.e., $\{\neg AB(c_1), \neg AB(c_2), \neg AB(c_3), ..., \neg AB(c_n)\}$. Therefore, the set $SD_{circ} \cup \{\neg AB(x1), \neg AB(x2), \neg AB(a1), \neg AB(a2), \neg AB(o1)\}$ represents the system behavior on the assumption that its all components $COMPS_{circ} = \{x1, x2, a1, a2, o1\}$ are working properly. Hence, the observation OBS_{circ} conflicts with what the system $\langle SD_{circ}, COMP_{circ}\rangle$ should do; i.e.,

$$SD_{circ} \cup \{\neg AB(x1), \neg AB(x2), \neg AB(a1), \neg AB(a2), \neg AB(o1)\} \cup OBS_{circ}$$

is inconsistent.

Definition 3.31 (inconsistency). *The observation OBS conflicts with what the system $\langle SD, COMP\rangle$ should do, when:*

$$SD \cup OBS \cup \{\neg AB(c_1), \neg AB(c_2), ..., \neg AB(c_n)\}$$

is inconsistent.

where all component c_i are assumed to be functioning correctly.

Definition 3.32 (diagnosis). *A diagnosis (Δ) for $\langle SD, COMPS, OBS\rangle$ is a minimal set $\Delta \subseteq COMPS$ such that:*

$$SD \cup OBS \cup \{AB(c)|c \in \Delta\} \cup \{\neg AB(c)|c \in COMPS - \Delta\}$$

is consistent.

In other words, a diagnosis determined by a smallest set of components with the assumption that each of these components is faulty (abnormal), together with the assumption that all remaining components are working properly (normal), is consistent with the observation and the system description.

Example 3.22. *For the circuit system $\langle SD_{circ}, COMPS_{circ}\rangle$ of Figure 3.2 with observations OBS_{circ} there exists three diagnoses; $\Delta_1 = \{x_1\}$, $\Delta_2 = \{x_2, o_1\}$, and $\Delta_3 = \{x_2, a_2\}$.*

3.4.1 Computing Diagnosis (Δ)

Computing a diagnosis involves three steps: first generating all Δs such that $\Delta_i \subseteq COMPS$, after that Δs with minimum cardinality are computed, and finally consistency is tested for:

$$SD \cup OBS \cup \{\neg AB(c) | c \in COMPS - \Delta\}$$

The process of computing Δ is thoroughly discussed in Reiter's contribution [Rei87]. This process is based upon the concept of *Conflict and Hitting Sets.*

Definition 3.33 (conflict set). *A Conflict set $C = \{c_1, c_2, c_3, ..., c_k\}$ for $\langle SD, COMPS, OBS\rangle$ is a set $C \subseteq COMPS$ such that;*

$$SD \cup OBS \cup \{\neg AB(c_1), \neg AB(c_2), \neg AB(c_3), ..., \neg AB(c_k)\}$$

is inconsistent.

A conflict set C is minimal iff *no proper subset of it is a conflict set for $\langle SD, COMPS, OBS\rangle$.*

Definition 3.34 (hitting set). *A Hitting set HC for a collection of sets C, is a set $H \subseteq \cup_{S \in C} S$ such that $H \cap S \neq \{\}$ for each $S \in C$.*
A Hitting set HC for C is minimal iff *no proper subset of it is a Hitting set for C.*

The following theorem provides the basis for the *diagnosis* Δ:

Theorem 3.1. *A $\Delta \subseteq COMPS$ is a diagnosis for $\langle SD, COMPS, OBS\rangle$* iff *$\Delta$ is a minimal* Hitting set *for the collection of* Conflict sets *for $\langle SD, COMPS, OBS\rangle$.*

Example 3.23. *For the circuit system $\langle SD_{circ}, COMPS_{circ}\rangle$ of Figure 3.2, there are two minimal Conflict sets $\{x1, x2\}$ and $\{x1, a2, o1\}$ respectively to the inconsistency of:*

$$SD \cup OBS \cup \{\neg AB(x_1), \neg AB(x_2)\}$$

and

$$SD \cup OBS \cup \{\neg AB(x_1), \neg AB(a_2), \neg AB(o_1)\}$$

So given by the two minimal Hitting sets $\{x1, x2\}$ and $\{x1, a2, o1\}$, there are three diagnoses:
$\Delta_1 = \{x1\}$
$\Delta_2 = \{x2, a2\}$
$\Delta_3 = \{x2, o1\}$.

Computing Hitting sets provides an efficient and fast computation for the diagnosis computation.

3.5 Planning

In the presented work we use a repair engine (discussed in Chapter 7) which exploits planning to generate plan for its repair actions. In recent years, the planning has gained increasing attention of the researchers. Its significance can been judged by the annual *ICAPS* International Planning Competitions[5]. The concepts of the planning discussed here are similar as in [GNT04].

3.5.1 Plannning problem

The planning is on the reasoning side of acting, i.e., it uses actions of a system to change state of the system. A model for planning needs a general model for a state-transition system.

Definition 3.35 (state-transition system). *A state-transition system is a deterministic, static , finite , and fully observable system defined as a tuple* $\sum = (S, A, \gamma)$ *such that:*

- ○ *a set* $S = \{s_1, s_2, ..., s_n\}$ *of states;*
- ○ *a set* $A = \{a_1, a_2, ..., a_m\}$ *of actions; and*
- ○ $\gamma : S \times A \rightarrow S$ *a state-transition function.*

$\sum$ is 'deterministic' means it offers at most one state from a state on an action, it is 'static' meaning it does not change, it is 'finite' means it has finite number of states and actions, and it is 'fully observable' means that its all knowledge is already known. A planning problem for a $\sum$ can be define as:

Definition 3.36 (planning problem). *A planning problem is a triple* $\mathcal{P} = (\sum, s_i, g)$ *such that:*

- ○ $\sum$ *is state-transition system;*
- ○ s_i *is an initial state, i.e.,* $s_i \in S$ *from* $\sum$*; and*
- ○ g *is a goal state, i.e.,* $g \in S$ *from* $\sum$*.*

We call such a planning problem ($\mathcal{P}$) a *classical planning problem*. A solution to $\mathcal{P}$ is a sequence of actions $(a_l, a_m, a_n, ..., a_y)$ which when applied to the initial state s_i reaches to the goal state g, i.e., $\langle s_l = \gamma(s_i, a_l), \gamma(s_l, a_m), \gamma(s_m, a_n), ..., g = \gamma(s_{y-1}, a_y)\rangle$. A function represented by $\gamma(s_x, a_j)$ means that it returns a state s_j when an action a_j is applied on a state s_x [Wel99].

There can be three different kind of representations for $\mathcal{P}$, namely, *set-theoretic*, *classical*, and *state-variable* representation. As we deal with the *classical planning problem*, therefore, we briefly discuss about *classical* representation.

[5] http://ipc.icaps-conference.org/

3.5.2 Classical representation

In classical representation of planning problem $\mathcal{P}$ the states are represented by logical atoms which are *true* or *false*, and actions are represented by *planning operators* which change the truth values of these logical atoms.

3.5.2.1 States

A state s is a set of ground atoms of first-order language $\mathcal{L}$ where $\mathcal{L}$ contains predicate symbols, constant symbols, and variable symbols, but does not contain function symbols. A set $S = \{s_1, s_2, ...\}$ is a finite set containing all possible states where each s_i comprises ground atom(s) such that an atom q holds in s_j if and only if $q \in s_j$. If h is a set of literals, i.e., atoms and negated atoms, then $s_i \models h$ (s_i satisfies h) if there is a substitution σ such that every postitive literal of $\sigma(h)$ is in s_i and no negated literal of $\sigma(h)$ is in s_i.

Example 3.24. *Suppose we want to formulate a planning domain in which there are two rooms (room1, room2), one robot (robot1), and two files (fileA, fileB). The language* $\mathcal{L} = \{room1, room2, robot1, fileA, fileB, at, with\}$ *with two binary predicates* at *and* $with$*. One of the states can be* $s_k \in S = \{at(robot1, room2), with(fileA, robot1), at(fileB, room1)\}$

3.5.2.2 Planning operator

A planning operator in a classical representation acts like a transition function which changes the truth value of an atom in a state.

Definition 3.37 (planning operator). *A planning operator can be defined as a triple* $o = \langle name(o), precond(o), \mathit{effects}(o)\rangle$ *such that:*

- $name(o)$*: the name of the operator in the form* $n(x_1, x_2, .., x_n)$ *where* n *is called operator symbol, and* x_i *is a variable symbol that appears in* o*. The operator symbol* n *is unique in the language* $\mathcal{L}$*.*
- $precond(o)$*: a set of literals (atoms or negated atoms) which have to be true for invoking planning operator* o*;*
- $\mathit{effects}(o)$ *: a set of literals (atoms or negated atoms) which the planning operator* o *makes true when invoked.*

Example 3.25. *(Continued) For example the planning operator "*mov*" is used for describing movement of the robot "*$robot1$*" between the rooms "*$room1$*" and "*$room2$*". For the* $mov = \langle name(mov), precond(mov), \mathit{effects}(mov)\rangle$

name(mov): move(r,x,y)
precond(mov): $\{$at(r,x),$\neg$ at(r,y)$\}$
effects(mov): $\{$at(r,y),$\neg$ at(r,x)$\}$

The precondition describes that the planing operator mov changes the state of r from x to y.

The set of all the n planning operators is represented by $O = \{o_1, o_2, ..., o_n\}$. Having defined *states* (S) and *planning operators* (O) for the *classical planning problem*, now we can define its *domain* and *problem* as:

Definition 3.38 (planning domain). *A classical planning domain in $\mathcal{L}$ is $\sum = (S, A, \gamma)$ as:*

- *S is a set of ground atoms such that $S \subseteq 2^{\{\text{all ground atoms in } \mathcal{L}\}}$;*
- *A is a set of all ground instances of planning operators o;*
- *$\gamma(s, a) = (s \setminus \mathit{effect}^{-}(a)) \cup \mathit{effect}^{+}(a)$ if $a \in A$ is applicable to $s \in S$;*
- *S is closed under γ, i.e., if $s \in S$ then for every action a that is applicable to s, $\gamma(s, a) \in S$.*

Definition 3.39 (planning problem). *A classical planning problem is a triple $\mathcal{P} = (\sum, s_i, g)$ where:*

- *s_i is an initial state such that $s_i \in S$;*
- *g is goal state, is any set ground literals;*
- *$S_g = \{s \in S | s \models g\}$.*

3.5.3 PDDL

For the domain-problem planning specifications we use widely recognized Planning Domain Definition Language (PDDL) [KBC+98] which is a descriptive language used for description of planning problems in the planning. It is an action-centred language which is basically inspired by STRIPS planning formulations. It is more advanced than STRIPS because of the following features: type specifications for objects, negated preconditions, conditional ADD/DEL effects, and numeric variables, etc. Besides these extensions, *PDDL* is an expressive language, capable of expressing challenging behaviors in different domains. Every PDDL planning task comprises following five components:

1. **Objects:** All things in a world for a planning problem are objects, e.g., a robot, a room, or a file.
2. **Predicates:** Properties of objects, e.g., Is x a robot?, Is file x with robot y?. Is file x in room y. A predicate can be either true or false but not both.
3. **Initial state:** The beginning state of the world of a planning problem, e.g., a file and a robot are in the first room, and the file is not with the robot.
4. **Goal specification:** The state of the world we want to achieve, e.g., the file and the robot should be in the second room, and the file should not be with the robot.
5. **Actions:** An action defines the way how to change from state to state, e.g., the robot can move, and pick and drop a file. Each action contains *parameters* (action arguments), a *precondition* (the condition to be fulfilled by the parameters) and an *effect* (that takes place after the action completes).

The PDDL planning definition contains two parts, namely the *domain definition* and the *problem definition*:

3.5.3.1 PDDL-Domain definition:

The domain definition of PDDL contains *predicates* and *actions*. It may declare requirements like *strips* (only STRIPS subset), *equality* (for using predicate =), *typing* (allows use of types), and *ADL* which allows to use disjunctions and quantifiers. The domain description provides a functionality in terms of actions. An action defines its preconditions and effects. It is invoked when its preconditions become true, and then it makes its predicates true in the facts. The general syntax of a domain definition is as follows:

```
(define (domain domain_name)
  <PDDL code for predicates>
  <PDDL code for first action>
  ...
  <PDDL code for last action>
)
```

where *domain_name* is the user defined name of the domain definition.

Example 3.26. *A domain description describing the behavior of a robot for file transfer between the rooms as given below:*

```
(define (domain robot_file_transfer)
  (:requirements :strips :typing)
  (:types room file)
  (:predicates (room ?x) (file ?x)
               (at-robot ?x)(at-file ?x ?y)
               (with_robot ?x))
  (:action move
      :parameters (?x ?y)
      :precondition (and (room ?x)(room ?y)(at-robot ?x))
      :effect (and (at-robot ?y)(not (at-robot ?x))))
   (:action drop
      :parameters (?x ?y)
      :precondition (and (file ?x)(room ?y)(with_robot ?x)(at-robot ?y))
      :effect (and (at-file ?x ?y)(not (with_robot ?x))))
  (:action pick-up
      :parameters (?x ?y)
      :precondition (and (file ?x)(room ?y)(at-file ?x ?y)(at-robot ?y)
                         (not (with_robot ?x)))
      :effect (and (not (at-file ?x ?y))(with_robot ?x)))
)
```

This description defines a domain named *robot_file_transfer* for transferring files between rooms. It gives object types, predicates used, and three actions, namely *move*, *drop*, and *pick-up*. The action further includes parameters that can describe preconditions which should be initially fulfilled and the effects become true after the action is finished.

3.5.3.2 PDDL-Problem definition:

The problem definition of PDDL contains objects, initial, and goal state description.

```
 (define (problem <problem_name>)
   (:domain <domain_name>)
   <PDDL code for objects>
   <PDDL code for initial state>
   <PDDL code for goal specification>
)
```

where *problem_name* is a user defined name for the problem definition, and *domain_name* is the name of a domain for which the problem is defined.

Example 3.27. *The problem description* robot_file_transfer_problem *specifies objects, initial state, and a goal state to be achieved.*

```
(define (problem robot_file_transfer_problem)
     (:domain robot_file_transfer)
     (:objects
              fileA fileB - file
              room1 room2 - room)
      (:init
              (at-file fileA room1)
              (at-file fileB romm1)
              (at-robot room1)
              (not (with_robot fileA))
              (not (with_robot fileB))
      )
      (:goal (and (at-file fileA room2)
                  (not (with_robot fileA)))
      )
)
```

The objective of the PDDL planner is to find a set of actions (a plan) that, if applied to the initial states, reaches at goal state given in the problem description.

3.6 Significance Test

We also measure the significance of the diagnosis models learned during the learning phase (Chapter 9). Measuring significance is a statistical term which tells how much difference or relationship exists in data [EM, Die07]. Mostly, a relationship or difference already exists in the groups of the data, but whether it is a strong, moderate, or weak relationship? is solved by the significance test. The significant differences can be small or large which depends on the size of the samples.

The significance test is about setting up hypotheses and testing them for the significance. The candidate problem is simplified into two competing hypotheses: (1) a *Null hypothesis*, and (2) an *Alternative hypothesis*. The *null hypothesis* is denoted by $\mathcal{H}_0$, and it represents a theory that has been put forward, or is believed to be true, or is to be used as a basis for argument but has not been yet proved. The *alternative hypothesis* is denoted by $\mathcal{H}_a$, and it is a statement of what a statistical hypothesis test is set up to establish.

Example 3.28. *Suppose a new product P_{new} of some object has to be compared with its current product P_{curr}. The null and alternative hypotheses will be as:*

$\mathcal{H}_0$ = *The product P_{new} is not different than the product P_{curr}*

$\mathcal{H}_a$ = *The product P_{new} is different than the product P_{curr}*

A value is measured to make selection between $\mathcal{H}_0$ and $\mathcal{H}_a$. The value is called *test statistic* and is in fact a quantity calculated from the sample of data to decide whether or not the null hypothesis should be rejected. The choice of the test statistic depends on the assumed probability model and the hypotheses under question. There are different significance tests that can be employed in order to measure the significance of data, for example, z-test, t-test, etc. We discuss here z-test which we use for measuring the significance of learned diagnosis models..

3.6.1 Z-Test

Z-test is one of the tests of statistical significance which helps us decide whether or not to reject the *null hypotheses* ($\mathcal{H}_0$). It measures z-score which is the measure of the standard deviation. An associated value is p-value which provides the probability we have falsely rejected the $\mathcal{H}_0$. A very high or low z-scores associated with very small p-value, come at the tails of the normal distribution.

The significance level (α) for a given significance test is a value for which a p-value less than or equal to is considered statistically significant. The most common value used for α is 0.05, i.e., the z-score values when using a confidence level 0.05 are -1.96 and $+1.96$ standard deviations where z_α is $+1.96$ and $-z_\alpha$ is -1.96. The z-test computes z-score using the following equation:

$$z = \frac{\mu_2 - \mu_1}{\sqrt{\frac{\sigma_1}{n_1} + \frac{\sigma_2}{n_2}}} \tag{3.5}$$

where μ is mean value of the data and σ is the standard deviation:

$$\sigma = \sqrt{\frac{\sum^{n}(x-\mu)}{n}} \tag{3.6}$$

The *null hypothesis* ($\mathcal{H}_0$) is not rejected if the value $|z|$ remains in z-scores interval ($\pm z_\alpha$), i.e., if $-z_\alpha \leq z \leq +z_\alpha$ holds then $\mathcal{H}_0$ stays true. It should be noted that for $\alpha = 0.05$ the values of $-z_{0.05}$ and $+z_{0.05}$ are -1.96 and $+1.96$ respectively in the standard normal distribution [Die07].

Example 3.29. *Suppose we have two data sets X_1 and X_2 with the number of samples $n_1 = 18$ and $n_2 = 18$ respectively. The mean values of the data sets are $\bar{X_1} = 1917$ and $\bar{X_1} = 3983$. The standard deviations s_1 and s_2 are 317 and 2287 for the data sets respectively. Considering $-z_\alpha$ and $+z_\alpha$ for the significance level $\alpha = 0.05$. Using the equation 3.5 the z-test computes $z = 3.80$. As $-z_{0.05} \leq 3.8 \leq +z_{0.05}$ does not hold, i.e., $3.8 > 1.96$, therefore, the null hypothesis ($\mathcal{H}_0$) will be rejected and alternative hypothesis ($\mathcal{H}_a$) will be taken.*

Chapter 4

System Architecture

This chapter presents the architecture of the diagnosis and repair system. Moreover, it also describes the formalization of a robot architecture model ($\mathcal{RAM}$) from a robotics system, and its conversion to an abstract behavioral model used for fault detection.

4.1 Introduction

The objective of our work is to detect, localize, and repair faults in a robotics system at runtime without any external intervention. The faults can occur both in software (deviation in a publishing data frequency) and hardware (joint broken or component switched off). In order to meet these requirements under specified assumptions we developed an advanced diagnosis and repair architecture which is based on ROS and is able to inter-operate with the existing ROS-based diagnostics stack. Because of its generalized methods the proposed architecture is also applicable to other ROS-based software than robot control software.

The development of the architecture of the proposed diagnosis and repair system was guided by the following goals: (1) compatibility with ROS and its diagnostics stack, (2) minimizing the need to alter or annotate existing system code, (3) integration of both hardware and software observations and repair actions, (4) use of advanced diagnosis and repair approaches, (5) general abstract interfaces for observations and actions and (6) easy modeling of the diagnosis and repair domains. The system is publicly available on an open-source basis[1].

The architecture offers diagnosis and repair for the ROS-based robotics systems. It monitors the robotics system components, and on observing any inconsistencies between the observed and predicted behavior of the robotics system a fault is detected. The architecture offers a diagnosis module that localizes the detected fault in order to distinguish faulty component from the working ones. The diagnosis module uses logical model that represents the correct behavior of the

[1] http://www.ros.org/wiki/tug_ist_model_based_diagnosis

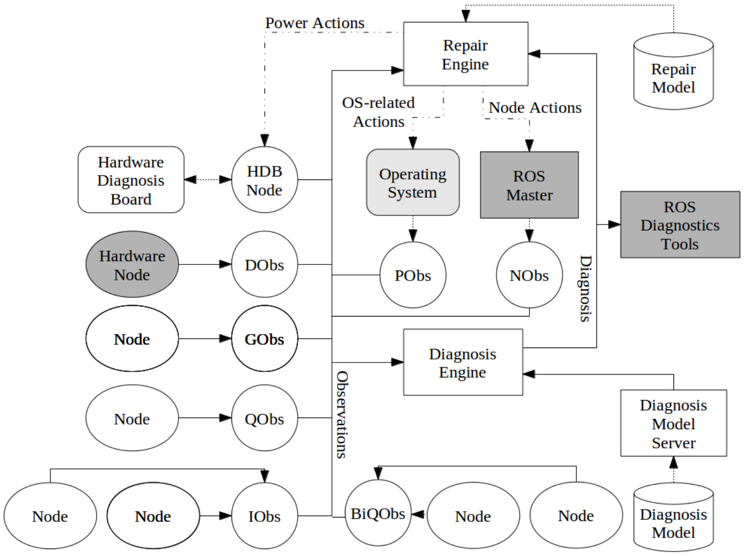


Figure 4.1: Architectural overview of presented methodology

robotics system. The repair module of the architecture is to repair the faulty component in order to bring the robotics system to its normal correct behavior. Figure 4.1 depicts the architecture and connection to the ROS nodes for software and hardware components of the robotics system.

4.2 Robotics System

A robotics system comprises a significant number of heterogeneous hardware and software components in their structure and functionality. The hardware components include sensors, motors, actuators, etc., while software components are processing entities and hardware drivers. A robotics system uses its components in order to achieve its mission successfully. Regardless of the type and mission every robotics system has a control system which controls the robot's hardware, processes internal information, and allows all components to communicate with each other to make it possible for the robot to achieve its task. In the context of the ROS-framework, a control software of a robotics system consists of a number of entities called nodes, both for the processing (e.g., navigation, image processing) and for driving the hardware components (e.g., hardware drivers). In order to achieve a task collaboratively, these nodes exchange their computed information through exchanging messages with each other concurrently over the communicational channels called topics. Each node has a number of incoming and/or outgoing topics in order to receive and/or publish the messages. There is a number of fields dedicated to a message which along with the field's values travel over a topic.

A robotics system's communication graph can be formalized through the robot architecture model ($\mathcal{RAM}$) which considers the software and hardware components of the robotics system, the communication structure between the components, and the connection dependencies in the communication. We define it in the following section:

4.2.1 Robot Architecture Model ($\mathcal{RAM}$)

The *robot architecture model* ($\mathcal{RAM}$) presents a robotic system in terms of the components (hardware and software), connections between the components (communication channels), and dependencies among components. The formalization of this architectural general view ($\mathcal{RAM}$) and the structural properties within the components of the robotics system is represented in the form of sets and functions.

Definition 4.1 (field). *A field is a named atomic data type which holds a value that travels on a node's topic.*

Definition 4.2 (driver). *A driver is a node that controls a hardware component, and acts as a hardware driver.*

Definition 4.3 (switchable hardware). *A switchable hardware is a component connected to a diagnostic board, and can be powered up or shut down.*

Definition 4.4. *The robot architecture model ($\mathcal{RAM}$) is a tuple $\langle \mathcal{COMPS}, \mathcal{T}, \mathcal{F}, \delta \rangle$ such that:*

- $\mathcal{COMPS}$ *: a set of soft and hardware components having the following disjunctive sets:*
  - $\mathcal{S} = \{s_1, s_2, ...\}$ *: a set of all software components.*
  - $\mathcal{H} = \{h_1, h_2, ...\}$ *: a set of non switchable hardware components.*
  - $\mathcal{HN} = \{hn_1, hn_2, ...\}$ *: a set of drivers (nodes) for hardware components $h_i \in \mathcal{H}$.*
  - $\mathcal{SH} = \{sh_1, sh_2, ...\}$ *: a set of switchable hardware components.*
  - $\mathcal{SN} = \{sn_1, sn_2, ...\}$ *: a set of drivers (nodes) for hardware components $sh_i \in \mathcal{SH}$.*
  - $\mathcal{N} = \{n_1, n_2, ...\}$ *: a set of nodes other than in $\mathcal{HN} \cup \mathcal{SN}$.*
- $\mathcal{T} = \{\tau_1, \tau_2, ..., \tau_k\}$ *: a set of all topics.*
- $\mathcal{F} = \{f_1, f_2, ..., f_n\}$*; a set of all fields on all topics.*
- δ *: a set of the following functions.*
  - $\delta_{out} : \mathcal{S} \longmapsto 2^{\mathcal{T}}$ *: returns all outgoing topics of a software component.*
  - $\delta_{in} : \mathcal{S} \longmapsto 2^{\mathcal{T}}$ *: returns all incoming topics of a software component.*

- $\delta_{aff} : \mathcal{S} \times \mathcal{T} \longmapsto 2^{\mathcal{T}}$: *returns those incoming topics of a software component that affect outgoing topic, i.e, the outgoing topic depends on the incoming topics.*

- $\delta_{hw} : \mathcal{SN} \longmapsto 2^{\mathcal{SH}}$: *returns the hardware related to a hardware driver node with switchable hardware.*

- $\delta_{fld} : \mathcal{T} \longmapsto 2^{\mathcal{F}}$: *returns all the fields of a message on a topic $\tau \in \mathcal{T}$.*

- $\delta_{rel} : \mathcal{F} \times \mathcal{F} \longmapsto \{True, False\}$: *returns $True$ if both fields are related, $False$ otherwise.*

Proposition 4.1. *For a topic $\tau \in \mathcal{T}$ and a node $n \in \mathcal{S}$, if $\tau \in \delta_{out}(n)$ is true then it does not hold that $\tau \in \delta_{aff}(n, \tau)$.*

Example 4.1. *(Continued) The robot architecture model ($\mathcal{RAM}$) for the running example given in Chapter 3 is as follows:*

$$
\begin{aligned}
\langle \quad & \{\{in, jyn, jn, hn, lan, jtn, hm, lac, rviz\}, \{imu, jys\}, \{in, jyn\}, \{ja, hu, la\},\\
& \{jn, hn, lan\}, \{jtn, hm, lac, rviz\}\},\\
& \{cmd\_vel, pose, scan, la\_servo1\_moving, la\_servo2\_moving, joy, map, imu\_data\},\\
& \{header, pos.linear.x, ..., map, ..., header, ranges, ...header, x, y, z, w, ...\},\\
& \{\delta_{out}(jn) = \{pose\}, \delta_{out}(hn) = \{scan\}, \delta_{out}(in) = \{imu\_data\}, \delta_{out}(jtn) = \{cmd\_vel\},\\
& \ \delta_{out}(lac) = \{la\_servo1\_moving, la\_servo2\_moving\}, \delta_{out}(jyn) = \{joy\},\\
& \ \delta_{out}(hm) = \{map\}, \delta_{out}(lan) = \varnothing, \delta_{out}(rviz) = \varnothing,\\
& \ \delta_{in}(jn) = \{cmd\_vel\}, \delta_{in}(jtn) = \{joy\}, \delta_{in}(jyn) = \varnothing, \delta_{in}(hn) = \varnothing, \delta_{in}(in) = \varnothing,\\
& \ \delta_{in}(hm) = \{pose, scan\}, \delta_{in}(rviz) = \{map\}, \delta_{in}(lac) = \{imu\_data\},\\
& \ \delta_{in}(lan) = \{la\_servo1\_moving, la\_servo2\_moving\},\\
& \ \delta_{aff}(hm, map) = \{pose, scan\}, \delta_{aff}(lac, la\_servo1\_moving) = \{imu\_data\},\\
& \ \delta_{aff}(lac, la\_servo2\_moving) = \{imu\_data\}, \delta_{aff}(jtn, cmd\_vel) = \{joy\},\\
& \ \delta_{hw}(jn) = \{ja\}, \delta_{hw}(hn) = \{hu\}, \delta_{hw}(lan) = \{la\},\\
& \ \delta_{fld}(cmd\_vel) = \{header, pos.linear.x, ...\}, \delta_{fld}(scan) = \{header, ranges, ...\},\\
& \ \delta_{fld}(la\_servo1\_moving) = \{pos\_angle, neg\_angle\}, \delta_{fld}(map) = \{header, grid[1][1], ..\}\\
& \ \delta_{fld}(la\_servo2\_moving) = \{pos\_angle, neg\_angle\}, \delta_{fld}(joy) = \{header, button_1, ..\},\\
& \ \delta_{fld}(imu\_data) = \{header, x, y, z, w\}, \delta_{fld}(pose) = \{header, twist_{xyzw}, pose_{xyzw}\}\\
& \ \delta_{rel}(grid[1][1], imu.x) = False, \delta_{rel}(pos.linear.x, imu.x) = True, ...\} \quad \rangle
\end{aligned}
$$

4.3 Diagnosis and Repair

Following the requirements described in the introduction section we developed a diagnosis and repair system whose architecture is shown in Figure 4.1. The architecture resembles five modules: (1) a set of system observers (Chapter 5) each of which observes a particular property of the robotics system, (2) a diagnosis model server (Chapter 6) which provides an abstract behavioral model describing the correct behavior of the robotics system, (3) a model-based diagnosis engine (Chapter 6) that detects the root cause if observers detect any violations, (4) a planner-based repair engine (Chapter 7) in order to find the set of repair actions for executing repair server actions for the repair process, and (5) a hardware diagnosis and repair board (Chapter 8) in order to diagnose and repair the hardware faults. Each module provides a particular functionality which is described below in more detail. The modules communicate with each other in order to pass information among them.

4.3.1 Communication

The modules in our diagnosis and repair architecture communicate with each other using some means of communication. The means of communication a module uses, depend on the nature of the module. For example the observers always output the status of components, therefore, they need unidirectional communication with other modules. A repair engine needs bidirectional communication; i.e., it sends a goal to a repair action and waits until it gets a result back. There are three types of communication means adapted to the modules.

Standard Topics and Messages: All observers (briefly discussed in the next section) publish their observations in the form of a set of first-order logic literals on the ROS topic */observations*. For instance, the string `"ok(temp_cpu1) and ok(temp_cpu2)"` represents the fact that the temperature of CPU1 and CPU2 are within their proper range. The diagnosis results are published on the topic */diagnosis* using a diagnosis message containing sets of faulty, and working components (Δ_{bad},Δ_{good}). The diagnosis message may contain multiple diagnoses.

Repair Actions: All repair actions in the system are implemented using the ROS action server/client architecture[2]. Actions use a generic action definition accepting a list of strings as parameters and returning a single standard integer value for the feedback and the result. This allows for an easy integration of new repair actions. For instance `shutdown("laser")` represents the ground repair action for shutting down the power supply for the laser sensor.

[2] For further information on the ROS action library please refer to http://ros.org/wiki/actionlib.

Individual Communication: This category comprises special means of communication used by special nodes or services outside ROS. For instance the diagnosis system communicates with the hardware diagnosis and repair board via a proprietary protocol over TCP/IP. Moreover, some OS-related repair actions like resetting the USB bus, use special OS calls, e.g., *ioctl*.

4.3.2 Observers

Observers are general software entities implemented as ROS nodes that monitor particular properties of the system. There are different kinds of observers supervising different properties. The output Θ of the observers is published on the */observations* topic as a first order logical (FOL) literals. The observation is a representation of the observed property and its status. Observations are published as a set of strings. Currently only simple literals are supported, e.g., $ok(temp\_cpu1)$(`"ok(temp_cpu1)"`) or $\neg ok(temp\_cpu1)$(`"not ok(temp_cpu1)"`). However, more complex expressions, e.g., expression with quantifiers, can be easily integrated. Observers are thoroughly covered in Chapter 6.

Please note that obviously the representation and expressiveness of observations have to match with the capabilities of the used diagnosis approach. Currently used observer types are:

Diagnostic Observer (DObs) is a connection between the existing ROS diagnostics system and the integrated diagnosis and repair system. *DObs* receives diagnostic messages and generate the output based on similar rules as the existing ROS diagnostic analyzers, e.g., $temp\_cpu1 < 40.0 \Leftrightarrow ok(temp\_cpu1)$. These observers can easily reuse the existing information coming from hardware devices.

General Observer (GObs) simply subscribes to a particular topic and uses the received messages as input. It observes the frequency $f(\tau)$ for a topic. If $f(\tau)$ is correct then it submits $ok(\tau)$, $\neg ok(\tau)$ otherwise.

Node Observer (NObs) observes the running state of a software node. It submits $running(n)$ if node n actually runs, $\neg running(n)$ otherwise.

Qualitative Observer (QObs) observes the abstract trend of a particular value in the message of a topic. It reports if a value v increases ($inc(v)$), decreases ($dec(v)$) or stays constant ($cons(v)$). The observer fits a linear regression for a given time window ω and compares it with a given slope b. For details about the abstraction process please refer to [KSW09].

Binary Qualitative Observer (BiQObs) monitors match or mismatch between two values on the basis of their qualitative trends. It outputs $matched(v_1, v_2)$ if both v_1 and v_2 have same qualitative trends, otherwise $\neg matched(v_1, v_2)$.

Hardware Observer (HObs) subscribes to the measurements and status updates coming from the hardware diagnosis board to supervise the status of the different power supply channels. It provides the observation $on(m)$ if the module m is powered, otherwise $\neg on(m)$. Moreover, it provides all current measurements and power states in a special message on the topic $/board\_measurments$. This allows further observers to reuse the information.

Property Observer (PObs) supervises properties not directly related to a topic that may affect the robot's performance like CPU or memory usage of a particular service. The observer publishes individual observations $ok(n,p)$ if a property p of a node n functions properly, otherwise $\neg ok(n,p)$.

Interval Observer (IObs) monitors the interval difference between two values and reports $equal(v_1, v_2)$ if $|v_1 - v_2|$ does not exceed a certain threshold for the values v_1 and v_2, otherwise $\neg equal(v_1, v_2)$.

Example 4.2. *(Continued) Considering running example from Chapter 3 suppose we take the set of the observers $O = \{GObs_{cmd\_vel}, NObs_{imu\_node}, PObs_{hokuyo\_node\_cpu}\}$ where $GObs_{cmd\_vel}$ publishes either $ok(cmd\_vel)$ or $\neg ok(cmd\_vel)$, $NObs_{imu\_node}$ publishes either $running(imu\_node)$ or $\neg running(imu\_node)$, and output of $PObs_{Actuate_{cpu}}$ is either $ok(hokuyo\_node, cpu)$ or $\neg ok(hokuyo\_node, cpu)$. If the the exploration and the mapping on the robotics system is running correctly then the output of the observers ($o_i \in O$) provides a list of observations; i.e., $\Theta = \{ok(cmd\_vel), running(imu\_node), ok(hokuyo\_node, cpu)\}$. Suppose that $PObs_{hokuyo\_node\_cpu}$ monitors a deviation in cpu usage then the list of observations becomes $\Theta = \{ok(cmd\_vel), running(imu\_node), \neg ok(hokuyo\_node, cpu)\}$.*

4.3.3 Diagnosis Model Server

In addition to the observations the model-based diagnosis requires a model of the correct behavior of a system in order to localize a fault. In the presented work the correct behavior of the robotics system is provided by the model server in the form of logical rules. The diagnosis model server (*DMS*) is an action server that provides the robot behavioral model at run-time when requested for the diagnosis. It enables the system to make changes in the robot behavioral model ar run-time, or add new logical rules without disturbing the rest of diagnosis system at run-time. The robot behavioral model provides an expected behavior of the robotic system which is used by the diagnosis process to capture the discrepancies if the observed behavior deviates from the expected behavior. ($\mathcal{RBM}$) is an abstract model containing a set of logical rules in order to describe the robotics system's correct behavior. Both the robot behavioral model and required observers together make a diagnosis model ($\mathcal{DM}$) for a robotics system (the automatic diagnosis model learning and observers generation are discussed in Chapter 9).

The robot behavioral model is a logical system description of the robotics system. It consists of Horn clauses and the propositions in order to represent the correct behavior of the robotics system. The Horn clauses are to obtain the efficient logical deduction and reasoning during the diagnosis process. The Horn clauses representation is also compatible with the diagnosis engine used in the presented work. It supports Horn-clauses and propositions using the concepts presented in the contribution [Rei87]. The robot behavioral model uses the rules containing literal $\neg AB(c)$ ("not ABnormal component c"), for example, $\neg AB(c) \rightarrow running(c)$ stating that a component node n is working when it is running. Clearly the clause $\neg AB(c) \rightarrow running(c)$ is a Horn clause as the literal $\neg AB$ can be represented by a propositional atom NAB or not_AB where N and $not\_$ are called *negative prefixes*.

Example 4.3. *(Continued) Considering the running example from Chapter 3 the robot behavioral model ($\mathcal{RBM}$) contains the following rules:*

$$\neg AB(jaguar\_node) \rightarrow running(jaguar\_node)$$
$$\neg AB(hokuyo\_node) \rightarrow running(hokuyo\_node)$$
$$\neg AB(imu\_node) \rightarrow running(imu\_node)$$
$$\neg AB(laser\_alignment\_node) \rightarrow running(laser\_alignment\_node)$$
$$\neg AB(laser\_alignment\_control) \rightarrow running(laser\_alignment\_control)$$
$$\neg AB(hector\_mapping) \rightarrow running(hector\_mapping)$$
$$\neg AB(joy\_node) \rightarrow running(joy\_node)$$
$$\neg AB(joy\_teleop\_node) \rightarrow running(joy\_teleop\_node)$$
$$\neg AB(rviz) \rightarrow running(rviz)$$

The robot behavior model formalization will be thoroughly defined in Chapter 6.

4.3.4 Diagnosis Module

The diagnosis module is that part in the diagnosis system architecture which computes the root cause of the failure called diagnosis (Δ) (Chapter 6 for details). The kernel of the diagnosis module mainly consists of two concurrently running subtasks; i.e., collecting observations, detecting and localizing the faults. It also links a repair engine (briefly discussed in the next section) by publishing the diagnosis. It takes as input the observations Θ coming from the observers, and an abstract diagnosis model ($\mathcal{DM}$) from the diagnosis model server. The observations are in the form of *FOL* literals while the diagnosis model (system description) comprises logical rules expressed in the form of Horn clauses. Each single observation θ_t that occurs at time t on the topic */observations* is integrated into an observation set Θ; i.e., $\Theta = \Theta \oplus \theta_t$, where

$$S \oplus \theta = \begin{cases} (S \backslash \neg L) \cup L & for\ \theta = L \\ (S \backslash L) \cup \neg L & for\ \theta = \neg L \end{cases}$$

where L is a logical literal.

Example 4.4. *(Continued) Assuming the running example from Chapter 3 we start the diagnosis module which initially has the list of observations empty* $\Theta = \varnothing$*. Now we start the observer* $GObs_{cmd\_vel}$*, and the observation* $ok(cmd\_vel)$ *appears. The diagnosis engine searches the list* Θ *for the observation* $ok(cmd\_vel)$*. As* Θ *is empty the observation is simply added; i.e.,* $\Theta = \{ok(cmd\_vel)\}$*. The same thing happens when first observations from* $NObs_{imu\_node}$ *and* $PObs_{hokuyo\_node_{cpu}}$ *appear making* $\Theta = \{ok(cmd\_vel), running(imu\_node), ok(hokuyo\_node, cpu)\}$*. Now the observations* $ok(cmd\_vel)$*,* $running(imu\_node)$*, and* $ok(hokuyo\_node, cpu)$ *appear one after another but they are not added because* Θ *already contains them. At this point of time something happens and node* imu_node *stops. The observer* $NObs_{imu\_node}$ *monitors it and changes its observation from* $running(imu\_node)$ *to* $\neg running(imu\_node)$ *which comes to the diagnosis engine. The diagnosis engine searches* Θ *for* $\neg running(imu\_node)$ *but does not find it. Before adding* $\neg running(imu\_node)$ *the counter observation* $running(imu\_node)$ *is also searched, it found and removed, and appended* $\neg running(imu\_node)$ *in the list making:*

$$\Theta = \{ok(cmd\_vel), \neg running(imu\_node), ok(hokuyo\_node, cpu)\}$$

and the process continues.

The process of computing a diagnosis is finding a set of components that are faulty and a set of components that are still working properly. Predicate $\neg AB(m)$ denotes that module m is working properly whereas $AB(m)$ denotes that the module m shows a faulty behavior. The diagnosis engine follows the principles of model-based diagnosis presented in [Rei87]. The approach uses an abstract model (diagnosis model (DM) in our case) that defines correct behavior and current observations of the system. A fault is detected if the outcome of the model and the observation lead to a contradiction. Using a hitting set algorithm, the approach calculates hitting sets in order to compute diagnoses that resolve the contradiction; i.e., they explain the component's misbehavior. The engine locates that component or set of components that are the root cause for the contradiction. The results of the diagnosis engine are published on the */diagnosis* topic. As described earlier the diagnosis message may comprise several diagnoses build up by sets of working (*good*) and faulty (*bad*) components. The union of both sets is equal to the set of all system components ($\Delta_{good} \cup \Delta_{bad} = COMPS$) for each individual diagnosis. Apart from this, it also generates diagnostic messages compatible for the ROS diagnostics stack. The diagnosis is a *single fault* diagnosis if its Δ_{bad} set contains only one component, otherwise it is *multi fault diagnosis*.

Example 4.5. *(Continued) Suppose the diagnosis module acquires the observations list* $\Theta = \{ok(cmd\_vel), running(imu\_node), ok(hokuyo\_node, cpu)\}$ *and the assumption* $\mathcal{A} =$

$\{\neg AB(jaguar\_node), \neg AB(imu\_node), \neg AB(hokuyo\_node)\}$ that all components are working correctly. The robot behavioral model ($\mathcal{RBM}$) breifly discussed in the previous section, is a system description of the robotics system. The $\mathcal{RBM} \cup \Theta \cup \mathcal{A}$ is consistent because there is no conflict set and the diagnosis engine provides only a minimal diagnosis the empty set $\{\}$. The diagnosis module publishes the diagnosis as:

$$\Delta = \{\Delta_{bad} = \{\}, \Delta_{good} = \{jaguar\_node, imu\_node, hokuyo\_node\}\}$$

Now suppose the node imu_node stops because of some unknown reasons which updates the observations list $\Theta = \{ok(cmd\_vel), ok(hokuyo\_node, cpu), \neg running(imu\_node)\}$ making $RBM \cup \Theta \cup \mathcal{A}$ inconsistent. The minimal conflict set $F = \{imu\_node\}$ is calculated because:

$$RBM \cup \Theta \cup \{\neg AB(imu\_node)\} = \perp$$

so the Hitting set algorithm of the diagnosis engine provides a minimal diagnosis set $\{imu\_node\}$ and the diagnosis module's output:

$$\Delta = \{\Delta_{bad} = \{imu\_node\}, \Delta_{good} = \{jaguar\_node, hokuyo\_node\}\}$$

Chapter 6 discusses the diagnosis process in detail.

4.3.5 Repair Module

The system architecture does not only localize faults but also offers mechanism to repair the detected and the localized faults. The task of repairing a faulty component is performed by this module of the system architecture. It uses *Repair engine* that performs two main activities: first it finds out a plan of actions to be performed for necessary repair, secondly it uses *repair action servers* to execute the computed actions in the plan. In order to describe and enable repair actions we model the repair as a planning domain. The repair engine takes current observations (Θ) and diagnoses (Δ) as input. If the repair engine receives a diagnosis message it converts the diagnosis into to a planning problem and solves it. Please note that in the case of multiple faults, the diagnosis repair module always makes a repair plan for the first diagnosis obtained in the diagnosis set.

We use the widely recognized Planning Domain Definition Language (PDDL) to represent the domain and problem definitions for the planning [KBC^{+}98]. The description of a planning problem in PDDL comprises two parts: (1) a domain description and (2) a problem description. The advantage of this approach is that a wide range of existing high-performance planners and various extensions to classical planning like typing can be easily used directly. To parse the PDDL domain description that contains all definitions of actions and domain objects we use an open-source Java-based package *pddl4j* [Pel08] and a *GraphPlan* implementation to find a plan $\mathcal{P}$ (a sequence of repair actions) [BF97]. All possible repair actions are defined in the domain description. These actions include $start\_node(n)$, $stop\_node(n)$, $power\_up(h)$ and $shutdown(h)$. The actions $start\_node(n)$ and $stop\_node(n)$ are for starting and stopping a software node n while $power\_up(h)$ and $shutdown(h)$ are for powering up and shutting down a hardware component h. The following is a possible $start\_node(n)$ action description:

```
(:action start_node
   :parameters (?n)
   :precondition (and(bad ?n)(not(running ?n)))
   :effect (running ?n))
)
```

The first action description states that the planner can power up a hardware component h only if h is off and faulty. The effect of the action is that component h is powered and works correctly. The `start_node` action description states that a software component n only if it is declared faulty and observed not running. The action's effect is that component n is running. The repair planning problem description for a diagnosis Δ and a set of observations Θ is simply a PDDL description of the initial state I and the goal G.

Example 4.6. *(Continued) Considering the diagnosis with faulty and working diagnosis* $\Delta = \{\Delta_{bad} = \{imu\_node\}, \Delta_{good} = \{jaguar\_node, hokuyo\_node\}\}$ *the planner generates sequence of repair actions* $\mathcal{P} = \langle start\_node(imu\_node)\rangle$ *by using domain action description and problem description. In this case there is only one repair action* $start\_node$ *for the component* imu_node*.*

Once the planner has found a valid repair plan $\mathcal{P}$, it starts the execution of the sequence of repair actions. The execution of the action is triggered by an invocation of the appropriate repair action server. A repair action server is a ROS action server that gets a goal from its action client (repair kernel), fulfils the goal and returns as status either `SUCCESS` if it finishes the task, otherwise `FAILED`, back to the action client. Because of the standard signature for repair action servers (a unique name and a list of parameter strings) an easy matching between the planner and the ROS-based system is possible. Please note that the planner simply waits for the completion message (`SUCCESS/FAILED`) from the called repair action server and currently does not check the actions' effect. A better strategy for the future would be to wait until the effects have been established. For instance it might take longer for a node's output topics to become correct than simply to restart the node. The repair module is discussed in detail in Chapter 7.

Example 4.7. *(Continued)* $start\_node$ *repair action server will be invoked which will start the node* imu_node*.*

4.3.6 Diagnostic Hardware Board

Our experiences and previous research work have shown that in addition to software diagnosis and repair, hardware-based diagnosis and repair is needed for systems comprising devices with no direct support for diagnosis and reconfiguration. In order to get additional information for the diagnosis system we developed a micro-controller based hardware diagnostic board and a hardware diagnosis observer [ZL13, ZL14]. The diagnostic board and the hardware observer are capable to gather the hardware related information like current measurements and power supply detection of individual components of the system. The information that a particular component draws a certain amount of current or that a device is indeed powered can assist the diagnosis process. The diagnostic board contains the following features:

1. It can sense whether a hardware component is connected to its channel.
2. If a component starts consuming more current the board can immediately sense it.
3. It offers repair actions for the hardware components by powering them up/down.
4. It can also measure the voltage level flowing in the channels.
5. It also provides small memory to save some start up configuration for powering up certain components.

The diagnostic board provides the possibility to turn *ON* and *OFF* the power supply for different system components. This allows two different kinds of repair actions related to hardware: (1) power off/on cycles and (2) hardware reconfiguration. The board is organized in 10 individual channels that can be switched individually. The central part of the hardware diagnostic board is a micro-controller embedding a TCP/IP stack. It is responsible for the current measurements, the power monitoring and the power configuration of the individual channels. The board is connected with the outside world over Ethernet, and embeds a server with a proprietary protocol. The protocol allows the client to pull information like measurements and states of channels or to initiate broadcasts on a regular basis. The diagnostic board is discussed in Chapter 8.

Example 4.8. *(Continued) Following the running example from the previous chapter, there are three hardware components which are switchable, namely $jaguar, hokuyo$, and $laser\_alignment$. Therefore the $HObs$ will sense the diagnostic boards and will give observations:*

$$\Theta = \{on(jaguar), on(hokuyo), on(laser\_alignment)\}$$

4.4 Limitations

Locating and repairing faults at run-time is related to a number of issues. The foremost problem is the complexity and overhead of the diagnosis system. In the current implementation the load on the diagnosis system depends upon the size of robotics system and on how complex its communication structure is. The overhead of the diagnosis and repair modules of the diagnosis system in terms of memory is not high because the diagnosis and repair system does not keep the data streams. However, the overhead with respect to CPU usage can increase with significant increase in the number of components being monitored. Moreover, the CPU overhead also depends upon the capability of the machine being used. Moreover, learning a diagnosis model (Chapter 7) puts more overhead in terms of cpu and memory due to spawning multiple threats for monitoring the data and communication behavior, and storing the processed data at run-time. But this overhead does not change the operation of the control software if the machine is not significantly slow.

Another problem that may cause damage to the environment or the robot can arise if there is a significant delay between detection of the faults and repairing them. This delay must be as short as possible in order to avoid physical damage. In the presented diagnosis methodology, the diagnosis process receives observations with the frequency of 10 Hz and computes diagnosis with the frequency of $1Hz$. The diagnosed faults are repaired as soon as a diagnosis arrives at the repair system. In case of the hardware faults, the delay depends upon the nature of repair action (shut_down or re_start).

The repair engine we use does not repair every kind of faults. There are different kinds of the faults, namely transient, intermittent, and permanent, etc. A transient fault is a fault which can be repaired at run-time after some tries. The example of transient faults includes a software entity crash, frequency deviation on a channel, increasing consumption of power by hardware component, etc. An intermittent fault causes a component (software or hardware) to alternatingly function correctly and incorrectly, e.g., cpu consumption starts fluctuating. The permanent faults are those which cannot be repaired at run-time, for instance when a gear is broken, or a hardware component switched off due to a cable disconnected or cut. Currently our repair system can only repair transient faults.

Another issue can be that the model created and learned does not truly reflect the correct system behavior. We overcome this issue by assuming that the phase which learns the system's model is completely fault free; i.e., there are no faults both at the software and hardware level during the learning phase (Chapter 9).

The assumptions and requirements our work considers are as follows:

1. The scope of the diagnosis system are robotics systems that are based on the ROS platform.

2. The diagnosis system only considers transient faults, not permanent ones.

3. Repairing a fault should immediately start as long as diagnosis appears.
4. The diagnosis system assumes fault free run during model learning.
5. The model is presumed to be correct, and any violation indicates a component's fault.
6. The diagnosis system should not put any significant impact on the operation of the robotics system.

Chapter 5

Monitoring

This chapter discusses how the diagnosis system monitors a system's parts in order to find inconsistencies, violations and deviations in observed behavior as compared to the predicted behavior. It gives insights of different monitoring units (observers) and their functionalities. Parts of the work presented in this chapter have been published in [LMS^{+}12].

5.1 Overview

Diagnostic reasoning requires a means of declaring a component faulty based on observed behavioral discrepancies. In order to detect the behavioral discrepancies a diagnostic process demands a continuous run-time monitoring system. The monitoring system should have the capability to observe the properties or characteristics of a robotics system's behavioral communicational graph between its different components. A robotic system at run-time uses a number of software modules in order to accomplish its task. These software modules exchange and share their information with each other to reach the goal. However, due to interaction of robotics system with a time-varying dynamic environment and the heterogeneous nature of its own components, the possibility of robotic system's malfunctioning cannot be completely neglected. The environmental effects or internal complex communication of robotic system may change the properties of running modules and their communication behavior. This possible change in the properties inside a robotics system leads to a deviation from correct behavior. For example during the outdoor navigation the *Compass* or *IMU* sensor may start functioning abnormally while passing through a building or a bridge made up of heavy iron and steel. During the fault free execution, the properties of a robotic system's modules and its communication behavior collectively reflect the correct behavior of the robotic system. The task of a monitoring system is to closely observe these properties and report violation if it finds some significant change in these properties. Of course, there must be some level of tolerance for monitoring a property, e.g., if a normal running module gets a rise in cpu usage only one time due to some operating system reasons and gets back to normal again, it is not wise to halt the whole processing system and to start it again. As the focus of the presented diagnosis and repair system is the ROS-based robotics systems, we consider the running example (Chapter 3) in order to explain the concepts of fault detection, localization, and repair both in software and hardware level in this and the next chapters.

An occurrence on a topic is a message which contains a set of fields with their values. These values can be of primitive data types (e.g., integer, float) or a complex data structure, e.g., odomerty.

Definition 5.1 (Occurrence). *An occurrence o_t appearing at timestamp t on a topic $\tau \in \mathcal{T}$ is a tuple $\langle t, V \rangle$ such that:*

- *t is the timestamp at which the occurrence appears on the topic τ.*
- *$V = \{val_1, val_2, ..., val_n\}$; a set of values corresponding to all n fields $f_i \in \delta_{fld}(\tau)$ of a message on the topic $\tau \in \mathcal{T}$.*

The set of all messages in a row on a topic makes the sequence of the occurrences for that topic. The number of occurrences per unit time is the frequency of the topic $\tau \in \mathcal{T}$.

Definition 5.2 (Sequence). *A sequence $seq(\tau, t)$ is the sequence at time t of all the occurrences on a topic $\tau \in \mathcal{T}$ since its monitoring started. The occurrences in $seq(\tau)$ are in an ascending order with respect to the timestamps in the occurrences. A sequence of occurrences between the interval of timpestamps t_i and t_j, is denoted by $seq(\tau, [t_i, t_j])$ such that $0 \leq t_i \leq t_j$. The $seq(\tau, [t_i, t_j])$ is of course a subsequence of $seq(\tau, t)$.*

It is possible that at time t one topic receives an occurrence and others do not.

Definition 5.3 (Topic Data). *A function $data(\tau, t)$ for $\tau \in \mathcal{T}$ at time t is such that:*

$$data(\tau, t) = \begin{cases} o_t & \textit{if occurrence appears.} \\ \varnothing & \textit{otherwise.} \end{cases}$$

where o_t is the occurrence at time t on the topic $\tau \in \mathcal{T}$.

Definition 5.4 (window). *A sliding window ω specifies a time duration during which the occurrences o_i are collected. After this duration is passed, the computation takes place on the collected occurrences.*

Proposition 5.1. *If a window ω_j occupies the sequence $seq(\tau, [t_i, t_j])$ of occurrences on a topic τ from timestamps t_i to t_j, the change in ω_j changes $|seq(\tau, [t_i, t_j])|$.*

Proposition 5.2. *Given two windows ω_a and ω_b such that $\omega_a < \omega_b$, then the observer is more strict (less sensitive) to violations for the window ω_a, and less strict (more sensitive) for ω_b.*

The sliding window is basically used to deal with unwanted small noises in the data and absorb them without causing violation.

Example 5.1. *(Continued) The node jtn publishes command velocities on the topic cmd_vel with the frequency 10Hz. Let us suppose the interval (Δ) between successive occurrences is 100ms with a standard deviation (σ) of 0.0002ms in a normal run. If due to some processing overload Δ_j between occurrence o_i and o_j becomes 150ms, then an observer with the window ω of size 400ms can more easily detect this violation than when having the window (ω) of size 1000ms (1sec).*

5.2 Observers

The monitoring system consists of a number of monitoring units. The scope of the monitoring system are ROS-based robotics systems whose control software consists of nodes, topics, and messages. A node possesses certain properties, for example its state (e.g., running, aborted, or stopped), its cpu power consumption, and the number of topics it is associated with, etc. Like a node a *topic* also possesses some properties, for example, the number of associated nodes, the frequency of data flowing on it, the behavior of transmission (e.g., random, regular, or periodic). The properties of a field in a *message* travelling over a topic can be, for example, its value increasing, decreasing, or remaining constant. Each of the monitoring units, called observer, monitors one of such properties to report if any violation in the property is reported.

Definition 5.5 (Observer). *An observer Obs is a tuple $\langle \mathcal{P}, \psi, \Theta \rangle$ for testing a property where*

- $\mathcal{P} = \{p_1, p_2, ...\}$*, a set of parameters for testing the property;*
- $\psi(\mathcal{P})$*, a condition to satisfy property on parameters* $\mathcal{P}$*;*
- Θ*, a set of logical literals corresponding to the observations.*

An observer is a general software entity implemented as a ROS node to monitor particular properties of the system or its parts. Based on different properties, there are different kind of observers that receive a number of parameters, test a particular condition, generate output specifying whether or not the property holds. The output of the observer is in the form of first order logical (FOL) literal which is a representation of the observed property and its status, e.g., $ok(sensor\_voltage)$, i.e., voltage on sensor is ok, or $\neg ok(sensor\_voltage)$, i.e., voltage on sensor is not ok.

In order to deal with different properties there are a number of observers namely Diagnostic Observer ($DObs$), General Observer ($GObs$), Node Observer ($NObs$), Qualitative Observers ($QObs$), Binary Qualitative Observers ($BiQObs$), Hardware Observer ($HObs$), Property Observer ($PObs$), and Interval Observer ($IObs$) as follows:

5.2.1 General Observer (GObs)

General observer (GObs) simply subscribes to a particular topic and uses its occurrences o_i. This allows the observer to monitor the communication behavior of a topic's node. For this general case, it is done by checking the frequency property of the topic. Figure 5.1 describes the template of a general observer.

	Observer Name: General Observer (GObs)
Test property:	Topic's Frequency
Parameters ($\mathcal{P}$):	$\{\tau, \delta, \sigma, \omega, \theta\}$
Description :	Monitors topic's (τ) frequency; given occurrences interval δ, deviation σ, mismatch threshold θ, and window ω
Condition (ψ):	$\begin{cases} ok(\tau), \text{if average occurrance interval on } \tau \text{ is in } [\delta \pm \sigma] \\ \neg ok(\tau), \text{otherwise.} \end{cases}$
Output (Θ):	$\{ok(\tau), \neg ok(\tau)\}$

Figure 5.1: Template for ”General Observer ($GObs$)”.

The observer computes time interval Δ_t between successive occurrences o_{t-1} and o_t. The average of all intervals within a varying sized sliding window w is calculated as μ. If the observer observes the right frequency *frq* (Equations 5.1) for a topic τ; i.e., ($|\mu - \delta| < \sigma$), it submits $ok(\tau)$, $\neg ok(\tau)$ otherwise. The frequency is measured as:

$$frq = \frac{1}{\mu} \tag{5.1}$$

where

$$\mu = \frac{\sum_{t=2}^{n}(o_t - o_{t-1})}{|seq(\tau, [t_1, t_n])| - 1} \tag{5.2}$$

where o_t is an occurrence at timestamps t and $|seq(\tau, [t_1, t_n])$ is the sequence of n occurrences collected in a window ω. Typically a window w is chosen such that the average number of occurrences for the topics with different frequencies should be same. This is achieved by the following equation 5.3:

$$\omega = c \times \frac{1}{frq} \tag{5.3}$$

where c is a constant representing desired number of average occurrences for a topic with frequency frq.

Algorithm 1: $GObs(\tau, \delta, \sigma, \omega, \theta)$

```
   input : $\tau$ ... publishing topic
   input : $\delta$ ... expected interval between occurrences
   input : $\sigma$ ... interval deviation
   input : $\omega$ ... window size
   input : $\theta$ ... mismatch threshold
   output: $\Theta$ ... Observation
 1 $W = \varnothing$
 2 $O_{mid} = O_1$
 3 $mismatch = 0$
 4 foreach new occurrence $O_i$ from $\tau$ do
 5     if $(O_{mid} - \omega/2) <= O_i <= (O_{mid} + \omega/2)$ then
 6         $W = W \cup O_i$
 7         continue
 8     end
 9     $\mu = mean(\Delta W)$
10     if $(|\mu - \delta| \leq \sigma)$ then
11         if $mismatch > 0$ then
12             $mismatch = mismatch - 1$
13         end
14     else
15         $mismatch = mismatch + 1$
16     end
17     if $mismatch > \theta$ then
18         $\Theta = \neg ok(\tau)$
19     else
20         $\Theta = ok(\tau)$
21     end
22     $O_{mid} = O_{mid+1}$
23     $pop(W[1])$
24 end
```

The functionality of the general observer (GObs) is described in Algorithm 1. The collection of the occurrences in the window of size ω occurs in Lines 4-8. Line 9 computes average interval time μ. The variable $mismatch$ keeps the number of consecutive violations observed by the observer. If the number of violations exceed the mismatch threshold θ then the observer reports an error, otherwise consistency is reported (Lines 17-21). After computations the window slides in the lines 22 and 23.

Example 5.2. *Continuing with the running example from Chapter 3, following is the list of general observers (one for each topic),* $O_g = \{GObs_{cmd\_vel}$, $GObs_{scan}$, $GObs_{imu\_data}$, $GObs_{la\_servo1\_moving}$, $GObs_{la\_servo2\_moving}$, $GObs_{map}$, $GObs_{joy}\}$ *where subscript is the name of topic. The output of the observers when every thing is properly working;* Θ=$\{ok(cmd\_vel)$, $ok(scan)$, $ok(imu\_data)$, $ok(la\_servo1\_moving)$, $ok(la\_servo2\_moving)$, $ok(map)$, $ok(joy)\}$

5.2.2 Node Observer (NObs)

Node Observer (NObs) observes the status of a node $\eta \in \mathcal{S}$ where $\mathcal{S}$ is a set of software nodes (SAM in previous Chapter). The status of a node (η) can be either *running* or *not running*. NObs interacts with ROS communicational graph in order to extract the set of currently executing *nodes*. It searches the node η in the executing nodes list. If the node η is found in the list meaning that the node is currently executing, the node observer reports $running(\eta)$, otherwise $\neg running(\eta)$ is reported. Here a tolerance level is defined by an argument denoted by θ. It can be the case that a node exists in the ROS computation graph but due to complex communication between the nodes and frequent updates of the computational graph by ROS the list may occasionally drop a node. The higher the θ the less sensitive is the observer, and vice versa. Figure 5.2 describes the node observer (NObs).

	Observer Name: Node Observer
Test Property:	Node's Status
Parameters ($\mathcal{P}$):	$\{\eta, \theta\}$
Description :	It monitors running status of a node (η).
Condition (ψ):	$\begin{cases} running(\eta), \text{if } \eta \text{ is currently executing} \\ \neg running(\eta), \text{otherwise.} \end{cases}$
Output (Θ):	$\{running(\eta), \neg running(\eta)\}$

Figure 5.2: Template for "Node Observer ($NObs$)".

The node observer (NObs) functions in the way described in Algorithm 2. It continuously checks if the ROS system is working (Line 2). The functions $capture$ and $extract\_running\_nodes$ respectively grasps the computation graph and extracts list of currently executing nodes (Lines 3-4) by using *ROS Master APIs* through XML-RPC[1]. Lines 6-10 keep

[1] http://wiki.ros.org/ROS/Master_API

Algorithm 2: $NObs(\eta, \theta)$

```
   input : $\eta$ ... node
   input : $\theta$ ... mismatch threshold
   output: $\Theta$ ... Observation
1  $mismatch = 0$
2  while $isOk(System)$ do
3      $sys\_struct = capture(System)$
4      $\mathcal{N} = extract\_running\_nodes(sys\_struct)$
5      if $\eta \in \mathcal{N}$ then
6          if $mismatch > 0$ then
7              $mismatch = mismatch - 1$
8          end
9      else
10         $mismatch = mismatch + 1$
11     end
12     if $mismatch > \theta$ then
13         $\Theta = \neg running(\eta)$
14     else
15         $\Theta = running(\eta)$
16     end
17 end
```

the number of violations. The $mismatch$ variable is decreased only if there is no violation and the number of previously detected violations is more than zero. The observer reports its output (lines 12-16) on the basis of mismatch threshold θ.

Example 5.3. *(Continued) There are nine different nodes. One node observer for each node therefore the set of node observers;* $O_n = \{NObs_{laser\_alignment\_control}$, $NObs_{joy\_node}$, $NObs_{imu\_node}$, $NObs_{laser\_alignment\_node}$, $NObs_{jaguar\_teleop\_node}$, $NObs_{hector\_mapping}$, $NObs_{jagaur\_node}$, $NObs_{hokuyo\_node}\}$ *where subscript is the name of a node. When all nodes are working the output of the observers will be;* $\Theta = \{running(laser\_alignment\_control)$, $running(joy\_node)$, $running(imu\_node)$, $running(laser\_alignment\_node)$, $running(jaguar\_teleop\_node)$, $running(hector\_mapping)$, $running(jaguar\_node)$, $running(hokuyo\_node)\}$

5.2.3 Diagnostic Observer (DObs)

Diagnostic Observer (DObs) is the connection between the existing ROS diagnostics system and the presented diagnosis and repair system. *DObs* receives diagnostic messages and generates an output based on similar rules as the existing ROS diagnostic analyzers, e.g., $temp\_cpu1 < 40.0 \Leftrightarrow ok(temp\_cpu1)$. The description of the diagnostic observer is given in Figure 5.3.

	Observer Name: Diagnostics Observer
Test Property:	Device's Status
Parameters ($\mathcal{P}$):	$\{\mathcal{D}\}$,
Description :	Diagnostics Observer checks device ($\mathcal{D}$) status on /diagnostics topic.
Condition (ψ):	$\begin{cases}\neg ok(\mathcal{D}), \text{if } \mathcal{D} \text{ is erroneous}\\ ok(\mathcal{D}), \text{otherwise.}\end{cases}$
Output (Θ):	$\{ok(\mathcal{D}), \neg ok(\mathcal{D})\}$

Figure 5.3: Template for "Diagnostic Observer ($DObs$)"

Algorithm 3: $DObs(\mathcal{D}, \tau, \theta)$

```
input  : $\mathcal{D}$ ... Device name
input  : $\tau$ ... topic /diagnostics
input  : $\theta$ ... mismatch threshold
output: $\Theta$ ... Observation
$mismatch = 0$
foreach $o_i$ on $\tau$ do
    $state = extract\_state(\mathcal{D}, o_i)$
    if $state \in \{OK, WARNNING\}$ then
        if $mismatch > 0$ then
            $mismatch = mismatch - 1$
        end
    else
        $mismatch = mismatch + 1$
    end
    if $mismatch > \theta$ then
        $\Theta = \neg ok(\mathcal{D})$
    else
        $\Theta = ok(\mathcal{D})$
    end
end
```

	Observer Name: Qualitative Observer
Test Property:	Value's Qualitative Trends
Parameters ($\mathcal{P}$):	$\{\tau, f, \beta_p, \beta_n, \omega\}$
Description :	It computes linear regression of values in windows a compares with +iv (β_p) and -ive (β_n) slopes.
Condition (ψ):	$\begin{cases} inc(v), \text{if } v \text{ increases} \\ dec(v), \text{if } v \text{ decreases} \\ cons(v), \text{otherwise.} \end{cases}$
Output (Θ):	$\{inc(v), dec(v), cons(v)\}$

Figure 5.4: Template for "Qualitative Observer ($QObs$)".

A number of hardware sensor drivers sends diagnostics data about the hardware state on a topic $/diagnostics$. The state of the hardware is published in the form of either Ok, $Warning$, or $Error$. Algorithm 3 explains how DObs works. It takes occurrence from the topic $/diagnostics$ (Line 2) and tracks the state of the specified device (lines $5-7$). The number of consecutive device's erroneous state is counted in Line 9. The observer reports $ok(device)$ if the state of the device on $/diagnostics$ topic appears to be 'Ok' or '$Warning$'. If due to some reasons the state of the device changes to the state '$Error$' the DObs reports $\neg ok(device)$ after tolerating a number of consecutive violations (lines 12 to 16).

Example 5.4. *(Continued) The driver of Hokuyo laser sensor named as* `hokuyo_node` *(hn) publishes diagnostics of hokuyo laser on the topic* `/diagnostics`*. The observer* $DObs_{hokuyo\_node}$ *will publish* $\neg$`ok(hokuyo)` *if the topic* `/diagnostics` *carries hokuyo's state '*`Error`*', and it publishes* `ok(hokuyo)` *if the state is '*`Ok`*' or '*`Warning`*'.*

5.2.4 Qualitative Observer (QObs)

Qualitative Observer (QObs) observes the abstract qualitative trend of a particular value in a message of a topic. The observer computes qualitative trend by fitting a linear regression on the values for a given time window ω.

Definition 5.6 (Qualitative trend). *A qualitative trend of a value v defines one of the three patterns the value v can have. A value can be either (1) increasing, (2) decreasing , or (3) not changing at all. Hence a value has three different qualitative trends increasing, decreasing, and constant.*

Figure 5.4 describes qualitative observer with respect to its input, parameters, condition, and output. Linear regression tries to model the relationship between two variables by fitting a linear

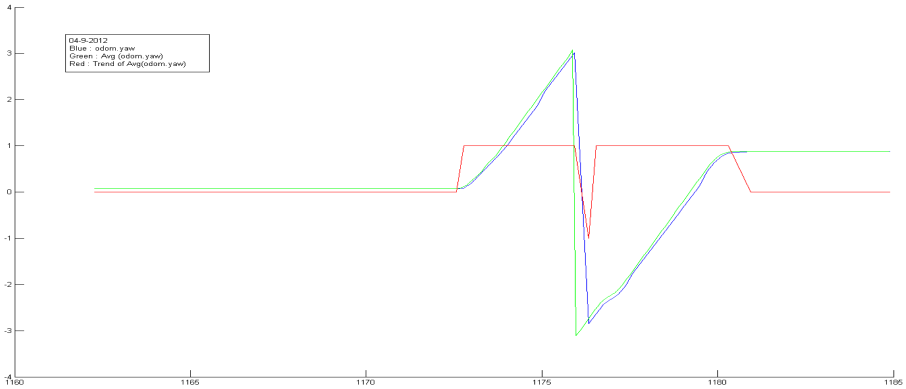


Figure 5.5: Odometry signal (Blue), its average (Green) and the qualitative trend (Red).

equation. One variable is a dependent variable while the other is independent variable. The qualitative observer collects n data values v_i and their corresponding n timestamps t_i in a sliding window ω. Considering data values v as dependent and timestamps t as independent variables the qualitative observer fits a linear regression on v and t which gives slope β of the linear line for the data values and timestamps in the window ω. If the slope β is greater than a specified positive slope β_p it means that the qualitative trend of the values is increasing, and if β is less than a specified negative slope β_n meaning qualitative trend of the values is decreasing, otherwise it computes second and third derivatives of the values in order to decide about if the value remains constant. The output of the observer is $inc(v)$, $dec(v)$ or $const(v)$ depending upon v's qualitative trend. Figure 5.5 shows the signal information (green), average (blue), and qualitative trends (red) for odometry data from the topic */pose* where trend's value 0 means constant, 1 means increasing, and -1 decreasing. For details about the abstraction process please refer to [KSW09]. Algorithm 4 presents the functionality of the qualitative observer. Lines 5-8 fill the window ω with the values v and timestamps t from the occurrences on the topic, then linear regressions are calculated. The linear regression calculations (Lines 9-11) uses formula given in the Equation 5.4. Lines 13 and 15 decides if trend of values is increasing or decreasing respectively.

$$\beta = \frac{n \times \sum_{i=1}^{n} v_i t_i - \sum_{i=1}^{n} v_i \times \sum_{i=1}^{n} t_i}{n \times \sum_{i=1}^{n} v_i^2 - (\sum_{i=1}^{n} v_i)^2} \tag{5.4}$$

Whether the trend is constant is checked in Line 18 which uses second and third derivatives. At the end in Algorithm window is moved next. It is notable that the qualitative trend is calculated for the middle value in the window ω.

Algorithm 4: $QObs(\tau, f, \beta_p, \beta_n, \omega)$

```
input  : $\tau$ ... topic
input  : $f$ ... field in topic message
input  : $\beta_p$ ... positive slope limit
input  : $\beta_n$ ... negative slope limit
input  : $\omega$ ... window size
output: $\Theta$ ... Observation
$W = \varnothing$
$O_{mid} = O_1$
$\Theta_p = \varnothing$
foreach new occurrence $O_i$ from $\tau$ do
    if $(O_{mid} - \omega/2) <= O_i <= (O_{mid} + \omega/2)$ then
        $W = W \cup O_i$
        continue
    end
    $l1 = linear\_regression(W)$
    $l2 = linear\_regression(l1)$
    $l3 = linear\_regression(l2)$
    if $l1 > \beta_p$ then
        $\Theta = inc(\tau\_f)$
    else if $l1 < \beta_n$ then
        $\Theta = dec(\tau\_f)$
    else
        if $(\beta_n < l2 < \beta_p) \wedge (\beta_n < l3 < \beta_p)$ then
            $\Theta = con(\tau\_f)$
        else
            $\Theta = \Theta_p$
        end
    end
    $O_{mid} = O_{mid+1}$
    $pop(W[0])$
    $\Theta_p = \Theta$
end
```

Example 5.5. *(Continued) The qualitative observer namely $QObs_{la\_servo1\_moving\_angle}$ for the message's field* `angle` *of the topic* `la_servo1_moving`*. It reports* `inc(la_servo1_moving_angle)` *if robot is climbing up the ramp, and* `cons(la_servo1_moving_angle)` *if the robot is moving on the flat surface.*

5.2.5 Binary Qualitative Observer (BiQObs)

Binary Qualitative Observer (BiQObs) observes matches between the abstract trends of two particular values v_1 and v_2 from the topics τ_1 and τ_2. The topics τ_1 and τ_2 can be same or different, however, the values v_1 and v_2 are always different. The observer issues $matched(v_1, v_2)$ if the abstract qualitative trend of v_1 is similar to the abstract qualitative trends of v_2 otherwise it reports $\neg matched(v_1, v_2)$. Binary qualitative observer (BObs) is described in Figure 5.6. Binary qualitative observer computes trends of two fields the same way as qualitative observer (QObs) does for one field's value. BiQObs receives two sets of the parameters for computing qualitative trends for each of the two values v_1 and v_2. The reason why we consider different window size for each of the topics is that both the topics may have different frequencies. If one window size is selected then topics with different frequencies will have different number of occurrences in the window. For example, considering the running example from Chapter 3 the topic $/imu\_data$ has publishing frequency 100Hz and topic $/scan$ has 20Hz. A window of size $100msec$ can have 10 occurrences of $/imu\_data$ and 2 occurrences of $/scan$. However, two windows of sizes $100ms$ and 500ms for the topics $/imu\_data$ and $/scan$ respectively, will have same average number of the occurrences; i.e., 10 occurrences. Figure 5.7 shows the output of a BiQObs alarming for the mismatch of qualitative trends between signals from odometry (green) and imu (blue). Algorithm 5 for BiQObs observer is self explanatory. Line 3 and 4 gets qualitative trends for each of the two fields f_1 and f_2. Lines 6-11 deal with the consecutive violations and at the end of the algorithm the output is reported (lines 12-16).

	Observer Name: Binary Qualitative Observer
Test Property:	Matches between Qualitative Trends
Parameters ($\mathcal{P}$):	$\{\tau_1, f_1, \beta_{1p}, \beta_{1n}, \omega_1, \tau_2, f_2, \beta_{2p}, \beta_{2n}, \omega_2, \theta\}$
Description :	It checks if two values have same qualitative trends.
Condition (ψ):	$matched(f_1, f_2)$, if trends of f_1 and f_2 are similar $\neg matched(f_1, f_2)$, otherwise
Output (Θ):	$\{matched(f_1, f_2), \neg matched(f_1, f_2)\}$

Figure 5.6: Template for "Binary Qualitative Observer ($BiQObs$)".

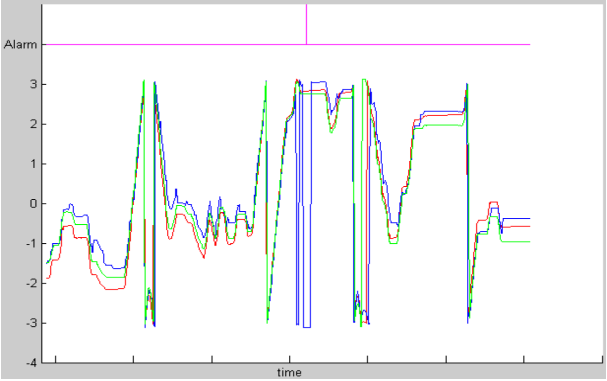


Figure 5.7: Three signals odometry pose (green), mapping pose (red), and imu_data pose (blue). BiQObs alarms when significant change is observed between imu_data and odometry.

Algorithm 5: $BiQObs(\tau_1, f_1, \beta_{1p}, \beta_{1n}, \omega_1, \tau_2, f_2, \beta_{2p}, \beta_{2n}, \omega_2, \theta)$

```
input  : $\tau_1, \tau_2$ ... topics
input  : $f_1, f_2$ ... fields in topic's messages
input  : $\beta_{1p}, \beta_{2p}$ ... positive slopes limit
input  : $\beta_{1n}, \beta_{2n}$ ... negative slopes limit
input  : $\omega_1, \omega_2$ ... two windows
input  : $\theta$ ... mismatch threshold
output: $\Theta$ ... Observation
$mismatch = 0$
while $isOk(System)$ do
    $QTrend_1 = QObs(\tau_1, f_1, \beta_{1p}, \beta_{1n}, \omega_1)$
    $QTrend_2 = QObs(\tau_2, f_2, \beta_{2p}, \beta_{2n}, \omega_2)$
    if $QTrend_1 = QTrend_2$ then
        if $mismatch > 0$ then
            $mismatch = mismatch - 1$
        end
    else
        $mismatch = mismatch + 1$
    end
    if $mismatch > \theta$ then
        $\Theta = \neg matched(\tau_1\text{-}f_1, \tau_2\text{-}f_2)$
    else
        $\Theta = matched(\tau_1\text{-}f_1, \tau_2\text{-}f_2)$
    end
end
```

Example 5.6. *(Continued) The* `BiQObs` *for the filed* `pitch` *from the topic* `imu_data`, *and the field* `angle` *from the topic* `la_servo1_moving` *gives* `matched(imu_data_x,la_servo1_moving_angle)` ***if robot moves up to the ramp.***

5.2.6 Hardware_board Observer (HObs)

Hardware_board Observer (HObs) subscribes to the measurements and status updates coming from the hardware diagnostic board to monitor the status of different power supply channels. The board is discussed in Chapter 8 in detail. The board provides information about the hardware components attached to the channels of the board. Moreover, the board provides voltage, current measurements and power states of the channels on the topic $/board\_measurments$. The hardware observer listens to the topic and provides observation $on(h)$ if board shows that hardware h is powered up, otherwise $\neg on(h)$. This allows the diagnosis and repair system to further use this information. Hardware observer (HObs) is described in Algorithm 6. Lines 3 and 4 extracts channels information to which hardware components are connected. From line 5 till 11 the status (powered up/down) of each channel is converted into observations. Function $extract\_info$ gets measurements from the topic while get_device_name and $get\_channel\_status$ extract relevant information about device and its state. The hardware observer publishes the observation with a specified delay δ (Line 6). Figure 5.8 gives an overview of the hardware observer.

Algorithm 6: $HObs(\tau, \delta)$

```
   input : $\tau$ ... /boar_measurements topic
   input : $\delta$ ... publishing delay (msecs)
   output: $\Theta$ ... Observation
 1 $\Theta \leftarrow \varnothing$
 2 foreach $o_i$ on $\tau$ do
 3     $channels = extract\_info(o_i)$
 4     foreach $ch_i \in channels$ do
 5         $device\_name = get\_device\_name(ch_i)$
 6         $status = get\_channel\_status(ch_i)$
 7         if $status == High$ then
 8             $\bar{\Theta} = on(device\_name)$
 9         else
10             $\bar{\Theta} = \neg on(device\_name)$
11         end
12         $\Theta = \Theta \cup \bar{\Theta}$
13     end
14     $\Theta = \varnothing$
15     $sleep(\delta)$
16 end
```

	Observer Name: Hardware Observer
Test Property:	Hardware Connections
Parameters ($\mathcal{P}$):	$\{\delta\}$
Description :	It outputs status of hardware component connected with diagnosis board with delay δ
Condition (ψ):	$\begin{cases} on(h), \text{if channel with h is powered up} \\ \neg on(h), \text{otherwise.} \end{cases}$
Output (Θ):	$\{on(h), \neg on(h)\}$

Figure 5.8: Template for ”Hardware Observer ($HObs$)”.

Example 5.7. *(Continued) The running example gives three hardware components namely* `jabuar_base`, `hokuyo` *sensor, and* `laser_alignment` *system. These three hardware components are connected with diagnosis board. If the board transmits the information of its connected components as powered up the hardware observer* `HObs` *with delay* $100ms$*, will publish observations* `on(jabuar_base)`, `on(hokuyo)`, `on(laser_alignment)` *with the frequency of about* 10*Hz.*

5.2.7 Property Observer (PObs)

Property Observer (PObs) supervises properties that may affect the robot’s performance. A node uses CPU power and certain amount of memory during execution. A node may start using more cpu or memory if something abnormal happens in the system. The property observer takes care of such properties. Currently we offer property observer that can take care of CPU and memory usage for a specified service. It takes maximum usage limit of the property as input. It computes the property usage, and monitors it constantly by comparing with the specified limit. If the calculated property violates the specified value the observer reports $\neg ok(property)$, otherwise reports $ok(property)$. The property observer publishes individual observations and has to be individually implemented according to the actual needs. Figure 5.9 describes property observer.

The functionality of the property observer (PObs) is step-wise given in Algorithm 7. Line 5 extracts current value v of the property (cpu or memory) for a node η. This value v is stored in a circular queue ($\mathcal{Q}$) that acts like a window in this case. The average of the values in the queue $\mathcal{Q}$ is computed in line 7. Here an issue arises that in the beginning the property observer always reports violations for n times. This issue we resolve by keeping the size of circular queue ($\mathcal{Q}$) equal to 10, and compute property value with the frequency of 10Hz, therefore, the observer fills the circular queue and violation stops in about 1sec. Lines 9-14 count the violations occurred in a row. The output is generated in lines 15-19.

	Observer Name: Property Observer
Test Property:	ρ (Memory or CPU usage)
Parameters ($\mathcal{P}$):	$\{\eta, \rho, v_{max}, \omega, \theta\}$
Description :	It observers the cpu or memory usage of a service
Condition (ψ):	$\begin{cases} ok(\eta, \rho), \text{if } \rho \text{ for } \eta \text{ is under limits} \\ \neg ok(\eta, \rho), \text{otherwise.} \end{cases}$
Output (Θ):	$\{ok(\eta, \rho), \neg ok(\eta, \rho)\}$

Figure 5.9: Template for ”Property Observer ($PObs$)”.

Algorithm 7: $PObs(\eta, \rho, v_{max}, \mathcal{Q}, \theta)$

```
input : $\eta$ ... node
input : $\rho$ ... node's property CPU/Mem
input : $v_{max}$ ... maximum limit
input : $\mathcal{Q}$ ... Circular Queu of size $n$
input : $\theta$ ... mismatch threshold
output: $\Theta$ ... Observation
$\mathcal{Q} = zeros(n)$
$mismatch = 0$
while $isRunning(\eta)$ do
    $pop(\mathcal{Q}[0])$
    $v = extract\_property\_value(System, < \eta, \rho >)$
    $append(\mathcal{Q}, v)$
    $\mu = mean(\mathcal{Q})$
    if $\mu \leq v_{max}$ then
        if $mismatch > 0$ then
            $mismatch = mismatch - 1$
        end
    else
        $mismatch = mismatch + 1$
    end
    if $mismatch > \theta$ then
        $\Theta = \neg ok(\eta, \rho)$
    else
        $\Theta = ok(\eta, \rho)$
    end
end
```

Example 5.8. *(Continued) Property observer $PObs_{hector\_mapping}$ for monitoring memory usage of the node* `hector_mapping` *(hm). If the node* `hector_mapping` *uses memory under limits then $PObs_{hector\_mapping}$ will generate observation* `ok(hector_mapping,mem)`*. If the observer monitors violations up to a certain number (θ) of times it reports* $\neg$`ok(hector_mapping,mem)`*.*

Algorithm 8: $IObs(\tau_1, f_1, \tau_2, f_2, \Delta, \theta)$

```
input  : $\tau_1$ ... first topic
input  : $\tau_2$ ... second topic
input  : $f_1$ ... field of topic $\tau_1$
input  : $f_2$ ... field of topic $\tau_2$
input  : $\theta$ ... mismatch threshold
output: $\Theta$ ... Observation
$mismatch = 0$
while $isOk(System)$ do
    $v_1 = value(\tau_1, f_1)$
    $v_2 = value(\tau_2, f_2)$
    if $|v_1 - v_2| \leq \Delta$ then
        if $mismatch > 0$ then
            $mismatch = mismatch - 1$
        end
    else
        $mismatch = mismatch + 1$
    end
    if $mismatch > \theta$ then
        $\Theta = \neg equal(\tau_1\text{-}f_1, \tau_2\text{-}f_2)$
    else
        $\Theta = equal(\tau_1\text{-}f_1, \tau_2\text{-}f_2)$
    end
end
```

5.2.8 Interval Observer (IObs)

Interval Observer (IObs) monitors the interval difference between two values v_1 and v_2. Like $BiQObs$ these two values can be from one topic or two different topics τ_1 and τ_2. In a robotics system it can be the case that two or more values have the same interval. For example in a working condition the interval between pitch measurement from imu_node and servo $angle$ measurement from $laser\_alignment\_control$ node is always same (running example). If something goes wrong either with IMU or servo sensor the interval is changed. The interval observer (IObs) monitors such intervals between two values. It reports $equal(v_1, v_2)$ if $|v_1 - v_2|$ does not exceed a certain threshold for the values v_1 and v_2, otherwise $\neg equal(v_1, v_2)$. Figure 5.10 describes the template of the interval observer (IObs). The functionality of IObs is given in Algorithm 8. Lines 3 and 4 takes field's values from the topics. The interval is computed and checked against a specified limit (Δ). Lines 6-10 keep the number of consecutive violations. Finally lines 12-16 give output either $equal(v_1, v_2)$ or $\neg equal(v_1, v_2)$ depending upon number of violations and mismatch threshold (θ).

	Observer Name: Interval Observer
Test Property:	Interval between two values v_1 and v_2
Parameters ($\mathcal{P}$):	$\{\tau_1, f_1, \tau_2, f_2, \Delta, \theta\}$
Description :	It checks the interval between two values.
Condition (ψ):	$\begin{cases} equal(v_1, v_2), if(\lvert v_1 - v_2\rvert \leq \Delta) \\ \neg equal(v_1, v_2), otherwise \end{cases}$
Output (Θ):	$\{equal(v_1, v_2), \neg equal(v_1, v_2)\}$

Figure 5.10: Template for "Interval Observer ($IObs$)".

Example 5.9. *(Continued) The interval observer* `IOBs` *for monitoring interval between the values of* `pitch` *from the topic* `imu_data`*, and the servo* `angle` *field from the topic* `la_servo_moving`*. The observer outputs* $equal(imu\_data\_pitch, la\_servo\_moving\_angle)$ *if the interval does not exceed certain threshold, otherwise it reports* $\neg equal(imu\_data\_pitch, la\_servo\_moving\_angle)$.

5.2.9 Difference between IObs and BiQObs

The interval observer (IObs) seems to be quite similar to the binary qualitative observer (BiQObs) in its behavior because as long as the qualitative trends remain same the interval between the signals also remains same. However, there is a slight difference between both of them as depicted in Figure 5.11. The figure shows two signals green and red, both have the same trends and interval until time t. At t the trend of the green signal is "decreasing" for a very short and negligible duration of time, and then it becomes "increasing" making qualitative trend same as other signal. However, the interval gets changed after time t and continue with the same change. In such scenario the qualitative observer (pink) can ignore such slight change in the qualitative trend because of its sliding windows for noise, but the interval observer (blue) finds the inconsistency in the interval between both the signals after a very short period of time, and reports a fault.

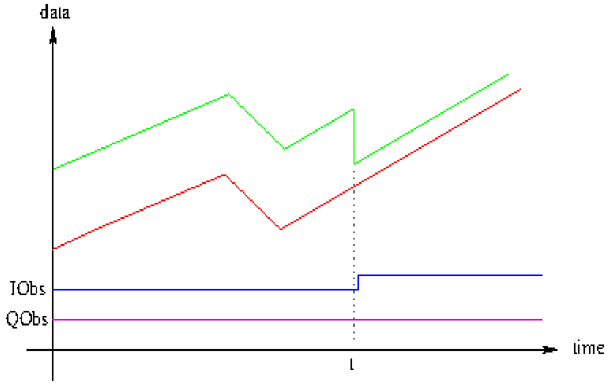


Figure 5.11: Two signals having same qualitative trends but different intervals.

In the section above a number of observers, namely diagnostic observer (DObs), general observer (GObs), node observer (NObs), qualitative observers (QObs), binary qualitative observers (BiQObs), hardware observer (HObs), and property observer (PObs) are discussed in detail. Each of these observers is implemented as a `ROS` node which publishes the output on the topic `/observations`. The observations are predicates of the form $s(c)$ where s represents the status and the argument c is the component. Table 5.1 lists all the observers and their possible output observations. A collection of the observers nominated for a robotic system makes a monitoring system for it. The observations from the observer reflect the behaviour (called observed behavior) of the system being monitored. The presented work follows *model-based* reasoning that uses correct behavior of a system and fault is detected if there is any contradiction between the observed and the correct behaviour. The diagnosis_engine (next Chapter) takes the observed behavior in the form of observations coming from the observers on the topic `/observations` and derives root cause of the discrepancy between the model (predicted behavior) and observation (observed behavior). The `repair_engine` (Chapter 7) also needs observation in order to generate a problem description for computing a plan for the repair actions. The main objective of the monitoring system presented here is to provide observed behavior for the diagnosis process.

Observers (Obs)	Possible observations
$GeneralObserver(GObs)$	$ok(topic), \neg ok(topic)$
$NodeObserver(NObs)$	$running(node), \neg running(node)$
$DiagnosticObserver(DObs)$	$ok(device), \neg ok(device)$
$HardwareObserver(HObs$	$on(channel), \neg on(channel)$
$QualitativeObserver(QObs)$	$inc(val), dec(val), cons(val)$
$BinaryQualitativeObserver(BiQObs)$	$matched(val_1, val_2), \neg matched(val_1, val_2)$
$PropertyObserver(PObs)$	$ok(node, prop), \neg ok(node, prop)$
$IntervalObserver(IObs)$	$equal(val_1, val_2), \neg equal(val_1, val_2)$

Table 5.1: Possible observations for different observers

Chapter 6

Diagnosis

This chapter provides insights into diagnosis model and robot behavioral model ($\mathcal{RBM}$). It also talks about how diagnosis process exploits the diagnosis model in order to localize the faults. The diagnosis presented in this chapter has been partially published in the contribution [ZSM$^+$13].

6.1 Introduction

In this dissertation we follow one of the model-based diagnosis's approaches called *consistency-based* diagnosis. In order to diagnose a misbehavior in a system's components the consistency-based approach requires two things; (1) the expected correct behavior of the system and (2) the observations from the system's components at run-time. It defines a diagnosis as a set of assumptions about a system component's abnormal behavior such that the observations of one component's misbehavior are consistent with the assumptions that all the other components are acting correctly [PW03, dKW87]. The correct behavior of a system is represented in the form of component-oriented system model. The consistency-based diagnosis uses this system model as well as the assumptions that all the components of the system are acting correctly to derive root cause of a fault. If there is an observation which leads to a contradiction with the assumptions that the component that resolves this contradiction is diagnosed as faulty. In this work by the term "model-based diagnosis" we mean "consistency-based diagnosis" and vice versa.

A formalization of the concrete behavior of a robotics system for the successful model-based diagnosis is to describe knowledge of the problem domain in such a way that it should be easily computable from the machine. This can be achieved in different ways, e.g., logic-based model, or constraints-based model. We consider the logical representation for modeling the correct behavior of the problem domain. One of the reasons for selecting the logic-based model for the behavioral representation is that the diagnosis engine we use supports propositions and logical Horn clauses. Figure 6.1 depicts the process involved in the model-based diagnosis which along

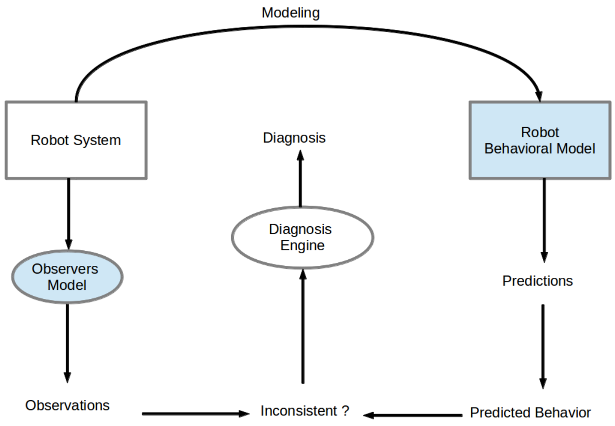


Figure 6.1: Consistency-based diagnosis: Blue components are the models. The Observers model is a set of observers to monitor the robotics system

with the observations uses two important artifacts, namely behavioral model and diagnosis engine. The conversion of robot architectural model ($\mathcal{RAM}$) discussed in Chapter 4 to the robot behavioral model ($\mathcal{RBM}$) is presented in the following sections.

6.2 Diagnosis Model

In the context of our diagnosis system the diagnosis model is a combination of two structures: (1) a logical model also called robot behavioral model ($\mathcal{RBM}$) and (2) a set of the observers called the observers model ($\mathcal{OM}$).

Definition 6.1 (diagnosis model). *A diagnosis model (DM) is a tuple $\langle \mathcal{RBM}, \mathcal{OM} \rangle$ such that:*

- $\mathcal{RBM}$: *a model comprising a set of the Horn clauses to represent a robot's behavior.*

- $\mathcal{OM}$: *a model comprising a set of observers for monitoring a robot's behavior.*

The robot behavioral model ($\mathcal{RBM}$) describes the correct behavior of the robotics system in terms of its individual components behavior. It contains logical rules that collectively reflect the correct behavior of a robotics system. The observers model ($\mathcal{OM}$) contains a number of observers in order to extract observed behavior of a system at run-time. Following sections discuss both of these structures in detail.

6.2.1 Robot Behavioral Model ($\mathcal{RBM}$)

The robot behavioral model ($\mathcal{RBM}$) as also briefly discussed in Chapter 4, is an abstract model that contains a set of logical rules in order to describe the robotics system's correct behavior. It contains logical sentences (Horn clauses) to get efficient logical deduction and reasoning for deriving a diagnosis. This way of representation also makes the model compatible with the diagnosis engine used in the presented diagnosis system. The diagnosis engine supports propositions and Horn-clauses based on the concepts presented in the contribution [Rei87] where the term "system description (SD)" is "behavioral model" in our case. In the context of the presented work, the behavioral model uses the set of Horn clauses to model individual system component's behavior using logical literal $\neg AB(c)$ ("not ABnormal component c"). For instance, a logical model for the correct behavior of a Bulb-system can be described by the following logical rule:

$$\neg AB(Bulb) \land high(voltage) \land on(switch) \rightarrow light(Bulb)$$

which states that, if we assume the Bulb is not faulty, and if switch is turned "On" and voltage in wire stays "$High$" the Bulb must always emit light. The above rule in the form of a

Algorithm 9: $RBModel(\{\mathcal{S}, \mathcal{H}, \mathcal{HN}, \mathcal{SH}, \mathcal{SN}, \mathcal{N}\}, \mathcal{T}, \mathcal{F}, \{\delta_{out}, \delta_{in}, \delta_{aff}, \delta_{hw}, \delta_{fld}, \delta_{rel}\})$

```
input  : $RAM$ ... robot architectural model
output: $\mathcal{M}$ ... set of Horn clauses
$\mathcal{M} = \varnothing$
foreach $h \in \mathcal{SH}$ do
    $\mathcal{M} = \mathcal{M} \cup \{\neg AB(h) \rightarrow on(h)\}$
end
foreach $s \in \mathcal{SN}, \tau \in \delta_{out}(s)$ do
    $\mathcal{M} = \mathcal{M} \cup \{\bigwedge_{h \in \delta_{hw}(s)} \neg AB(h) \land \neg AB(s) \rightarrow ok(t)\}$
end
foreach $h \in \mathcal{HN}, \tau \in \delta_{out}(s)$ do
    $\mathcal{M} = \mathcal{M} \cup \{\neg AB(h) \rightarrow ok(t)\}$
end
foreach $n \in \mathcal{N}, \tau \in \delta_{out}(n)$ do
    $\mathcal{M} = \mathcal{M} \cup \{\neg AB(n) \land \bigwedge_{i \in \delta_{in}(n,t)} ok(i) \rightarrow ok(t)\}$
end
foreach $n \in \mathcal{S}$ do
    $\mathcal{M} = \mathcal{M} \cup \{\neg AB(n) \rightarrow running(n)\}$
end
foreach $n_1, n_2 \in \mathcal{S}, \tau_1 \in \delta_{out}(n_1), \tau_2 \in \delta_{out}(n_2), f_1 \in \delta_{fld}(\tau_1), f_2 \in \delta_{fld}(\tau_2), \delta_{rel}(f_1, f_2), f_1 \neq f_2$ do
    $\mathcal{M} = \mathcal{M} \cup \{\neg AB(n_1) \land \neg AB(n_2) \rightarrow matched(f_1, f_2)\}$
end
return $\mathcal{M}$
```

logical clause is exactly an Horn clause as $\neg AB$ can be represented by a propositional atom. The concrete behavior of a robotics system can be determined by the behavior of its individual components. A concrete behavior formalization needs a transformation of the whole system into a set of logical rules describing individual component's behavior and hence the whole system's behavior. We follow similar model formalization approach as in [SW05].

The robot behavioral model ($\mathcal{RBM}$) presents similar logical rules in order to model the behavior of individual components. Each of the rules uses predicates 'AB' and 'ok' for respectively a abnormal behavior and a correct condition. Algorithm 9 describes the formalization of the robot behavioral model ($\mathcal{RBM}$) from the robot architectural model ($\mathcal{RAM}$). Lines 2-4 model rules for switchable hardware components. We consider an hardware component as a switchable hardware which is connected with the diagnosis hardware board (discussed in Chapter 8). The software nodes of switchable hardware are modelled in Lines 5-7. Lines from 8-10 model the software nodes for the hardware other than switchable. The software nodes with the outgoing topics are considered in Lines 11-13 for the modeling. From Line 14 till 16 all the software nodes are modeled. At the end, the fields which are related to each other on the basis of a qualitative trend of their values, are modeled with their publishing nodes.

Example 6.1. *(Continued) The $\mathcal{RBM}$ from the $\mathcal{RAM}$ for the running example, given in Chapter 4:*

$\neg AB(jaguar) \rightarrow on(jaguar)$
$\neg AB(hokuyo) \rightarrow on(hokuyo)$
$\neg AB(laser\_alignment) \rightarrow on(laser\_alignment)$
$\neg AB(jaguar) \wedge \neg AB(jaguar\_node) \rightarrow ok(pose)$
$\neg AB(hokuyo) \wedge \neg AB(hokuyo\_node) \rightarrow ok(scan)$
$\neg AB(imu\_node) \rightarrow ok(imu\_data)$
$\neg AB(joy\_node) \rightarrow ok(joy)$
$\neg AB(joy\_teleop\_node) \rightarrow ok(joy)$
$\neg AB(hector\_mapping) \rightarrow ok(scan)$
$\neg AB(laser\_alignment\_control) \rightarrow ok(la\_servo1\_moving)$
$\neg AB(laser\_alignment\_control) \rightarrow ok(la\_servo2\_moving)$
$\neg AB(jaguar\_node) \rightarrow running(jaguar\_node)$
$\neg AB(hokuyo\_node) \rightarrow running(hokuyo\_node)$
$\neg AB(imu\_node) \rightarrow running(imu\_node)$
$\neg AB(laser\_alignment\_node) \rightarrow running(laser\_alignment\_node)$
$\neg AB(laser\_alignment\_control) \rightarrow running(laser\_alignment\_control)$
$\neg AB(joy\_teleop\_node) \rightarrow running(joy\_teleop\_node)$
$\neg AB(joy\_node) \rightarrow running(joy\_node)$
$\neg AB(hector\_mapping) \rightarrow running(hectla\_servo1\_movingor\_mapping)$

$\neg AB(rviz) \rightarrow running(rviz)$
$\neg AB(imu\_node) \wedge \neg AB(jaguar\_node) \rightarrow matched(imu\_data.yaw, pose.yaw)$
$\neg AB(jaguar\_node) \rightarrow matched(pose.twist.angular.z, pose.yaw)$
$\neg AB(imu\_node) \wedge \neg AB(laser\_alignment\_control) \rightarrow matched(i\_d.pitch, l\_s\_m.angle)$
$\neg AB(imu\_node) \wedge \neg AB(laser\_alignment\_control) \rightarrow matched(i\_d.yaw, l\_s\_m.yaw)$

where i_d *and* l_s_m *denote respectively* imu_data *and* la_servo1_moving *in last two rules.*

6.2.2 Observers Model ($\mathcal{OM}$)

In addition to the robot behavioral model ($\mathcal{RBM}$) the diagnosis process also requires the observed behavior of the robotics system for localizing a fault. The observations about a system at run-time are provided by a set of the observers. The observers according to the size and the complexity of a robotics system are initiated in order to continuously monitor the robotics system's components, and provide the logical observations to the diagnosis engine for locating a faulty component if any. We call this set of observers for observing the robotics system an observers model ($\mathcal{OM}$).

Definition 6.2 (observer model). *The observer model is a set of the observers* $\mathcal{O} = \{o_1, o_2, ..., o_n\}$ *for a robotics system, that collectively provides the observed behavior of the robotics system at run-time.*

The generation of the observers model ($\mathcal{OM}$) is discussed in Chapter 9 in learning phase because most of the observers require learned parameters that need to be learned, for example, a general observer (GObs) is instantiated for a topic $\tau \in \mathcal{T}$ if the topic is regular. A topic is considered regular if the fraction of the mean (μ) and the standard deviation (σ) of its occurrences interval time (Δ_t) is above a certain threshold (α). The parameters μ and σ are learned during learning process. Similarly a binary qualitative observer (BiQObs) for two signals is instantiated if the two signals are correlated with each other in terms of their qualitative trends which are also learned during the learning phase. For each node one node observer (NObs) and two property observers (PObs), i.e., one for cpu usage and other for memory usage, are instantiated. The cpu and memory values are also learned during the learning phase.

The set of general observers is denoted by $\mathcal{O}_g$. Likewise $\mathcal{O}_n$, $\mathcal{O}_p$ $\mathcal{O}_{biq}$, and $\mathcal{O}_i$ are the set of node, property, binary qualitative, and interval observers respectively. The diagnostics and hardware observers are instantiated without learning. The observer diagnostics ($DObs$) is instantiated if there exists the "/diagnostics" topic, and hardware observer ($HObs$) is instantiated when at least one component is connected with the hardware diagnostics board which broadcasts its information on the topic "/board_measurements".

Example 6.2. *(Continued) The running example given in Chapter 3 includes eight different nodes, therefore, there will be* 8 *node observers* $(|\mathcal{O}_n| = 8)$*, i.e.,* $\mathcal{O}_n = \{$NOb(jn),NOb(hn),NOb(in),NOb(lan),NOb(lac),NOb(jtn), NObs(jyn),NOb(hm),NObs(rviz)$\}$ *and* 16 *property observers* $(|\mathcal{O}_p| = 16)$*, i.e.,* $\mathcal{O}_p = \{$POb(jn,cpu),POb(jn,mem),POb(hn,cpu),POb(hn,mem),POb(in,cpu), POb(in,mem),POb(lan,cpu),POb(lan,mem),POb(lac,cpu),POb(lac,mem), POb(jtn,cpu),POb(jtn,mem), PObs(jyn,cpu),POb(jyn,mem),POb(hm,cpu), POb(hm,mem)$\}$ *and for five regular topics* $(|\mathcal{O}_g| = 5)$*, i.e.,* $\mathcal{O}_g = \{$GObs(pose), GObs(imu_data), GObs(scan), GObs(la_servo1_moving), GObs(la_servo2_moving)$\}$*. As hokuyo_node is publisher to the topic "/diagnostics", therefore, one diagnostics observer* $\mathcal{O}_d = \{$DObs(hokuyo)$\}$*, there are three hardware components attached to the diagnostic board so there will also be hardware observer* $\mathcal{O}_h = \{$HObs(/board_measurements)$\}$ *and there can be one binary qualitative observer* $\mathcal{O}_{biq} = \{$BiQObs(imu_data.pitch,la_servo1_moving.angle)$\}$ *and one interval observer* $\mathcal{O}_i = \{$IQObs(imu_data.pitch,la_servo1_moving.angle)$\}$ *making total number of observers* $|\mathcal{O} = 32|$*. The terms* jn, hn, in, lan, lac, jtn, jyn, hm *denote respectively* jaguar_node, hokuyo_node, imu_node, laser_alignment_node. laser_alignment_control, jaguar_telelop_node, joy_node, *and* hector_mapping*.*

Once the observers are instantiated and invoked they start continuously providing observations about robotics system's components on a ROS topic "/observations". In the implementation we launch a file to invoke the whole bunch of the observers. Following is a simple launch file describing a node, a property, and a general observer:

```
<node pkg="tug_ist_diagnosis_observers" type="NObs.py" name="hmNObs">
        <param name="node" value="hector_mapping" />
</node>
<node pkg="tug_ist_diagnosis_observers" type="PObs.py" name="hmCPObs">
        <param name="node" value="hector_mapping" />
        <param name="property" value="cpu" />
        <param name="max_val" value="14.851168" />
        <param name="mismatch_th" value="5" />
</node>
<node pkg="tug_ist_diagnosis_observers" type="GObs.py" name="idGObs">
        <param name="topic" value="imu_data" />
        <param name="delta" value="0.0100006230955" />
        <param name="dev" value="0.0323852059052" />
        <param name="ws" value="0.100006230955" />
        <param name="mismatch_th" value="5" />
</node>
```

Having obtained both the structures for the diagnosis model, i.e., the robot behavioral model ($\mathcal{RBM}$) and the observers model ($\mathcal{OM}$), it fulfills the requirements of the mode-based diagnosis process by providing respectively as system description (SD) and the observations (Θ) in order to compute the diagnosis (Δ).

6.3 Diagnosis Model Server

The diagnosis process requires the robot behavioral model ($\mathcal{RBM}$) at run-time for computing a diagnosis if any fault occurs. It is diagnosis model server (*DMS*) which acts like an action server and provides the required robot behavioral model when requested by the diagnosis engine at run-time. The diagnosis engine is used for computing the diagnosis in our work. It considers a model represented with the logical Horn clauses and propositions as presented in [PW03]. The propositions are for dealing with observations and Horn clauses to represent correct behavior of a robotics system. Fulfilling these characteristics of the diagnosis model we need to convert the rules in $\mathcal{RBM}$ model into Horn clauses. Consider the following logical rule:

$$\neg AB(hokuyo\_node) \wedge \neg AB(hokuyo) \rightarrow ok(scan)$$

This logical rule states that when hokuyo laser sensor and its driver (hokuyo_node) are assumed to be working correctly then the topic $/scan$ must also be publishing the laser scans properly. The robot behavioral model contains such logical rules, therefore, in order to make these rules compatible with the diagnosis engine, i.e., these rules need to be formed in such a way that it should be a Horn clause. First issue is to deal with the negation (¬) symbol for describing normality in the component's behavior. This is done by introducing a proposition `NAB` meaning "NotABnormal", e.g., the normal and correct behavior of a component 'c' can be represented by the proposition '`NAB(c)`'. Hence, the above logical rule becomes a following Horn clause compatible for the diagnosis process:

$$NAB(hokuyo\_node) \wedge NAB(hokuyo) \rightarrow ok(scan)$$

To make it more flexible the structure of the behavioral model provided by the model server contains sections for symbolizing normal (`NAB`) and abnormal (`AB`) behavior for components. The second issue to deal with the negation (¬) symbol for the observation, e.g., $\neg ok(scan)$, and to make it compatible for the diagnosis process. This issue is resolved by introducing a section for "negative prefix" which can allow replacing $\neg ok$ with something like nok. However, for the sake of understandability we adapt the notation 'not_ok' instead of '$\neg ok$' where prefix '$not\_$' is called a negative prefix.

The diagnosis model server is a ROS-based action server which follows a strictly typed message structure for the behavioral model. In our work we store all the logical Horn clauses and propositions from robot behavioral model ($\mathcal{RBM}$) in a structure in the form of a YAML file (similar to XML) that contains five sections: (1) a string that denotes the proposition to represent the AB predicate, (2) the same for ¬ AB, (3) a string for a prefix denoting a negative literal, (4) a set of all propositions, (5) a set of clauses that defines the correct behavior of the robotics system. We have to describe the model this way as the currently used diagnosis engine only supports propositions and Horn clauses. Following is a simple behavioral model in the form of sectioned structure:

```
ab: "AB"
nab: "NAB"
neg_prefix: "not_"
props:
  - running(imu_node)
  - running(laser_alignment_control)
  - ok(imu_node,cpu)
  - ok(imu_node,mem)
  - ok(laser_alignment_control,cpu)
  - ok(laser_alignment_control,mem)
  - ok(imu_data)
  - ok(la_servo_moving)
  - matched(imu_data,pitch,la_servo_moving,angle)
rules:
  - NAB(laser_alignment_control) -> running(laser_alignment_control)
  - NAB(imu_node) -> running(imu_node)
  - NAB(laser_alignment_control) -> ok(la_sero_moving)
  - NAB(imu_node) -> ok(imu_data)
  - NAB(imu_node) -> ok(imu_node,cpu)
  - NAB(imu_node) -> ok(imu_node,mem)
  - NAB(laser_alignment_control) -> ok(laser_alignment_control,cpu)
  - NAB(laser_alignment_control) -> ok(laser_alignment_control,mem)
  - NAB(l_a_c),NAB(i_n),ok(i_d),ok(l_s_m) -> matched(i_d,p,l_s_m,a)
  - AB(imu_node) -> not_ok(imu_data)
```

The last rule uses short names for the nodes and topics that are already used in the previous rules.

The diagnosis model server integrates the following features in the diagnosis and repair system:

1. It acts as a standalone ROS action server, and hence modularizes the system architecture.
2. A request for the model can be sent to the model server and can be cancelled at run-time .
3. It permits changes in the the model at run-time.

6.4 Diagnosis Computation

The diagnosis computation process in our diagnosis and repair system is a task of deriving root cause of a fault if any contradiction between an observed and a behavioral model rises. It follows the principles of model based diagnosis presented in [Rei87]. It takes as inputs the observations Θ coming from the observers, and the abstract robot behavioral model (RBM) from the diagnosis model server. If there is any contradiction between observations and the behavioral model a fault is detected. The diagnosis engine localizes the fault by deriving the root causes (faulty components) for the fault and publishes the faulty components as a diagnosis on a ROS topic named ”`/diagnosis`”. The ”`/diagnosis`” topic not only provides the diagnosis but it also connects the diagnosis engine (DE) with the repair engine (RE) discussed in the next chapter. The kernel of the diagnosis module concurrently executes two main functionalities: the observation collection and the diagnosis. While the former continuously performs the task of making a consistent set of the logical observations, i.e., `OBS`, the latter performs localization of the faults using diagnosis engine at logical level.

6.4.1 Observation Collection

The observations are required for the diagnosis computation in order to find discrepancies between the observations and the robot behavioral model ($\mathcal{RBM}$). Therefore, the observations coming from the observers are collected into a consistent set of observations Θ. Each single observation θ_t that occurs at time t on the topic ”`/observations`” is integrated into an observation set as described in following process:
CollectObservation:

1. set a global set $\Theta_{OBS} = \varnothing$
2. subscribe to the topic (τ) ”$/observations$”
3. subscriber get each new observation θ from τ in the function `callback` which merges θ in the set Θ_{OBS} of the observations.

The process of collecting observations initializes the set of the observations Θ_{OBS} as empty set. At step 2 it subscribes to the topic ”`/observations`” to obtain the observations from

Algorithm 10: $callback(\theta)$

```
    input : $\theta$ ... new observation
 1  if $\theta \notin \Theta_{OBS}$ then
 2      if $\theta == \neg L$ then
 3          if $L \in \Theta_{OBS}$ then
 4              $\Theta_{OBS} = \Theta_{OBS} \setminus L$
 5          end
 6      else
 7          if $\neg L \in \Theta_{OBS}$ then
 8              $\Theta_{OBS} = \Theta_{OBS} \setminus \neg L$
 9          end
10      end
11      $\Theta_{OBS} = \Theta_{OBS} \cup \theta$
12  end
```

the observers. The subscriber can receive each new individual observation θ from the topic in the function `callback` (Algorithm 10). If the observation is already in the set Θ_{OBS} the `callback` does nothing (Line 1) otherwise it removes the θ's counterpart negative observation literal $\neg L$ (Lines 2-6) or positive literal L (Lines 7-10) from the set Θ_{OBS}, and appends newly coming observation θ (Line 11).

Example 6.3. *(Continued) Assuming the robotics system of the running example from Chapter 3 we start diagnosis module, the set of observations is initially empty $\Theta_{OBS} = \emptyset$. Now we start observer $GObs_{scan}$ and the observation $ok(scan)$ appears. The diagnosis engine searches the Θ_{OBS} for the observation $ok(scan)$. As Θ_{OBS} is empty therefore the observation is added, i.e., $\Theta_{OBS} = \{ok(pose)\}$. The same thing happens when first observations from $NObs_{jaguar\_node}$ and $PObs_{jyn_{cpu}}$ appear, so $\Theta_{OBS} = \{ok(scan), running(jaguar\_node), ok(jyn, cpu)\}$. Now the observations $ok(scan)$, $running(jaguar\_node)$, and $ok(jyn, cpu)$ appear one after another but they are not added because Θ_{OBS} already contains them. Suppose at this point of time something happens and the node $jaguar\_node$ stops. The observer $NObs_{jaguar\_node}$ senses it and changes its observation from $running(jaguar\_node)$ to $\neg running(jaguar\_node)$ which comes to the diagnosis engine. The diagnosis engine searches Θ_{OBS} for $\neg running(jaguar\_node)$ but does not find it. Before adding $\neg running(jaguar\_node)$, it also searches for its counter observation $running(jaguar\_node)$, it finds and removes it, and appends $\neg running(jaguar\_node)$ in the set making it as:*

$$\Theta_{OBS} = \{ok(scan), ok(jyn, cpu), \neg running(jaguar\_node)\}$$

the process continues.

6.4.2 Diagnosis Engine

In order to compute diagnosis we use an open-source Java-based implementation of the approach [PW03]. The diagnosis engine computes a diagnosis (Δ) which is set of components that are faulty (Δ_{bad}) and components that are working properly (Δ_{good}). Given a set of logical observations Θ_{OBS}, a set of components $\mathcal{COMPS}$ and the robot behavioral model $\mathcal{RBM}$ (system description) the diagnosis (Δ) computed as a minimal set $\Delta \subseteq \mathcal{COMPS}$ such that:

$$\mathcal{RBM} \cup \Theta_{OBS} \cup \{AB(c)|c \in \Delta\} \cup \{\neg AB(c)|c \in \mathcal{COMPS} - \Delta\}$$

is consistent.

The predicate $\neg AB(m)$ denotes that the module m is working properly whereas $AB(m)$ denotes that the module m shows a faulty behavior. The approach uses an abstract model (diagnosis model (DM) in our case) that defines correct behavior and current observations of the system. A fault is detected if an outcome of the model and the observation lead to a contradiction. Using a Hitting set algorithm the approach calculates the Hitting sets in order to compute the diagnoses that resolve the contradiction, i.e., explain the misbehavior. The engine locates that component or set of components that are the root cause for the contradiction. The results of the diagnosis engine are published on the */diagnosis* topic. As described earlier the diagnosis message may comprise several diagnoses build up by the set of working (Δ_{good}) and the faulty (Δ_{bad}) components. The union of both sets is equal to the set of all the system components ($\Delta_{good} \cup \Delta_{bad} = \mathcal{COMPS}$) for each individual diagnosis. Apart from this it also generates diagnostic messages compatible for the ROS diagnostics stack.

Example 6.4. *(Continued) Suppose the diagnosis process gets the observations set* $\Theta_{OBS} = \{running(jn), .., running(rviz), ok(scan), ..ok(map), .., ok(jn, cpu), .., ok(rviz, mem)\}$. *The assumption* $\mathcal{A} = \{\neg AB(jn), \neg AB(hn), .., \neg AB(rviz)\}$ *that all components are working correctly is considered then the robot behavioral model ($\mathcal{RBM}$) defined in the previous section is the system description. The* $\mathcal{RBM} \cup \Theta_{OBS} \cup \mathcal{A}$ *is consistent because there is no conflict set and the diagnosis engine provides only a minimal diagnosis being empty set* $\{\}$ *for faulty components. The diagnosis engine publishes diagnosis as:*

$$\Delta = \{\Delta_{bad} = \{\}, \Delta_{good} = \{jn, hn, in, lan, jtn, hm, lac, jyn, rviz\}\}$$

Now suppose the node $jaguar\_node$ *(jn) stops because of some unknown reasons, and this updates the observations set as* $\Theta_{OBS} = \{\neg running(jn), .., running(rviz), ..., \neg ok(jn, cpu), \neg ok(jn, mem), .., ok(rviz, mem)\}$ *making* $\mathcal{RBM} \cup \Theta \cup \mathcal{A}$ *inconsistent . The minimal conflict set calculated is* $F = \{jn\}$ *because:*

$$\mathcal{RBM} \cup \Theta_{OBS} \cup \{\neg AB(jn)\} = \perp$$

so the Hitting set algorithm of diagnosis engine provides minimal diagnosis set $\{jaguar\_node\}$ *and the diagnosis module's output:*

$$\Delta = \{\Delta_{bad} = \{jn\}, \Delta_{good} = \{hn, in, lan, jtn, hm, lac, jyn, rviz\}\}$$

We have implemented the diagnosis computation process's kernel in C++ that communicates with the java-based diagnosis engine over a TCP connection, relying on a text-based communication protocol. It obtains the diagnosis behavioral model at run-time from the diagnosis model server.

Chapter 7

Repair

This chapter talks about the repair engine and how it repairs a faulty component both hardware and software at run-time. Moreover, it also discusses how the repair engine uses planner to acquire a plan for the repair actions. Parts of the work presented in this chapter have been published in [ZSM$^+$13].

7.1 Repair Engine

After having diagnosed the faulty components via model-based diagnosis engine (previous Chapter), the diagnosis and repair system uses a planner-based repair engine in order to repair the faults. The repair engine takes the current observations (Θ) and the diagnoses (Δ) as input. The observation comes from the set of observers on a topic ”`/observations`” whereas the diagnosis are published by the diagnosis engine on a topic ”`/diagnosis`”. As long as the diagnoses (Δ) is empty meaning there is no faulty component the repair engine does not perform any repair actions. When the repair engine receives a diagnosis message ($\Delta \neq \varnothing$) it converts the diagnosis as well as the observations into to a planning problem description while the domain description is provided to the repair engine as an input. In order to describe and enable repair actions we model the repair as a planning domain where the problem description is simply a Planning Domain Definition Language (PDDL) description of an initial and a goal state. Within the repair engine we use a Java-based GraphPlan implementation to find a plan (sequence of repair actions) [BF97]. Once the planner has found a valid repair plan, the repair engine starts execution of the sequence of the repair actions. The execution of the actions is triggered by invocation of an appropriate repair action server. Because of the standard signature for repair action servers (a unique name and a list of parameter strings) the matching between the planner and the ROS-based system is easy to achieve. Algorithm 11 describes how repair engine performs to accomplish the repair task. In Line 1 and 2 the diagnosis (Δ) and the observations (Θ) are obtained from the corresponding subscribed topics. The Δ and the Θ are then converted into a planning problem description (Line 3). The set of the repair actions in the form of a plan $\mathcal{P}$ is obtained (Line 4) using a problem and a domain description. At the end an action server is invoked against each of the repair actions in the plan.

Algorithm 11: $Repair(\Delta, \Theta, \mathcal{DOM})$

```
  input : $\Delta$ ... a set of good and bad diagnosis
  input : $\Theta$ ... a set of literals observations
  input : $\mathcal{DOM}$ ... a planning domain description
1 $\mathcal{PROB} \leftarrow makePlanningProblem(\Delta, \Theta)$
2 $\mathcal{P} \leftarrow callPlanner(\mathcal{PROB}, \mathcal{DOM})$
3 foreach $i = 1$ to $length(\mathcal{P})$ do
4     Call $\rho_i(\delta)$ ;$\rho_i$ is a repair action server on arguments $\delta$
5     Wait until $\rho_i$ completes
6 end
```

Like diagnosis module kernel, the repair engine kernel also executes two main concurrently running activities; collection of the observations and the execution of the repair process.

7.1.1 Observation Collection

The repair engine uses the observations along with the diagnosis for a planning purpose. It obtains the observations by simply subscribing to the topic "`/observations`". Each single observation θ_t that occurs at time t on the topic is integrated into a logical observation set Θ as described in section 6.4.1 of Chapter 6.

As an observation provides information about a component with its status, for instance, $running(jaguar\_node)$ which means component $jaguar\_node$ has the status $running$. The repair engine exploits information from observations and constructs a planning problem description. The components are used in the $objects$ section and their status is used in $init$ section of the problem description.

Example 7.1. *An observation set $\Theta_t = \{ok(map), ok(joy), \neg ok(scan)\}$ gives three objects* $\{$`map, joy, scan`$\}$ *for the* `object` *section, and* $\{$ `(ok map) (ok joy) (not(ok scan))` $\}$ *for the $init$ section of the planning problem description.*

7.1.2 Repair Execution

One of the two main activities of the repair engine is executing the repair actions. This activity comprises two sequential processes; generating the repair plan, and invoking the repair action servers one by one as discussed in the following sections.

7.1.2.1 Generating the repair plan

Given a way of describing the world (W), an initial state of the world (S_i), a goal state G, and a set of possible actions (A=$\{\delta_i\}$) to change the world the generation of the plan (P) is the process of determining set of actions which when applied to the initial state (S_i) reaches to a state that satisfies the goal G, e.g., $P = \langle \delta_1(S_i, S_{i+1}), \delta_2, \delta_3, ..., \delta_n(S_n, G) \rangle$

In order to describe and enable the repair actions, the repair process is modeled as a planning problem. We use widely recognized Planning Domain Definition Language (PDDL) to represent the domain and problem definitions for the planning [KBC+98]. The advantage of this approach is that a wide range of existing high-performance planner and various extensions to the classical planning like ”typing” can be easily used directly. The description of a planning problem in PDDL comprises two parts: (1) a domain description and (2) a problem description. Using domain and problem description a planner finds a plan for a problem. The plan consists of a sequence of the actions which if executed solve the planning problem.

A. Domain description: The PDDL domain description contains all definitions of actions and domain objects on which actions are applied. To parse the PDDL domain description, we use an open-source Java-based package *pddl4j* [Pel08] and a *GraphPlan* implementation to find a plan $\mathcal{P}$(a sequence of the repair actions) [BF97]. All the possible repair actions are defined in the domain description. We have two kinds of repair actions (1) for the software components, i.e., a running software entity can be stopped or a stopped software entity can be started, and (2) for the hardware components, i.e., a hardware component can be switched `On` or `Off`. Here we define four repair actions $start\_node$ and $stop\_node$, $power\_up$ and $shutdown$.

Example 7.2. *The repair action for stopping a software node n is:*

```
(:action stop_node
   :parameters (?n)
   :precondition (and (bad ?n)(running ?n))
   :effect (and (bad ?n)(not (running ?n)))
)
```

The action describes that a software node n can be started only if it is declared faulty and also observed as not running. The action's effect is that the node n is running.

B. Problem description: The problem description is based on combination of the diagnosis $\Delta = \{\Delta_{bad}, \Delta_{good}\}$ and the set of observations Θ. The set Δ_{bad} contains the components that are faulty, and Δ_{good} is set of the working components. The repair planning problem description is simply a PDDL description of the initial state I and the goal G:

- an initial state
  $I = \Theta \cup \bigcup_{c \in \Delta_{bad}} AB(c) \cup \bigcup_{c \in \Delta_{good}} \neg AB(c)$
- a goal state $G = \bigcup_{c \in COMPS} \neg AB(c)$

The initial state I is union of the actual observations and the state of all components. The goal G of a repair plan is that all components work correctly again.
Considering the robot architectural model ($\mathcal{RAM}$) discussed in Chapter 4, with the following obtained sets:

$$
\begin{aligned}
&COMPS = \{ja, jn, hu, hn, imu, in, la, lan, jtn, hm, lac, jyn, rviz\}\\
&\Theta = \{running(jn), running(hn), running(in), running(lan), running(jtn),\\
&\quad running(hm), running(lac), running(jyn), running(rviz), ok(cmd\_vel),\\
&\quad running(hm), running(lac), running(jyn), running(rviz), ok(cmd\_vel),\\
&\quad ok(scan), ok(imu\_data), ok(scan), ok(imu\_data), ok(joy), not\_ok(map),\\
&\quad ok(la\_servo1\_moving), ok(la\_servo2\_moving), on(ja), on(hu), on(la)\}\\
&\Delta = \{\Delta_{bad} = \{hm\}, \Delta_{good} = \{ja, jn, hu, hn, imu, in, la, lan, jtn, lac, jyn, rviz\}\}
\end{aligned}
$$

A simple planning problem description will look like as follows:

```
(define
   (problem repair_problem_description)
   (:init (running jn)(running hn)(running in)
          (running lan)(running jtn)(running hm)
          (running lac)(running jyn)(running rviz)
          (ok cmd_vel)(ok scan)(ok imu_data)(ok joy)
          (ok la_servo1_moving)(ok la_servo2_moving)
          (not_ok map)(on ja)(on hu)(on la)(good jn)
          (good hn)(good in)(good lan)(good jtn)
          (bad hm)(good lac)(good jyn)(good rviz))
   (:goal (and (good jn)(good hn)(good in)
               (good lan)(good jtn)(good hm)
               (good lac)(good jyn)(good rviz)))
)
```

Example 7.3. *(Continued) Given the planning domain description and planning problem description as above, a plan generated will be as:*

$$\mathcal{P} = \{\langle stop\_node, hm\rangle, \langle start\_node, hm\rangle\}$$

7.1.2.2 Invoking repair action servers

Once a plan containing the repair actions is generated by the planner the repair engine executes the repair actions. Our diagnosis and repair system offers a number of repair action servers. A repair action server is a ROS action server that receives request for goal from its action client (repair system), fulfils the goal and returns the status `SUCCESS` if it finishes the task successfully or `FAILED`, back to the action client. The name of the repair action server is kept same as the action name obtained in the plan. For instance against the repair action $\langle start\_node, hm \rangle$, the repair engine will invoke a repair action server called `start_node` for the parameter `hm`. The repair action server `start_node` is an executable process (a ROS node) that uses a Linux based operating system commands to launch the node given as a parameter, e.g., start_node(hector_mapping) will cause repair action server `start_node` to launch `hector_mapping` node. We have built four ROS-based repair action servers:

1. **start_node:** It uses "`system()`" command to execute a ROS command "`roslaunch node`" where the `node` is a parameter provided by the repair engine.

2. **stop_node:** It also uses the command "`system()`" to execute a ROS-based shell command "`rosnode kill node`" where the `node` is a parameter provided by the repair engine.

3. **power_up:** It uses the TCP/IP protocol to send text-based command "`[SWT,component,1]`" to ask the diagnostic board (discussed in next chapter) to switch ON the `component`. Where "`SWT`" is a board protocol command.

4. **shutdown:** Same as above, it also uses the TCP/IP protocol to send text-based command "`[SWT,component,0]`" to ask the diagnostic board to switch `OFF` the `component` where "`SWT`" is a board protocol command.

The repair engine calls a repair action server, waits for the completion message, and invokes the next repair action server by taking next repair action in the plan. Because of the standard signature for repair action servers (a unique name and a list of parameter strings) the matching between the planner and the ROS-based system is easy to achieve. Please note that the planner simply waits for the completion message of the called action server and currently does not check the action's effect defined in planning domain description. A better strategy for the future would be to wait until the effects have been established. For instance, it might take longer for a node's output topics to become correct than simply the time to restart the node.

7.1.2.3 Integrating additional repair action servers

The repair action package allows integration of new repair action servers needed to meet user's need. In order to integrating additional repair actions the repair engine requires:

- the required domain actions need to be defined in the domain description.
- the ROS-based repair action servers need to be implemented according to the need, keeping the same signature as in domain description actions.

Chapter 8

Hardware Integration

This chapter deals with the faults in the hardware components. It presents a diagnostic board used for detecting faults in the hardware sensors of the robotics systems. Moreover, it also describes how the diagnostic board works with the diagnosis and repair system for detecting and repairing hardware faults. The work presented in this chapter has been published in [ZL13, ZL14].

8.1 Overview

The proposed diagnosis and repair architecture for this dissertation aims to be an initiative for recovering ROS-based robotics systems from both software and hardware faults. Diagnosing and repairing a software fault is although effort requiring, but the hardware faults detection is also complex, technical, and carries critical issues about hardware. For the software fault localization and recovery, one has to deal with the software issues on both sides diagnosing system as well as diagnosed system. The software faults include software crash, malfunctioning, change in frequency of publishing data, etc., need a monitoring, a diagnosing, and a repair system which are themselves software programs. In contrary to this, typical hardware faults includes device broken, device shutdown, heated up, flat tire, wire cut, etc. In order to detect and repair such hardware problems not only requires software system but also an additional intelligent hardware unit that functions together with its counterpart software diagnosis system. Such an intelligent hardware unit closely monitors hardware components, and cooperates in diagnosing and repairing the faults by providing the hardware behavior, and has capability to control the hardware components. For example, a robotics system in a critical domain (e.g., space) may acquire parallel pairs of sensors where one sensor can be activated if other leaves working. Similar systems called power management systems [SRA^{+}11] had already been used for robotics systems. Such systems allow users to manually switch `ON` or `OFF` the individual on-board system components and perform current and voltage monitoring. In addition to this, our system is also capable to perform these steps autonomously without manually switching the hardware components.

8.2 Diagnostic Board

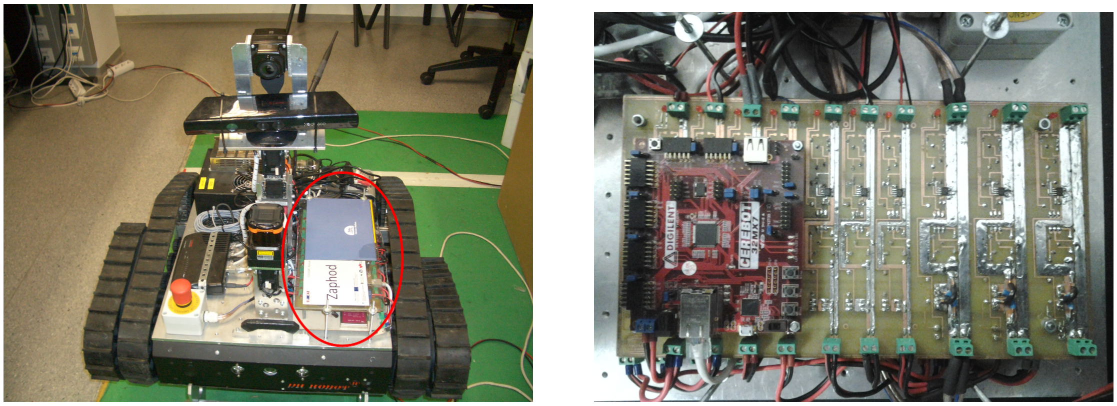
An intelligent hardware board called diagnostic board is developed for the hardware faults detection and repair. It is a micro-controller based intelligent board comprising input and output channels. Its main objectives include: (1) smartly distribution of power supply to the hardware components, (2) monitoring voltage level on its channels, (3) current measurements on the channels, (4) automatically set or reset the channels. Following sections explain the hardware, the protocol, and the controller software of the diagnostic board:

Definition 8.1 (**channel**). *A channel is an input or output connection on the diagnostic board.*

8.2.1 Architecture

The diagnosis board (Figure 8.1) comprises 10 input and 10 output channels as an interface between the hardware sensors and the diagnostic board. The inputs are usually connected to the power supply distribution center with DC-to-DC power converters. The robot sensors are connected with the output channels. The diagnostic board is controlled by a micro-controller that has the option to set/reset each individual channel and also main communication line over the Ethernet. A TCP/IP based protocol system is established for the diagnostic board to supervise system power consumption and performance of the actions. The diagnostic board can be supplied separate power supply source, for instance, additional small battery that can power up the diagnostic board independently from the robot. Out of the 10 channels, 3 are high power supply channels with specification of a maximum 60V voltages and a maximum of 15A current per channel, other 7 channels are limited to a maximum load of 20V voltages and 5A current. On each input channel there is DC-to-DC converter which is a compliment with power consumption on the output channels of the board. Each channel has three main components: (1) a solid state relay switch for powering up/down a channel, (2) an Optocoupler sensor for voltage detection between range 3.3-60V, (3) a current sensor for current measurement with analog output to micro-controller within range 0-30A.

The main processing unit is micro-controller which controls the diagnostic board's functionality. The communication with outer world is achieved over a standard 10/100 Ethernet interface. The board software is developed to open a server and wait for a client to connect on a specified port and IP, after the board is successfully connected the server broadcasts channels specifications to its clients. The diagnostic board can restart the whole system and act as a master which can be determined in different repairing scenarios. At start up two channels for pc and router are switched on, later all other channels can be powered up by the MBD system automatically.



(a) (b)

Figure 8.1: (b) Jaguar robot with the hardware diagnostic board mounted on it (red encircled) (a) Diagnostic hardware board with input/output channels.

8.2.2 Features

The diagnostic board provides two important functionalities to the diagnosis and repair system. Firstly it provides observations about the hardware components attached to it, and secondly it enables the repair system to execute hardware related repair actions. These both functionalities are briefly discussed below:

1. **Observations :** The board continuously monitors hardware connections to hardware components. The following observations and activities we obtain from the board:

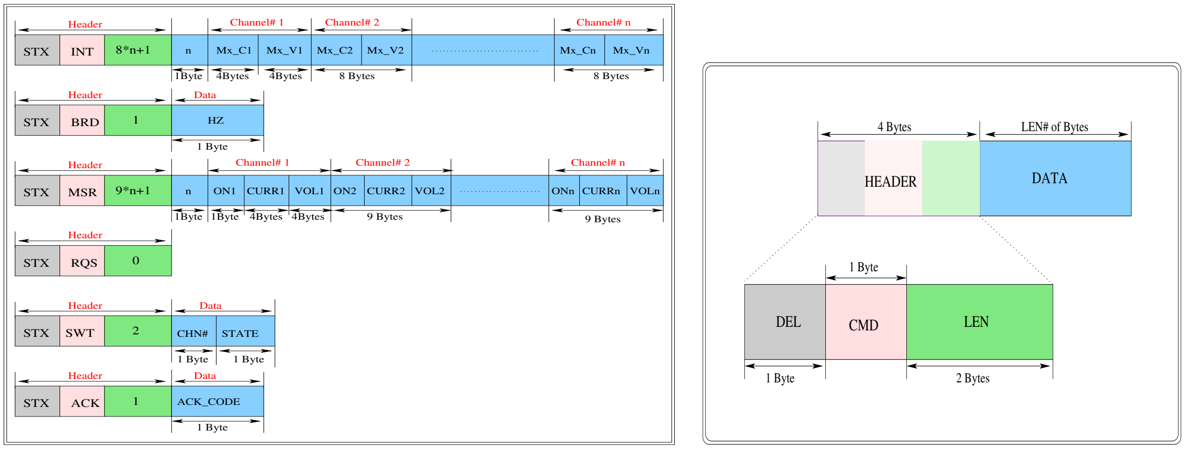
   - **Voltage Level :** As many as 10 different hardware components can be connected to the board. How much voltage level is consumed on a connection for a particular device, is monitored and provided as observation to the diagnosis system. A device may start consuming more voltage if its gears/motors are blocked.
   - **Current Consumption :** Every hardware device consumes current power. This is also measured on each channel and observations come out of the board.
   - **Connection Status :** `ON/OFF` status of the connected components is also provided by the board as the observations.

2. **Actions :** A hardware component attached to the board, if diagnosed as faulty should be either powered off or restarted. In order to achieve these capabilities the board provides two readily actions on its channels:

   - **Set Action :** The board uses this action to switch `ON` a channel, i.e., components are connected with the channels and switches `ON` a channel actually switches `ON` the attached hardware component. This action is executed if a device is connected but not powered up.

- ○ **Reset Action :** This action is performed to switch `OFF` a channel. This action can be required if a device need to be shutdown.

3. **Initial Setting :** The board also has memory to store default setting, i.e., it contains small memory where initial setting can be specified, e.g., in our case the board always initially power up two channels, one to which PC is connected and the other for Router.



(a) (b)

Figure 8.2: (a) Diagnostic board protocol commands (b) Diagnostic board protocol suite.

8.2.3 Protocol

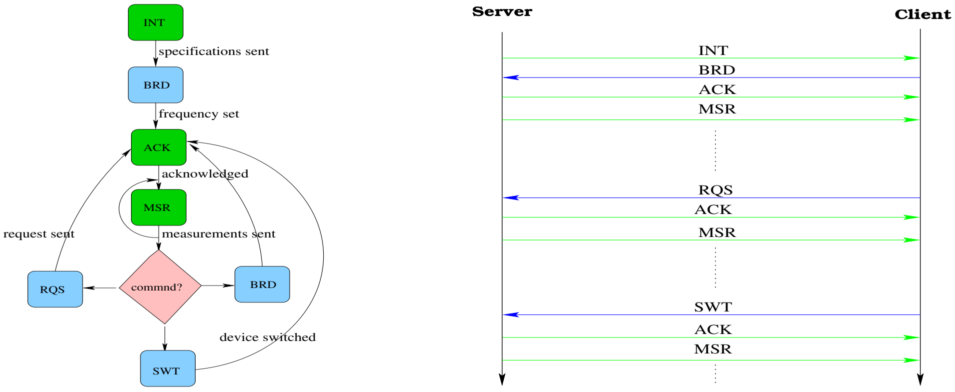
The diagnostic board protocol uses a command suit that consists of basically two parts, namely *HEADER* and *DATA* as shown in Figure 8.2(b). *HEADER* is further partitioned into three parts: (1) *DEL*-a delimiter, (2) *CMD*-a command number, and (3) *LEN*-the length of data in bytes . Despite the fact that TCP/IP protocol is significantly secure in terms of delivery of a packet, we additionally introduce a delimiter *DEL* that identifies start of a command. It is one byte long, and contains start of the command represented by a field *STX*. The value of *STX* is 6 in order to avoid ambiguity with command numbers. The command *CMD* is also 1 byte long, and it occupies command number in order to enable recipient recognize which command it has received. The command *LEN* is 2 bytes long field. It contains the size of *DATA* field in terms of number of bytes. The *DATA* field is a varying size part of the command. Its size is different for different commands. The protocol comprises six different commands, namely *INT, BRD, MSR, RQS, SWT* and *ACK*. Each of these commands is identified by a unique number, and is used for a particular purpose as described in Table 8.1. The *INT* command is initialization command which carries voltage and current specifications for the available channels. Each channel's specification in the *DATA* part of the command occupies 8 bytes with first 4 bytes for maximum current and next 4 bytes for maximum voltage supported by the channel. The *LEN* field in the header of the *INT* command is assigned numeric value $8n$+1 where n is number of available channels and 1 is due to first byte in *DATA* specifying total number of available channels. The purpose of the *BRD* command is to set the server's broadcasting frequency for measurements. Its *DATA* part is only 1 byte long which contains the value for frequency where 0 value means stop broadcasting. The *MSR* command carries voltage and current measurements present on the channels at run-time. Its *DATA* part contains $9n$+1 bytes. First byte is for the number of channels, and then sequence of 9 bytes for each of the n channels. First out of 9 bytes is for the state of the channel either 0 (`OFF`) or 1 (`ON`). Next pair of 4 bytes is reserved for the current and the voltage measured on the channels. The *RQS* command is used for requesting for the current and the voltage measurements. As this command is simply used for a request so it does not contain *DATA* field. The objective of the *SWT* command is to switch `ON` or `OFF` a particular channel on the board. It carries a channel number (1 byte long) and required status (1 byte) for the channel. Last command is the *ACK* command. It is used by server to send an acknowledgement of receipt of a command. It contains one byte *DATA* field which carries *ack_code* with value either 0, 1, or 2. The value 0 means `OK` , 1 means incorrect parameter, and 2 means incomplete command.

Example 8.1. *The acknowledgement command (ACK) after successfully receiving correct request command (RQS)* $[6, 3, 0]$*, will look like* $[6, 5, 1, 0]$.

Name	Code	No	Purpose
Initialization	INT	0	initialization after connection
Broadcasting	BRD	1	setting broadcasting frequency
Measurements	MSR	2	provides channels measurements
Request	RQS	3	request for measurements
Switch	SWT	4	switching ON/OFF a particular channel
Acknowledgement	ACK	5	receipt acknowledgement

Table 8.1: Board protocol commands with their numbers

Three commands, namely *INT, MSR,* and *ACK* are server side commands, i.e., these commands are issued from the server to the client. Other three commands, namely *BRD, RQS,* and *SWT* are client side commands, i.e., the client sends these commands to the server. The server runs inside the diagnostic board whereas the client communicates the server from remote computer. Figure 8.3 shows the flow and sequence of the commands between the board server and the client. In the beginning, after establishing a connection, the server sends the *INT* command with specification for the channels and receives the *BRD* command from the client for setting broadcasting frequency. The server then sends the *ACK* command to the client and afterwards starts sending the *MSR* commands giving measurements with specified broadcasting frequency. At this point the client may send *RQS, BRD* or *SWT* command.



(a) (b)

Figure 8.3: (a) Flow of server (green) and client (blue) commands(b) Sequence of commands.

8.2.4 Controllers

The diagnostic board is a hardware that needs to be operated through some kind of software program we call it *controller*. The diagnostic board is controlled and utilized for the diagnosis process through two simple controllers, namely *server controller* and *client controller*. The server controller acts as a driver and controls overall activities of the board. The client controller interacts with the server controller in order to get observations for the diagnosis, and makes requests for the repair actions for the hardware repair. Both kind of the controllers are discussed in the following sections:

8.2.4.1 Server Controller

The board server controller is a driver which runs inside the board's micro controller and controls the activities on the board. It provides all necessary information about the hardware components attached to it. This information is used as the observations for the hardware diagnosis process. Moreover, the board server controller also provides the facility to perform the repair actions by switching the hardware components connected to the board's channels. It opens a port on a specific *IP* to permit the clients to connect with the board on TCP/IP protocol allowing them

Algorithm 12: $Board\_Server\_Controller(ip, port)$

```
input  : $ip$ ... ip address
input  : $port$ ... port number
$client \leftarrow wait\_for\_conn\_request(ip, port)$
$channels \leftarrow get\_channels()$
$spf \leftarrow get\_specifications(channels)$
$\mathcal{P} \leftarrow prepare\_INT(spf)$
$send\_to(client, \mathcal{P})$
while $(rcv\_\mathcal{P} \leftarrow receive\_from(client))$ do
    $ack\_code \leftarrow check\_command(rcv\_\mathcal{P})$
    $\mathcal{P} \leftarrow prepare\_ACK(ack\_code)$
    $send\_to(client, \mathcal{P})$
    if $ack\_code \neq 0$ then
        $continue$
    end
    $\mathcal{H} \leftarrow rcv\_\mathcal{P}.HEADER$
    $\mathcal{D} \leftarrow rcv\_\mathcal{P}.DATA$
    if $\mathcal{H}.CMD == BRD$ then
        $frq \leftarrow \mathcal{D}.HZ$
    else if $\mathcal{H}.CMD == RQS$ then
        $set\_frequency(frq)$
        while $(no\_data\_from(client))$ do
            $msr \leftarrow get\_measurements(channels)$
            $\mathcal{P} \leftarrow prepare\_MSR(msr)$
            $send\_to(client, \mathcal{P})$
        end
    else
        $ch\_id \leftarrow \mathcal{D}.CHN\#$
        $state \leftarrow \mathcal{D}.STATE$
        $switch(ch\_id, state)$
    end
end
```

to talk with the board under its own protocol described above, and transmit necessary and required information from the board to the client. It provides the information like number of the total channels for the hardware components, maximum voltage and current specification on each channel, on/off status of individual channel, and present voltage and current measurements on each individual channel. Algorithm 12 describes functionality overview of the server controller. Lines 1-5 take the client request, collect and send channels specifications to the client using the *INT* command. Lines 7-9 send acknowledgement back to the client. Line 16 sets broadcasting frequency after receiving the *BRD* command from the client. Against the client's command *RQS*, it sends channels measurements to the client with the required frequency (Lines 18-23). The command *SWT* is dealt by switching a specified channel (Lines 22-24).

The board server controller when starts it firstly reads a default setting from the board's memory and performs the actions listed in the memory. For instance, in our case two components the *PC* and the *Router* are specified to be always powered up in the beginning. The *PC* is for running the robotics system and the *router* for enabling remote system to connect with *PC* and the board. After having performed default actions the board opens an *IP* address and waits for the clients requests for connection.

8.2.4.2 Client Controller

This is a main ROS node that communicates with the board server controller on TCP/IP using the board protocol discussed in the section 8.2.3. Firstly it takes measurements from the board and publishes them in such a form that it can be used by the monitoring system for the diagnosis and the repair process. Secondly it has the ability to request the board server controller for performing the hardware repair actions required for the repair process. Algorithm 13 describes functionality of the board client controller. The board client controller connects with the server, sets up broadcasting frequency and receives an acknowledgement from the server (Lines 1-6). Line 7 calls Algorithm 14 that checks the received acknowledgement, corrects the command if not, and sends back. Lines 13 and 14 collect the measurements and publish on the ROS topic */board_measurements* which makes it possible to get the observations for the diagnosis process. The client controller interacts with the repair planner in order to receive a planned repair actions (Line 15). If there is any repair action to be carried out by the diagnostic board the client controller prepares switch (*SWT*) commands and sends to the server controller (Lines 15-18). Finally the client controller resumes receiving the measurements by sending the request command *RQS* to the server controller (Lines 17-24).

Algorithm 13: $Board\_Client\_Controller(ip, port, f)$

```
input : $ip$ ... ip address of server
input : $port$ ... port number on server
input : $f$ ... initial frequency
$server \leftarrow make\_conn\_request(ip, port)$
$spf \leftarrow receive\_from(server, INT)$
$HZ \leftarrow f$
$\mathcal{P} \leftarrow prepare\_command(\{BRD, HZ\})$
$send\_to(server, \mathcal{P})$
$ack \leftarrow receive\_from(server)$
$check\_Acknowledgment(ack, \mathcal{P}, server, \{BRD, HZ\})$
$\mathcal{P} \leftarrow prepare\_command(\{RQS\})$
$send\_to(server, \mathcal{P})$
$ack \leftarrow receive\_from(server)$
$check\_Acknowledgment(ack, \mathcal{P}, server, \{RQS\})$
while $rcv = receive\_from(server, MSR)$ do
    $msg \leftarrow build\_ROS\_msg(rcv.DATA)$
    $ROS\_publish(/board\_measurments, msg)$
    $action \leftarrow look\_\_up\_planner()$
    if $action == switch$ then
        $\mathcal{P} \leftarrow prepare\_command(\{SWT, action\})$
        $send\_to(server, \mathcal{P})$
        $ack \leftarrow receive\_from(server)$
        $check\_Acknowledgment(ack, \mathcal{P}, server, \{SWT, action\})$
        $\mathcal{P} \leftarrow prepare\_command(\{RQS\})$
        $send\_to(server, \mathcal{P})$
        $ack \leftarrow receive\_from(server)$
        $check\_Acknowledgment(ack, \mathcal{P}, server, \{RQS\})$
    end
end
```

Algorithm 14: $check\_Acknowledgment(ack, \mathcal{P}, server, \Sigma)$

```
input : $ack$ ... acknowledgement
input : $\mathcal{P}$ ... message
input : $server$ ... server instance
input : $\Sigma$ ... set of command parameters
while $ack.DATA.ACK\_CODE \neq 0$ do
    $\mathcal{P} \leftarrow prepare\_command(\Sigma)$
    $send\_to(server, \mathcal{P})$
    $ack \leftarrow receive\_from(server)$
end
```

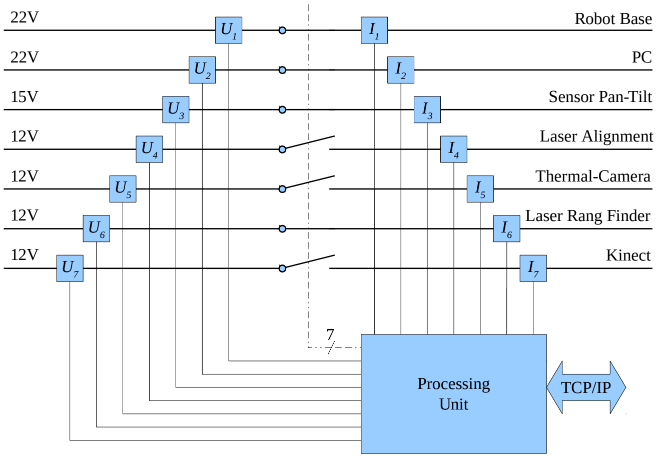


Figure 8.4: Different hardware components connected with the diagnostic board.

8.3 Hardware Observation

The model-based diagnosis process requires the observations besides a model. The presented diagnosis system uses the observations in the form of logical literals. In order to convert measurements coming from the diagnostic board into compatible observations we use an observer called hardware observer *HObs* discussed in Section 5.2.6 of Chapter 5. The hardware observer subscribes to the client controller's topic */board_measurements* and converts the information into first order literals like '*on(component)*' and '*¬on(component)*'. *HObs* publishes all observations on the ROS topic */observations*.

Example 8.2. *As depicted in Figure 8.4 the different hardware components, namely* `RobotBase`*,* `PC`*,* `SensorPanTilt`*,* `LaserAlignment`*,* `ThermalCamera`*,* `LaserSensor`*, and* `Kinect` *are connected with* 7 *different channels of the diagnostic board. Four of these components are powered up whereas three components are disconnected from the power supply. The server controller sends the measurements with the status* 1 *for powered up components and* 0 *for others, using MSR command. The client controller will receive the measurements and publish status of all seven components on the topic /board_measurements. The hardware observer* `HObs` *will publish* {`on(robot_base)`,`on(pc)`,`on(sensor_pan_tilt)`,¬`on(laser_alignment)`, ¬`on(thermal_camera)`,`on(laser_range_finder)`,¬`on(kinect)`} *on the topic /observations.*

8.4 Hardware Diagnosis

The diagnosis process is explained in detail in Chapter 6. The process requires the observations and a model in order to localize a fault. The hardware observations we obtain through *HObs*, and the model is obtained by converting robot architectural model (*RAM*) into robot behavior model (*RBM*) as discussed in Chapter 4. There are a number of scenarios where we utilized diagnostic board for the hardware diagnosis, for example, a servo motor switches off if there is enough pressure preventing it to rotate, this may happen when laser alignment system gets blocked while robot moves up or downwards.

We consider another scenario which is more likely to occur than the one discussed above. The robot base *jaguar* connects with its node *jagaur_node* over a wireless, and the *jagaur_node* publishes odometry information on a topic */pose*. The topic */pose* may get disturbed because of two reasons: *jagaur_node* may stop making the */pose* topic disappear, or the *jaguar* may disconnect from the *jagaur_node* causing the */pose* to stop publishing data. Therefore, the cause of the fault in the topic */pose* can be either the *jagaur* or the *jagaur_node*, and the optimum solution is to consider both of these possible root causes of the fault as stated in following rule from robot behavior model ($\mathcal{RBM}$):

$$\begin{aligned}&\neg AB(jaguar) \rightarrow on(jaguar)\\&\neg AB(jaguar\_node) \rightarrow running(jaguar\_node)\\&\neg AB(jaguar) \wedge \neg AB(jaguar\_node) \rightarrow ok(pose)\end{aligned}$$

which states that if we assume both the *jagaur* and the *jagaur_node* are working properly the topic */pose* must work correctly as well. If the robot is switched `ON`, and connected with the *jagaur_node*, and the topic */pose* is publishing odometry with the right frequency then the observers $HObs$, NOb_{jaguar_node}, and $GObs_{pose}$ collectively provide the observations:

$$\Theta = \{on(jaguar), running(jaguar\_node), ok(pose)\}$$

The only diagnosis will be:

$$\Delta = \{\Delta_{bad} = \varnothing, \Delta_{good} = \{jaguar, jaguar\_node\}\}$$

Now suppose the robot base *"jaguar"* disconnects from the *jagaur_node* so */pose* will stop publishing and the observations we get:

$$\Theta = \{on(jaguar), running(jaguar\_node), \neg ok(pose)\}$$

the diagnosis will be:

$$\Delta = \{\Delta_1, \Delta_2\}$$

where:
$$\Delta_1 = \{\Delta_{bad} = \{jaguar\}, \Delta_{good} = \{jaguar\_node\}\}$$
$$\Delta_2 = \{\Delta_{bad} = \{jaguar\_node\}, \Delta_{good} = \{jaguar\}\}$$

8.5 Hardware Repair

The repair process is discussed in detail in Chapter 7. We use a planner-based repair engine which takes the observations and the diagnosis as input and generates plan of the repair actions. The repair actions in the plan are then executed one by one by invoking the repair action servers. In case of the hardware repair we have two repair action servers, namely *power_up*, and *shutdown*. Both of these repair action servers use board client controller to request the board server controller in order to switch ON or OFF a particular component. Continuing with the same scenario discussed above the repair planner first gets both the observations (Θ) and the diagnosis (Δ_1):

$$\Theta = \{on(jaguar), running(jaguar\_node), \neg ok(pose)\}$$
$$\Delta_1 = \{\Delta_{bad} = \{jaguar\}, \Delta_{good} = \{jaguar\_node\}\}$$

and produces plan of the repair actions as:

$$\mathcal{P} = \{shutdown(jaguar), power\_up(jaguar)\}$$

the repair engine sends the actions $shutdown(jaguar)$ and $power\_up(jaguar)$ one by one to the board client controller causing it to send *SWT* command to the board server controller in order to switch OFF and then ON *jaguar*. This brings *jaguar* again on wireless network. In order to connect the *jaguar_node* with the *jaguar* it is necessary to restart the *jaguar_node*. For this purpose the planner takes the following observations and the diagnosis:

$$\Theta = \{on(jaguar), running(jaguar\_node), \neg ok(pose)\}$$
$$\Delta_2 = \{\Delta_{bad} = \{jaguar\_node\}, \Delta_{good} = \{jaguar\}\}$$

and repair planner generates new plan:

$$\mathcal{P} = \{stop\_node(jaguar\_node), start\_node(jaguar\_node)\}$$

and the repair engine invokes $stop\_node$ and $start\_node$ the repair action servers in a sequence for the node $jaguar\_node$. This restarts the *jaguar_node* and its connection with the *jaguar* is again established making the topic */pose* publishing the odometry data again.

Chapter 9

Model Learning

In this chapter we discuss an approach for how to generate a diagnosis model. Moreover, it also discusses how two fields are correlated on the basis of their qualitative trends. The research works related to the model learning have been published in [ZS13a, ZS13b].

9.1 Model Generation

During accomplishing a task it is quite likely that a robotics system's component shows an undesired behavior that can be caused by a wide range of faults such as defective hardware or software deadlocks. This phenomenon is caused by the complex interactions within the robot system and the non-deterministic interaction with the dynamic environment. In order to be able to automatically cope with such problems it is necessary to have a monitoring system that is not only able to detect such faults but is also able to repair them at runtime. If a model-based diagnosis approach is used one needs a model of the correct system behavior. This behavior model is then compared to the behavior observed at runtime in order to detect and localize the faults.

Such a diagnosis model can be acquired using three basic approaches. The first approach is to reuse requirements or engineering models that are already available if for instance a model-driven development process is used [BGVB10]. If no reusable models are available the diagnosis models can be created by hand. While this second approach is quite widespread it is cumbersome and error-prune in particular for complex systems. In this work we follow a third approach that learns the diagnosis model on-line during a controlled learning phase.

In order to acquire a model on-line one has to have some mechanisms to analyze the computation and interaction of the system's components. The analyzed data describes the behavior of the robot system at run-time and can be used to generate the diagnosis model. A basic assumption needed for this approach to work is that the system produces no faults during the learning phase. Moreover, it has to be assumed that the robot shows all possible behaviors during the learning phase in order to generate a complete diagnosis model.

9.1.1 Recording the Running System

This section proposes to use information recorded from a system actually performing a complete task to extract the diagnosis model automatically. ROS supports this approach because it allows easy access to information about the computation graph, the involved nodes, the communication between nodes and the content (data) of exchanged messages. The recorded information belong to three groups: First comprising running nodes and their properties, second group consists of communication patterns and final group is the exchanged data. The first group is directly available from ROS and can be directly transferred to rules in the model (e.g., which nodes have to run). For the second group ROS only provides information about exchanged messages. Here a statistical analysis has to be done in order to detect communication patterns (e.g., which nodes communicate regularly). Finally, the values contained in the messages can be correlated to detect functional dependencies (e.g., if one value increases another one has to increase as well). Here we follow a qualitative reasoning approach [BS04]. The input to the generation and instantiation step is recorded during a fault-free execution of tasks by the robot system and comprises three parts: (1) the computation graph, (2) the communication via all topics and (3) further property observation.

Definition 9.1 (topic-node relation). *A topic-node relation is a tuple $r_i = \langle t_i, N_{t_i} \rangle$ where t_i is a topic and N_{t_i} is a set of nodes publishing on or subscribing to topic t_i.*

The computation graph can be directly obtained using ROS system functions.

Definition 9.2 (computation graph). *A computation graph is a tuple $G = \langle N, T, P, S \rangle$ where $N = \{n_1, ..., n_{k_N}\}$ is the set of running nodes, $T = \{t_1, ..., t_{k_T}\}$ is the set of topics, $P = \{p_1, .., p_{k_P}\}$ is the set of publishing topic-node relation and $S = \{s_1, .., s_{k_S}\}$ is the set of subscribing topic-node relation.*

For better readability we assume access functions for tuples in the form $e(t)$ for accessing the entry e of tuple t. A communication via a topic t_i is a time-ordered list of exchanged messages that can be simply obtained by subscribing to topic t_i.

Definition 9.3. *The communication (CO) for a topic t_i is a time-ordered list $M_{t_i} = \langle m^1_{t_i}, ..., m^{k_{M_{t_i}}}_{t_i} \rangle$ where $m^j_{t_i}$ is a tuple $\langle \nu, t \rangle$ with $\nu = \{v_1, ..., v_{k_{\nu_{t_i}}}\}$ a set of atomic values in the exchanged message and t is the time of the occurrence of the message. Atomic values are data types that cannot be decomposed any further such as integer or floats. The set of all communications is denoted as M. We assume that message layouts do not change during runtime. Therefore, we treat the number of values $k_{\nu_{t_i}}$ in a message for topic t_i as a constant and define a* *function V_c that extract from a communication M_{t_i} the list of the $j^{th} value, j \in \{1, ..., k_{\nu_{t_i}}\}$, and their occurrence time: $V_c : CO \times \mathbb{N}^+ \rightarrow \{\langle \mathbb{R}, \mathbb{R} \ \rangle\}$. Finally, we define two functions Γ and Δ that extract from a communication M_{t_i} the list of the occurrences respectively the time difference to the previous occurrence: $\Gamma, \Delta : CO \rightarrow \{\mathbb{R}\}$. The time difference for the first occurrence is defined as* 0.

Property observations for a node are time-ordered list of real numbers. Currently two kinds of observations types are supported: (1) cpu usage and (2) memory usage. As nodes are processes these observations can be easily acquired using OS functionality (e.g., proc file system).

Definition 9.4. *A property observation (PO) for a node n_i is a time-ordered list $\Pi_{n_i} = \langle \pi_{n_i}^1, ..., \pi_{n_i}^{k_{\Pi_{n_i}}} \rangle$ where $\pi_{n_i}^j$ is a tuple $\langle \pi, \tau, t \rangle$ with $\pi \in \mathbb{R}$ the quantity of the observed property of type $\tau \in \{howtoput\tau_{CPU}, \tau_{MEM}\}$ at time t. The set of all property observations is denoted as* Π. *Moreover, we define a function V_p that returns for a property observation a set of all values of a particular type: $V_p : PO \times \tau \rightarrow \{\mathbb{R}\}$.*

Algorithm 15: $instantiateObs(G, M, \Pi)$

```
input  : $G$ ... the computation graph
input  : $M$ ... the communication set
input  : $\Pi$ ... the property observation set
output: a set of node observers $O_n$
output: a set of general observers $O_g$
output: a set of property observers $O_p$
output: a set of qualitative observers $O_q$
$O_n = \varnothing$,$= O_g\varnothing$,$O_q = \varnothing$
foreach $n_i \in N$ do
    $O_n = O_n \cup NObs(n_i)$
end
foreach $M_{t_i} \in M$ do
    $\bar{\Delta} = mean(\Delta(M_{t_i})), \sigma = stddev(\Delta(M_{t_i}))$
    if $\bar{\Delta}/\sigma > \alpha$ then
        $O_g = O_g \cup GObs(t_i, \bar{\Delta}, \sigma)$
    end
end
foreach $\Pi_{n_i} \in \Pi$, $t \in \tau$ do
    $\bar{\Pi} = mean(V_p(\Pi_{n_i}, t)), \sigma_\Pi = stddev(V_p(\Pi_{n_i}, t))$
    $O_p = O_p \cup PObs(n_i, t, \bar{\Pi}, \sigma_\Pi)$
end
$O_q = \cup instantiateQObs(M)$
```

9.1.2 Instantiating the Observers

The recording of the running system provides a computation graph G, a set of communications M and set of property observations Π. This information are fed into Algorithm 15 to derive the sets of observers that have to be instantiated to observe the running system during the diagnosis process. First the algorithm instantiates for each node a node observer and stores them in the set O_n (Lines 2-4). Then the algorithm calculates for each communication via a topic the mean and standard deviation of the time difference of successive messages. Using the heuristic that the fraction of the mean and the standard deviation is above a certain threshold α for topics communicating on a regular basis general observer are instantiated for such topics and stored in O_g (Lines 5-10). For each node where a property observation is available a property observer is instantiated with mean and standard deviation of the related property observation (Lines 11-14). Qualitative observer are instantiated using Algorithm 16 (Line 15). The algorithm first calculates the average communication differences for all pairs of topics (Lines 2). Then it is checked if both original or one original and one integrated communication is correlated (Lines 4-14). If two communications are qualitatively correlated a qualitative observer is instantiated where the last two parameter determines if a communication will be integrated during observation. $I(M)$ denotes the communication where the values in M are replaced by the sum of itself and all successive values. If two values in communications are qualitatively correlated is determined using Algorithm 17. The algorithm determines the qualitative correlation of two value by comparing their qualitative trend. Here we follow the ideas presented in [KSW09]. In technical systems and in particular in robot system related values can hardly be matched on an absolute scale. For instance the orientation measured by a compass and the odometry may start at different absolute angles. But the qualitative trend (e.g., increase, decrease, or constant) of both have to be the same for related values. Moreover, values of a physical system are noisy. Therefore, we use a sliding window and linear regression for calculating the slopes of a list of values. The sliding windows ws_i and ws_j (Line 1) ensures that in average C occurrences are used for the linear regression for the l^{th} value of topic t_i (Lines 3-6) respectively for the m^{th} value of topic t_j (Lines 7-10). According to [KSW09] a threshold b is necessary to classify a slope (trend) as increasing, decreasing or constant denoted by the symbols $+$,$-$ and 0. In order to be able to extract this parameter automatically we introduce two distinct parameter b_+ and b_- for increasing and decreasing trends (Lines 11-12). Assuming that the recorded data sufficiently represents the true probability distribution of trends we set the threshold for b_+ and b_- to the median of all positive slopes respectively all negative slopes. These parameters are finally used to classify the trends including checking higher-order derivatives as proposed in [KSW09] (Lines 13-14). Finally, the matches of the qualitative trends of both values are calculated with c representing the number of checked trends and m representing the number of matches (Lines 15-31). Because in general the number of occurrences of trends and their time is not equal for two topics we use an interpolation

Algorithm 16: $instantiateQObs(M)$

```
input  : $M$ ... the communication set
output: a set of qualitative observers $O_q$
$O_q = \varnothing$
foreach $\{\langle t_i, t_j \rangle | M_{t_i} \in M \wedge M_{t_j} \in M\}$ do
    $\bar{\Delta}_i = median(\Delta(M_{t_i}))$,$\bar{\Delta}_j = median(\Delta(M_{t_j}))$
    foreach
    $l \in \{1, ...., k_{\nu_{t_i}}\}$,$m \in \{1, ...., k_{\nu_{t_j}}\}$,$i \neq j \vee l \neq m$ do
        if $correlated(M_{t_i}, l, \bar{\Delta}_i, M_{t_j}, m, \bar{\Delta}_j)$ then
            $O_q = O_q \cup QObs(t_i, t_j, l, m, \bar{\Delta}_i, \bar{\Delta}_j, 0, 0)$
        end
        if $correlated(I(M_{t_i}), l, \bar{\Delta}_i, M_{t_j}, m, \bar{\Delta}_j, 0, 1)$ then
            $O_q = O_q \cup QObs(t_i, t_j, l, m, \bar{\Delta}_i, \bar{\Delta}_j)$
        end
        if $correlated(M_{t_i}, l, \bar{\Delta}_i, I(M_{t_j}), m, \bar{\Delta}_j)$ then
            $O_q = O_q \cup QObs(t_i, t_j, l, m, \bar{\Delta}_i, \bar{\Delta}_j, 1, 0)$
        end
    end
end
```

for the qualitative trend. If the occurrence of a qualitative trend q of topic t_i has to be matched we derive the closest occurrence of a trend in topic t_j before and after q. We interpolate the symbol q has to be matched again using Table 9.1. It is reasonable to assume that for instance if the corner symbols are $+$(increasing) and $-$(decreasing) there is a middle area in the interpolation with symbol 0 (constant). Finally, a match is reported if the ratio between the number of matches and the number of total checks is greater than a parameter Q (Line 32).

9.1.3 Generating the Diagnosis Model

As already mentioned above we follow the diagnosis principles form [Rei87]. Moreover, we use a more efficient model representation based on Horn clauses [PW03]. We extract the diagnosis model from the information collected through the recording phase and the instantiated observer using Algorithm 18. The algorithm gets the computation graph (G) and the sets of observers (O_n,O_g,O_p,O_q) and returns a set of Horn clauses M forming the diagnosis model.

The extraction of the clauses can be done straight forward using the information and relations contained in graph and the observers. The algorithm starts with an empty set of clauses (Line 1). For each node with a node observer we add a clause that states that, if a node is working correctly there should be a process for it (Lines 2-4). We use the common nomenclature that the literal $\neg AB(c)$ denoted that component c is working correctly.

Moreover, we add for each node n with a topic t' the node is publishing on and a general

Algorithm 17: $correlated(M_{t_i}, l, \bar{\Delta}_i, M_{t_j}, m, \bar{\Delta}_j)$

```
input  : $M_{t_i}$ ... the communication of topic $t_i$
input  : $l$... use $l^{th}$ value of topic $t_i$
input  : $\bar{\Delta}_i$ ... median message time difference of topic $t_i$
input  : $M_{t_j}$ ... the communication set of topic $t_j$
input  : $m$ ... use $m^{th}$ value of topic $t_j$
input  : $\bar{\Delta}_j$ ... median message time differences of topic $t_j$
output: $true$ if $t_i$ and $t_j$ are qualitatively correlated, $false$
        otherwise
$ws_i = C\bar{\Delta}_i, ws_j = C\bar{\Delta}_j$
$s_i^l = \varnothing, s_j^k = \varnothing$
foreach $v \in V_c(M_{t_i}, l)$ do
    $s = linreg(\{v' \in V_c(M_{t_i}, l) | time(v') \in$
    $[time(v) - ws_i/2, time(v) + ws_i/2]\})$
    $s_i^l = s_i^l \cup \langle time(v), s \rangle$
end
foreach $v \in V_c(M_{t_j}, m)$ do
    $s = linreg(\{v' \in V_c(M_{t_j}, m) | time(v') \in$
    $[time(v) - ws_i/2, time(v) + ws_i/2]\})$
    $s_i^m = s_i^m \cup \langle time(v), s \rangle$
end
$b_i^+ = median(\{val(s) | s \in s_i^l \wedge val(s) > 0\}),$
$b_j^+ = median(\{val(s) | s \in s_j^m \wedge val(s) > 0\}),$
$b_i^- = median(\{val(s) | s \in s_i^l \wedge val(s) < 0\}),$
$b_j^- = median(\{val(s) | s \in s_j^m \wedge val(s) < 0\})$
$q_i = trend(s_i, b_i^+, b_i^-, ws_i)$
$q_j = trend(s_j, b_j^+, b_j^-, ws_j)$
$c = 0, m = 0$
foreach $q \in q_i$ do
    $q_j^- = min_{q' \in q_j}(time(q) - time(q'))$
    $q_j^+ = min_{q' \in q_j}(time(q') - time(q))$
    if $match(q, q_j^-, q_j^+)$ then
        m = m + 1
    end
    c = m + 1
end
foreach $q \in q_j$ do
    $q_i^- = min_{q' \in q_i}(time(q) - time(q'))$
    $q_i^+ = min_{q' \in q_i}(time(q') - time(q))$
    if $match(q, q_i^-, q_i^+)$ then
        m = m + 1
    end
    c = m + 1
end
return $(m/c) > Q$
```

Algorithm 18: $generateModel(G, O)$

```
input  : $G$ ... the computation graph
input  : set of instantiated node observers $O_n$
input  : set of instantiated general observers $O_g$
input  : set of instantiated property observers $O_p$
input  : set of instantiated qualitative observers $O_q$
output: a set of clauses $M$
$M = \varnothing$
foreach $o \in O_n$ do
    $M = M \cup \{\neg AB(node(o)) \rightarrow running(node(o))\ \}$
end
foreach $n \in N$ do
    $P' = \{t' \in T | (\exists p \in P.node(p) = n \wedge t' \in topic(p)) \wedge$
    $\exists o \in O_g \wedge topic(o) = t'\}$
    foreach $t'' \in P'$ do
        $S' = \{t''' \in T | (\exists s \in S.node(s) = n \wedge$
        $t''' \in topic(s)) \wedge \exists o \in O_g \wedge topic(o) = t''\}$
        $M = M \cup \{\neg AB(n) \wedge \bigwedge_{t' \in S'} ok\_topic(t'') \rightarrow$
        $ok\_topic(t')\}$
    end
end
foreach $o \in O_p$ do
    $M = M \cup \{\neg AB(node(o)) \rightarrow$
    $ok\_prop(node(o), type(o))\}$
end
foreach $o \in O_q$ do
    $N_q = \{n \in N | \exists p \in P.node(p) = n \wedge$
    $t_1(o) \in topic(p) \vee t_2(o) \in topic(p)\}$
    $T' = \varnothing$
    if $\exists o \in O_g.t_1 = topic(o)$ then
        $T' = T' \cup t_1$
    end
    if $\exists o \in O_g.t_2 = topic(o)$ then
        $T' = T' \cup t_2$
    end
    $M =$
    $M \cup \{\bigwedge_{n \in N_q} \neg AB(n) \wedge \bigwedge_{t \in T'} ok\_topic(t) \rightarrow$
    $ok\_match(t_1(o), v_1(o), t_2(o), v_2(o))\}$
end
```

$s(q^-)$	$s(q^+)$	$\bar{s}(q)$			
		$t(q) \in$ $[t(q^-), t(q^-) + \Delta_t/3)$	$t(q) \in$ $[t(q^-) + \Delta_t/3, t(q^-) + \Delta_t/2)$	$t(q) \in$ $[t(q^-) + \Delta_t/2, t(q^-) + 2\Delta_t/3)$	$t(q) \in$ $[t(q^-) + 2\Delta_t/3, t(q^+)]$
$-$	$-$	$-$	$-$	$-$	$-$
$-$	0	$-$	$-$	0	0
$-$	$+$	$-$	0	0	$+$
0	$-$	0	0	$-$	$-$
0	0	0	0	0	0
0	$+$	0	0	$+$	$+$
$+$	$-$	$+$	0	0	$-$
$+$	0	$+$	$+$	0	0
$+$	$+$	$+$	$+$	$+$	$+$

Table 9.1: Interpolation for qualitative trend for time $t(q)$ between the times $t(q^-)$ and $t(q^+)$ where $\Delta_t = t(q^+) - t(q^-)$).

observer (check for regular communication) for t' a clause that specifies that this observer have to report $ok\_topic(t')$ if the node n works and all its subscribed topics t'' that have an observer report $ok\_topic(t'')$ as well (Line 5-13). These clauses cover the input/output relations of nodes. For each property observer we add a clause that, if a node n works all its observed properties of particular types have to be within the specified boundaries (Line 14-15). The literal $ok\_prop(n, t)$ denoted that the property of type t is ok for node n.

Finally, we specify that, if two topics t_1 and t_2 are qualitative correlated and there is a related observer the two topics have to qualitatively match (Line 27-23). Where the literal $ok\_match(t_1, l, t_2, m)$ that the l^{th} value of topic t_1 is qualitatively matching with the m^{th} value of topic t_2. For instance in a fault-free system the yaw angle reported by the odometry should have the same trend to a yaw reported by an IMU.

9.2 Multi Training Sets

We also consider more than one training sets for generating model, i.e., during model learning phase one can take more than one training sets for the same problem and robotics system. The idea came out due to the reason that it might happen during learning phase some data streams from sensors cannot fully capture the correct behavior or due to uncertainty training set captures some irregularities in data stream. Combining similar data streams from different training sets can bubble out these irregularities and produce better system behavior than only one training set. The process of combining multi data streams into one large training set is depicted in Figure 9.1. Algorithm 19 describes the process of integrating multiple training sets into one. It takes n training sets each with m data streams. Firstly all the starting and ending times from each of data in a training set is extracted. The minimum of all m start times (s_i) gives a start time

Algorithm 19: $mergeTrainingSets(\Pi)$

```
input  : $\Pi$ ... set of $n$ training sets
output: $\Gamma$ ... one training set
foreach $Train_n \in \Pi$ do
    foreach $data_m \in Train_n$ do
        $t_{min_m} = min(\{t_i | data_m = \langle v_i, t_i \rangle\})$
        $t_{max_m} = max(\{t_i | data_m = \langle v_i, t_i \rangle\})$
    end
    $t_{start_n} = min(t_{min_m})$
    $t_{end_n} = max(t_{max_m})$
end
$dif = \forall_{0<i<n}(t_{start_{i+1}} - t_{end_i})$
$\delta_t = 0$
$\Gamma = Train_1$
foreach $d_k \in dif$ do
    $\delta_t = \delta_t + d_k$
    foreach $data_m \in Train_{k+1}$ do
        $data_m = \{t_i - \delta_t | data_m = \langle v_i, t_i \rangle\}$
    end
    $\Gamma = \Gamma \cup Train_{k+1}$
end
return $\Gamma$
```

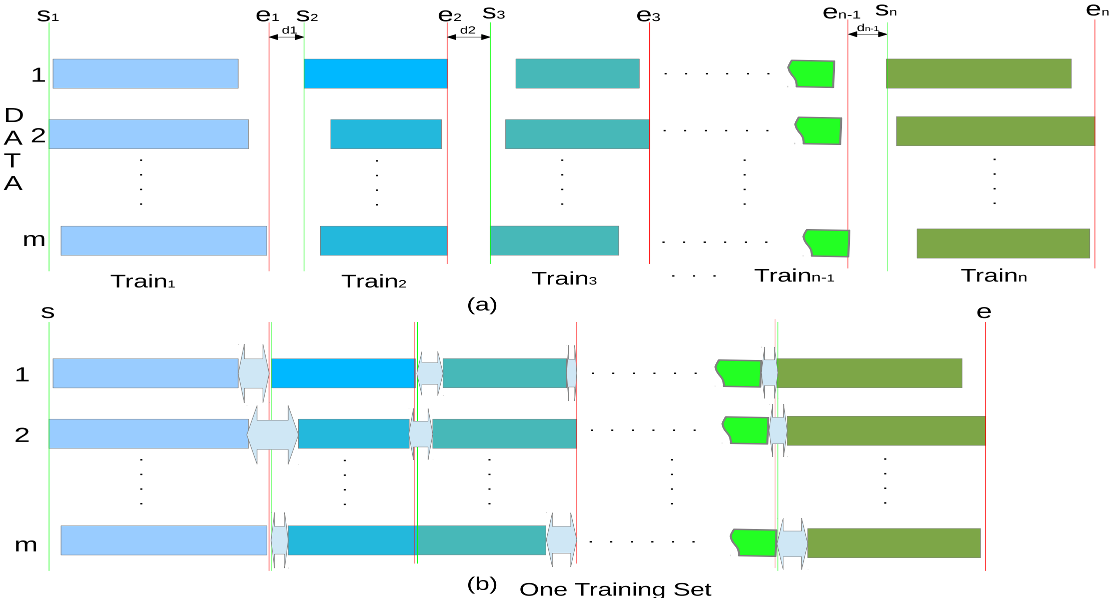


Figure 9.1: (a) Different length m data streams from n training sets. All same color data is in one training set. Every training set i has start time s_i and ending time e_i. (b) The m data streams from n training sets are combined into one large training set with one start time s and ending time e. Arrows show the gap between the data.

for a training set, and maximum of all m ending times (e_i) gives an ending time of the training set (Lines 1-8). Difference between start time of ith and ending time of i-1th training set gives time interval δ between two training sets. Therefore, each of the data streams from subsequent training set has to be shifted towards preceding training set by subtracting the interval δ. All corresponding data streams of training sets are combined into one data stream. This gives m combined data streams making one large training data sets. This training set can be used for correlation calculation between data streams.

Chapter 10

Experimental Results

In this chapter we provide experiments performed for validating our diagnosis and repair system. It also provides empirical evaluation of the generated diagnosis models through the learning process.

10.1 Diagnosis and Repair

In order to evaluate the diagnosis and repair process for both software and hardware faults we conducted experiments. Following is the evaluation set up and experimental results to evaluate diagnosis and repair process.

10.1.1 Experimental Setups

For evaluation of proposed work we used TEDUSAR robot based on Dr.Robot Jaguar mobile robot platform as shown in Figure 10.1. It contains two tracks and two articulated, tracked, independently controlled arms. The platform can move over various terrains and climb up slopes and stairs. The robot is equipped with a Hokuyo laser range finder (LRF) mounted on a leveling mechanism with 2 degrees of freedom to assure scanning in the horizontal plane for building maps. A sensor head with two degrees of freedom consisting of a thermal camera, and a Microsoft Kinect sensor is used to detect victims around the robot on the basis of body temperature and computer vision. An XSens MTi inertial measurement unit (IMU) determines the alignment of the robot. In addition, a wireless router is mounted on the robot to establish a connection to the remote operator control station, and a hardware diagnosis board [ZL13] was designed and mounted on the robot to observe power consumption behaviors of the hardware components (Chapter 8). The robot is controlled by an embedded PC running the Ubuntu Linux and control software based on the ROS framework. The operator station consists of a joystick connected to a laptop PC running an Ubuntu Linux, the joystick driver node and the visualization tool for the operator.

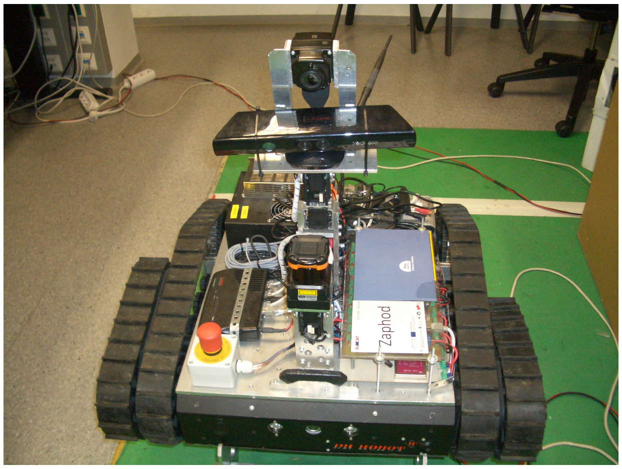


Figure 10.1: TEDUSAR search and rescue robot.

The scenario for the diagnosis example is a teleoperated exploration and mapping of an unknown environment. Figure 10.2 shows the components of the system and the communication between them. In the figure, abbreviations *ja, hu, la, jn, hn, in, lan, jtn, hm, lac, jyn* and *jys* represent *jaguar, hokuyo, laser_alignment, jaguar_node, hokuyo_node, imu_node, laser_alignment_node, jaguar_teleop_node, hector_mapping, laser_alignment_control, joy_node* and *joystick* respectively. The movement of the robot is remotely controlled by an operator using a joystick. The signals from the joystick are converted to velocity commands for the robot platform. The communication between the operator station and the robot is carried out via wireless network. Laser scans are used to generate a map of the explored space by using a flexible and scalable mapping approach [KMvSK11] with some mapping relevant issues presented in [ZSS11] . For building a map the laser range finder has to be aligned to the horizontal plane. Therefore, the posture of the robot is determined with the IMU, and the angles of the servos in the laser alignment system are set accordingly. The visualization tool Rviz is used to display the map to the operator.

We tested the diagnosis and repair system for both hardware devices as well as software nodes. The components comprise the devices and nodes introduced in Section 3.1.1. The diagnosis model and the repair domain description was created as described in Sections 6.2 and 7.1.2.1 respectively. Moreover, we set up one hardware observer (monitoring the power status of the hardware), one node observer for each software node and one general observer for each of the topics $/odom$, $/map$, $/scan$, $/imu\_data$ and $/cmd\_vel$.

Experiments were conducted for two different scenarios:

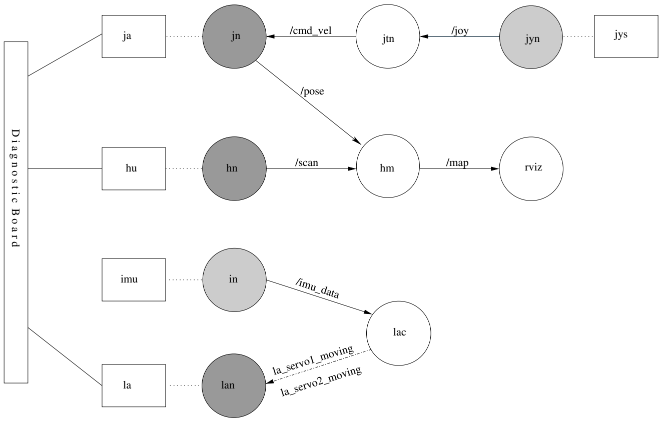


Figure 10.2: Simple control architecture for the search and rescue robot. Rectangles represent hardware modules. Gray circles represent hardware nodes. Dark gray circles represent hardware driver nodes with switchable hardware devices. White circles represent normal software nodes. Solid arrows represent publisher/subscriber communication. Dot-dashed arrows represent service calls. Dotted lines represent hardware connections.

10.1.1.1 System-Power-up scenario

In order to evaluate the correctness of the diagnosis model and the planning domain we conducted a power-up experiment. In the system-power-up scenario the whole robot system is initially switched off, and the diagnosis and repair system has to transfer it to a state ready for the mapping. In the beginning all hardware components are switched off and all software nodes are not running. The only powered hardware is the hardware diagnosis board and the PC. The only running software is the diagnosis and repair system. The diagnosis engine obtains the following diagnosis using the diagnosis model and observations from the observers:

$$\Delta_{good} = \{\}, \Delta_{bad} = \{j, h, la, lan, lac, hm, jn, hn, in, jtn, jyn\}$$

The diagnosis and observations are then used by the planner to plan and invoke proper action. Figure 10.3 shows for the system-power-up scenario the diagnosis and planner results for all the hardware and software components. At the top of the figure the sequence of the repair actions is depicted. Initially all hardware and software components are abnormal. The planner first powers up the *laser_alignment* hardware by invoking a *power_up* action. After *laser_alignment* powers up and becomes normal the planner continues to power up *hokuyo*. After all, the hardware is powered up the planner continues with the software nodes and starts them one by one by calling *start_node* actions. The planner starts *laser_alignment_node* then *hokuyo_node, hector_mapping, jaguar_teleop_node, jaguar_node, laser_alignment_control* and *imu_node*. All the components

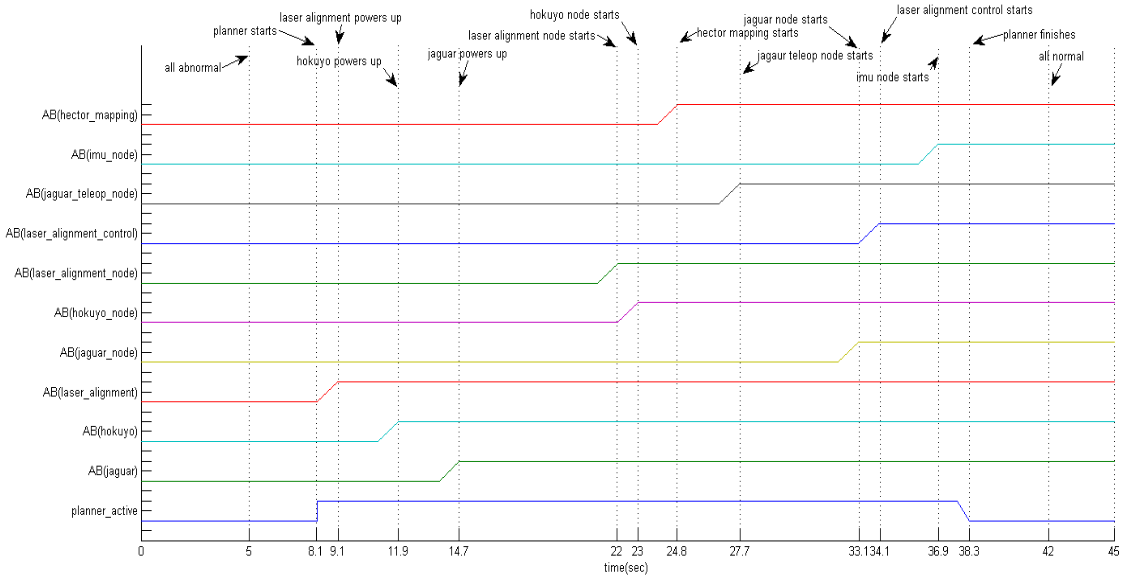


Figure 10.3: Behavior of the diagnosis and repair system for the System-Power-up scenario.

become normal at the end of the repair plan. As shown in Figure 10.3 the planner starts at time 8.1 seconds and finishes at 38.3 taking total of 30.2 seconds to bring all components into normal state. The results clearly show that our system is able to obtain the correct diagnosis and to execute a repair plan to set the system's hardware and software into a correct state.

10.1.1.2 Device-Shut-down scenario

In the second experiment we evaluated how the system reacts to a dynamic fault occurring at run-time. In the device-shut-down scenario the system is operating correctly at the beginning. Then suddenly one hardware device goes down. In this scenario *jaguar* hardware is switched off. As a result the *jaguar_node* gets disconnected from *jaguar* and stops publishing odometry data. So the required sequence of actions should be *power_up* the *jaguar*, *stop_node* and *start_node* for *jaguar_node*. When *jaguar* is switched off the diagnosis engine adds it to the faulty components in the diagnosis. The planner takes this diagnosis and invokes *power_up* action for the *jaguar*. When *jaguar* was powered up the planner kills (the still running) *jaguar_node* by invoking *stop_node* action. After *jaguar_node* was stopped successfully the planner invokes *start_node* action to start it again. Figure 10.4 shows this scenario with related diagnoses and sequence of the repair actions. Please note that in this scenario the planner is invoked twice. First, it is invoked for the *power_up* action for the *jaguar*. Second, it is invoked for the two actions *stop_node* and *start_node* for *jaguar_node*. Then *jaguar_node* becomes normal. After this the whole system is normally running again.

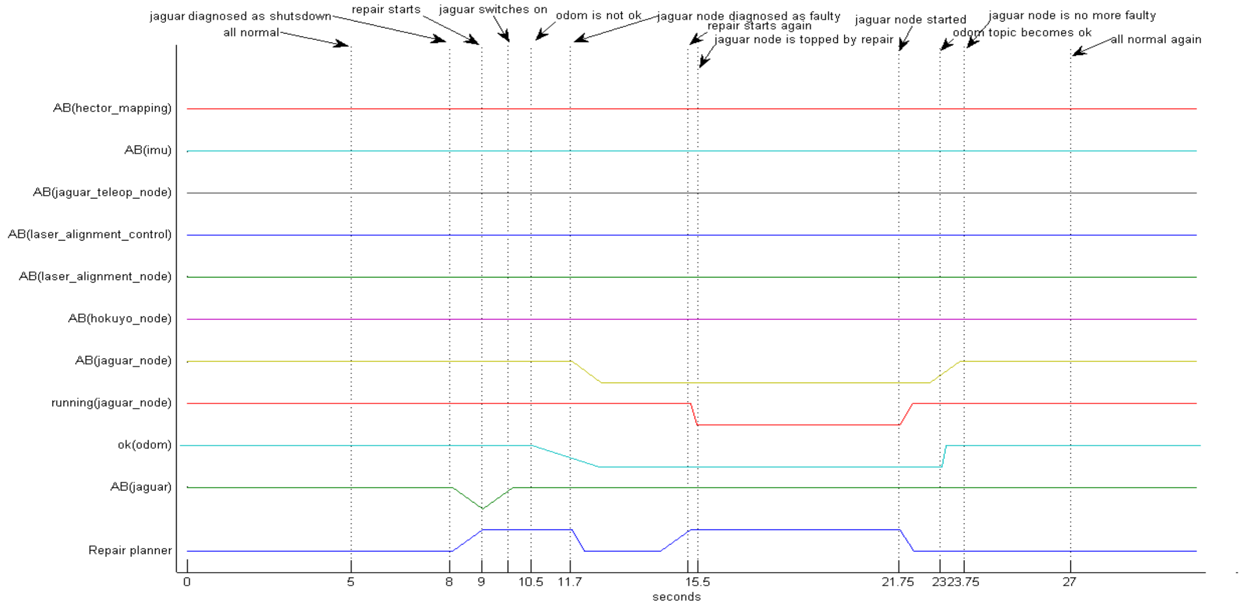


Figure 10.4: Behavior of the diagnosis and repair system for the Device-Shut-down scenario.

10.2 Model Evaluation

In order to evaluate the proposed diagnosis model learning we conducted a series of experiments. We used a mobile robot in a teleoperated mapping scenario. Using the proposed learning approach and a fault-free task execution we obtained different diagnosis models. Different models had been generated to evaluate the influence of the parameter in the generation algorithm. These automated generated diagnosis models were used for diagnosis in a run where different faults were injected. We evaluated if the system reports any false positives or negatives. These results are a metric for the quality of the generated diagnosis models.

10.2.1 Experimental Setup

For the evaluation we used a Pioneer DX3 robot (Figure 10.5) equipped with a Sick LMS 200 laser scanner and a XSense MTi IMU. The robot was controlled by a standard ROS installation. For mapping the open-source SLAM implementation gmapping[1] was used. The robot was teleoperated while mapping the corridors in a building of a size of $20m \times 14m$. The control

[1] see http://www.openslam.org/gmapping

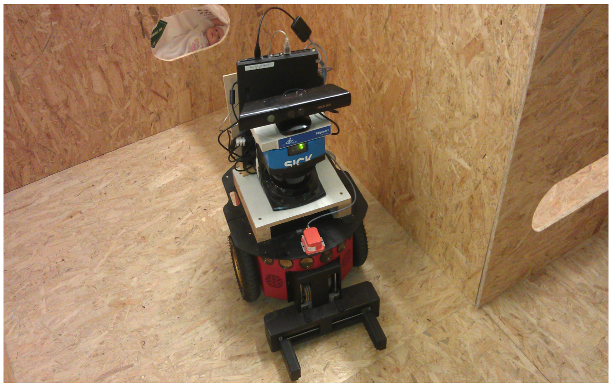


Figure 10.5: Pioneer 3DX robot equipped with Laser and IMU sensors.

software of the robotics system used a number of ROS nodes communicating with each other on their topics. The communicational graph with nodes and the topics is shown in Figure 10.6. For the fault-free task execution we ran the robot without injecting any faults. In the test runs we injected different faults we expected the system to identify using the generated models. The faults we injected are:

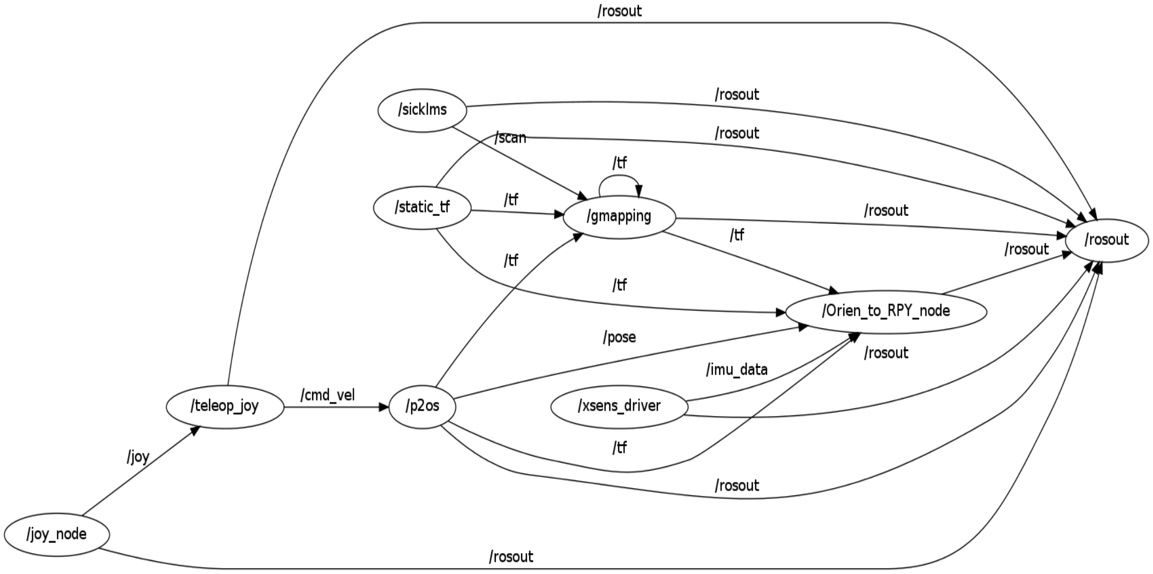


Figure 10.6: The computation graph used in the validation experiments.

- **Odometry Slip :** It is achieved by driving robot over slippery floor. In our case we had four slippery spots on the floor, the robot was expected to slip during navigation while passing over them.

Model	Parameters		Observers					
	Q	α	$BiQObs$	$GObs$	$NObs$	$PObs_{cpu}$	$PObs_{mem}$	Total
$M1$	0.9	10	0	2	7	7	7	23
$M2$	0.9	2.5	0	15	7	7	7	36
$M3$	0.9	0.5	0	20	7	7	7	41
$M4$	0.8	10	1	2	7	7	7	24
$M5$	0.8	2.5	1	15	7	7	7	37
$M6$	0.8	0.5	1	20	7	7	7	42
$M7$	0.5	10	6	2	7	7	7	29
$M8$	0.5	2.5	6	15	7	7	7	42
$M9$	0.5	0.5	6	20	7	7	7	47

Table 10.1: Experimented models for different learning parameters with different number of observers: binary qualitative (BiQObs), general (GObs), node (NObs), CPU property ($PObs_{cpu}$), and memory property ($PObs_{mem}$) observers.

- **Increasing CPU usage :** A node's cpu usage is increased using additional dummy thread. For example, the `CPU` usage of the node *p2os* was modelled as 4.11554554815% and we increased it upto 30%.
- **Increasing Memory usage :** Using additional thread for allocating additional memory. For example, the memory usage of the node *gmapping* was modelled as $1040626.0418 Bytes$ and we increased it to its double.
- **Changing Frequency :** Increased message frequency with additional dummy messages. For example, the frequency of the topic $/pose$ was modelled $10.0212312737 Hz$ and it was increased by 5 times.
- **Crashing Node :** We crashed node (simply killing the node). For example, we killed $teleop\_joy$ node because it was controlling the teleoperation of the robot.

10.2.2 Model Validation

For validation of the model generated we conducted a fault-free execution of the mapping task and recorded data. Using the proposed approach this data were transferred into diagnosis models. Using different values for the parameters Q and α we generated nine different models denoted M_1 to M_9. We used the different models to evaluate the influence of the manually set parameters on their quality. The hypothesis is the higher we select Q the less binary qualitative observer are in the model. We assume there is an optimal value for Q between high values that may miss faults because observer are missing and low values reporting too many faults because of irrelevant observers. The same assumption is made for α and the number of general observers. Table 10.1 shows the different models and the number of observers generated. In order to validate the

			Models								
	fault	target	1	2	3	4	5	6	7	8	9
CR	slip	4	0	0	0	3	0	1	2	4	3
	cpu	2	2	2	1	2	1	1	1	1	2
	mem	2	2	2	2	2	2	2	2	2	2
	frq	2	0	2	1	0	2	1	1	2	1
	crash	2	2	2	2	2	2	2	2	2	2
	sum	12	6	8	6	9	7	7	8	11	10
FN	slip	0	4	4	4	1	4	3	2	0	1
	cpu	0	0	0	1	0	1	1	1	1	0
	mem	0	0	0	0	0	0	0	0	0	0
	frq	0	2	0	1	2	0	1	2	0	1
	crash	0	0	0	0	0	0	0	0	0	0
	sum	0	6	4	6	3	5	5	5	1	2
FP	slip	0	0	0	0	1	0	2	1	0	0
	cpu	0	0	0	0	0	1	0	0	0	0
	mem	0	0	0	0	0	0	0	0	0	0
	frq	0	0	0	1	0	2	3	0	0	3
	crash	0	0	0	0	0	0	0	0	0	0
	sum	0	0	0	1	1	3	5	1	0	3

Table 10.2: Diagnoses reported for injected faults using different diagnosis models. CR denotes correctly reported diagnoses. FN and FP denote false negatives respectively false positives.

different models we conducted the same mapping task as used for the generation of the models. We conducted 2 runs for each model where the diagnosis system was active using that model. In each run we injected artificial faults. In each run we drove the robot twice over slippery floor, increased once the memory and cpu usage of a node, increased the frequency of messages on a topic, and killed once a node. All faults were injected and retracted in an sequential order to avoid interferences between the faults which can negatively affect the evaluation. For each run we noted if faults were reported correctly, faults were not reported (false negatives) or wrong faults were reported (false positives). The results are depicted in Table 10.2.

According to the results diagnosis model M_8 best models the true behavior of the system. It reported nearly all injected faults correctly. Moreover, it reported no false positives and one false negative. This is a satisfactory result but contradicts a little bit the hypothesis stated above. We see that models M_7 to M_9 showed a higher number of correct reported faults (in particular for the slippage/sensor). This shows that a low Q value which introduced more binary qualitative observer is important. Table 10.3 shows the generated observers and the values correlated. It shows that the relevant $BiQOb_5$ and $BiQOb_6$ which relate the two individual sensors odometry and IMU were introduced only for low Q values. Moreover, the table shows that for instance observer 6 makes perfect sense as it relates the yaw measurements of both sensors. Please note that this relation was automatically extracted from the training data. Apparently, the values of these observers have a weaker correlation than values origin from the single sensor odometry (e.g., $BiQOb_1$). Please note that the number of node and property observers was not affected by

BiQObs	Q	Value 1	Value 2
$BiQOb_1$	0.8	Intg(pose.twist.twist.angular.z)	Yaw(pose)
$BiQOb_2$	0.5	pose.pose.position.y	Intg(pose.pose.position.x)
$BiQOb_3$	0.5	pose.pose.position.y	Intg(Yaw(pose))
$BiQOb_4$	0.5	pose.twist.twist.angular.z	Yaw(pose)
$BiQOb_5$	0.5	Intg(pose.twist.twist.angular.z)	Yaw(imu)
$BiQOb_6$	0.5	Yaw(imu)	Yaw(odom)

Table 10.3: Values used by the binary qualitative observers. Intg. denotes the integration of a value. The value names correspond to the ROS message structure.

Q or α which led to an almost constant recognition rate of property or node faults for all models.

Moreover, the reaction to injected faults to the message frequency showed that M_1, M_4, and M_7 were not able to detect them. Because of the low α value only 2 general observers were generated for 2 very much regular topics. Summarizing, models like M_8 and M_9 with a low Q value and a moderate α value perform best. These models are able to reliably report the true faults which shows that the model learning works if Q and α are selected properly.

10.2.3 Significant Model

Using the z-test of significance (discussed in Sec. 3.6 Chapter 3) we try to find out significance difference between the models. Total 12 models were learned, and each model was tested against same 12 injected faults. Following table shows the number of the correctly identified faults (true positive). From the data given in the table we find that model M_8 is the best of all and model M_3 is the worst one. We consider these two models M_3 and M_8 for the significance test.

Model/Error	1	2	3	4	5	6	7	8	9	10	11	12	μ	σ
M_1	0	0	0	0	1	1	1	1	0	0	1	1	0.500	0.500
M_2	0	0	0	0	1	1	1	1	1	1	1	1	0.666	0.471
M_3	0	0	0	0	1	0	1	1	1	0	1	1	0.500	0.500
M_4	1	1	1	0	1	1	1	1	0	0	1	1	0.750	0.433
M_5	0	0	0	0	1	0	1	1	1	1	1	1	0.583	0.493
M_6	1	0	0	0	1	0	1	1	1	0	1	1	0.583	0.493
M_7	1	1	0	0	1	0	1	1	1	0	1	1	0.666	0.471
M_8	1	1	1	1	1	0	1	1	1	1	1	1	0.916	0.276
M_9	1	0	1	1	1	1	1	1	1	0	1	1	0.833	0.372

We formulate the *null* and *alternative* hypotheses as:

$\mathcal{H}_0$: $M_3 = M_8$

$\mathcal{H}_a$: $M_3 \neq M_8$

using z-test equation:

$$z_{3,8} = \frac{\mu_8 - \mu_3}{\sqrt{\frac{\sigma_3}{n_3} + \frac{\sigma_8}{n_8}}} \tag{10.1}$$

$$\Rightarrow \frac{0.916 - 0.500}{\sqrt{\frac{0.500}{12} + \frac{0.276}{12}}} \tag{10.2}$$

$$\Rightarrow \frac{0.416}{\sqrt{0.064666}} \tag{10.3}$$

which gives $z_{3,8}$ equal to 1.63586 that means $-1.96 \leq z_{3,8} \leq 1.96$ resulting not to reject *null hypotheses* ($\mathcal{H}_0$), therefore, the difference between model M_3 and M_8 is not found significant. Although from the table data it is clear that the model M_8 provides much better results than the model M_3 but the significant test concludes with no difference between them. The main reason is that the number of samples are very less. Typically for z-test the condition $|n_i$+$n_j| \geq 30$ should hold but in our case n_i+n_j is 24 for every two models i and j. Therefore, the z-test demands for increase in the number of samples in our case.

Chapter 11

Conclusion and Future Work

The work presented in this paper provides four contributions. First, the proposed system combines automated diagnosis for robot systems with automated repair. Second, the system incorporates software and hardware into the diagnosis and repair process. Thirdly, the proposed system is based on popular robotic framework Robot Operating System (ROS) and extends its existing diagnostics. Finally, the system also provides an initiative towards automatic learning of diagnosis model of a robotics system for model-based diagnosis.

The proposed system comprises of different modules: (1) set of observers, (2) a diagnosis model server, (3) a model-based diagnosis engine, (4) a planner-based repair engine, (5) a hardware diagnostic board. The observers monitor the state of hardware, software nodes and their topics. They include diagnostic observer ($DObs$), general observer ($GObs$), node observer ($NObs$), qualitative observer ($QObs$), binary qualitative observer ($BiQObs$), hardware observer ($HObs$), property observer ($PObs$), and interval observer ($IObs$). Every observer performs a specified monitoring task and publishes its monitored information on the ROS topic $/observations$. The output of the observers is a list (Θ) of the first-order logic (FOL) literals , e.g., $\neg ok(topic), running(node), matched(val_1, val_2), on(component)$. All the observations from the observers provide an observed behavior of the robotics system.

The diagnosis model server provides a model of the correct behavior of the robotics system at run-time. The model contains logical rules (Horn clauses) which describe the predicted behavior of each component of the robotics system. Each rule in the model uses a special predicate AB ”ABnormal”, i.e., $AB(m)$ states a faulty component m, $\neg AB(n)$ states a working component n. The diagnosis model server enables the diagnosis system to make changes in the model at-run time without halting the rest of diagnosis and repair process. The model from the diagnosis model server provides predicted behavior of the robotics system.

The model-based diagnosis engine follows the model-based diagnosis approach ([Rei87]). It takes observed behavior in the form of observations from the observers, and the predicted behavior in the form of set of the Horn clauses from the diagnosis model server and finds whether there is any discrepancies between the observed and the predicted behavior of the robotics system. If any discrepancy is encountered means a fault is detected. The diagnosis engine then derives root causes of the detected fault in order to localize the fault. The output of the diagnosis engine is diagnosis (Δ) which is a set of faulty components (Δ_{bad}) and working components (Δ_{good}), i.e., $\Delta = \{\Delta_{bad}, \Delta_{good}\}$.

The planner-based repair engine takes the diagnosis Δ from the diagnosis engine and Θ from the observers and converts them into a planning problem. The repair engine uses Planning Domain Definition Language (PDDL) for generating problem description for the planner. It uses Graphplan in order to generate the plan ($\mathcal{P}$) of repair actions (ρ). For every repair action ρ with some parameter δ the repair engine invokes repair action server. The process of invoking repair action servers is sequential, i.e., repair engine invokes a repair action server for the first ρ in $\mathcal{P}$ and waits until it completes and then it takes the next repair action (ρ).

For hardware diagnosis and repair we use a diagnostic board which contains 10 channels. It the diagnosis and repair system for hardware monitoring and reconfiguration. The hardware components, e.g., laser sensor, camera, router, etc., are connected to the board. The board has the capability of measuring current, voltage, and status of each component. Moreover, the board can automatically power up and down certain component. We developed a simple text-based protocol through which board can be accessed using TCP/IP.

Like every model-based diagnosis system the presented system requires a diagnosis model which reflects the correct behavior of the robotics system. To acquire and learn the diagnosis model online we consider a model generation and learning process. A fault free run of the robotic system enables us to record the model which is used during diagnosis process. The model generation approach extracts the information with minimum of user interaction. Moreover, using statistical learning correlations between data in the computation are extracted which can be used in the diagnosis model. An experimental evaluation with different training and test runs show that the approach is able to generate a valid diagnosis model to make system able to correctly detect and localize faults.

In future work we will further investigate the various interactions within the diagnosis and repair system that have a huge impact on the stability and quality of the diagnosis and repair process. Moreover, we intend to increase the capabilities of the system for monitoring and modeling. We have not yet compared our architecture with other existing architectures, therefore, as future work we consider to compare the system with other integrated systems like the Livingston and LAAS architectures [MNPW98b, ACF^{+}98].

Although, the approach needs only a minimum set of user-specified parameters for model learning there is still need for future work in order to automate the estimation of crucial parameters such as Q and α for correlation and regularity in the data respectively. Moreover, it is also needed to investigate how the approach scales with the size of the robot system. In particular, a more efficient implementation and more intelligent treatment of the value correlations have to be done. Finally, the automated modeling currently limits to the diagnosis model. It will also be very interesting to learn the repair model as well. Possibly, a combination of fault diagnosis and analyzing the user's reaction to faults can lead to such models. Finally, we demand that the learning runs are fault-free. This is a strong requirement of our system, therefore, we will also investigate how much influence the faults pose on the learning phase and its impact on the quality of the learned model.

The planner-based repair engine generates the plan and executes it without waiting for the effects of the repair action to appear. A better strategy for the future work would be to wait until the effects have been established. For instance it might take longer for a node's output topics to become correct than simply the time to restart the node.

The communicational system of a robotics system might also contain conditional and triggering nature of communication between the components. We also intend to include Multiple observer (MObs) in order to observer conditional communication between the components of a robotics system.

We do not claim that the presented architecture is complete but we believe that this can be a good initiative towards a complete automated diagnosis and repair systems.

Bibliography

[AA04] Brian Randell A. Avižienis, Jean-Claude Laprie. Dependability and its threats: A taxonomy. IFIP International Federation for Information Processing Volume 156, pp 91-120, 2004.

[AASB$^+$06] R. Alami, A. Albu-Schaeffer, A. Bicchi, R. Bischoff, R. Chatila, A. De Luca, A. De Santis, G. Giralt, J. Guiochet, G. Hirzinger, F. Ingrand, V. Lippiello, R. Mattone, D. Powell, S. Sen, B. Siciliano, G. Tonietti, and L. Villani. Safe and dependable physical human-robot interaction in anthropic domains: State of the art and challenges. Procceedings IROS Workshop on pHRI - Physical Human-Robot Interaction in Anthropic Domains, October, 2006.

[ACF$^+$98] R. Alami, R. Chatila, S. Fleury, M. Ghallab, and F. Ingrand. An architecture for autonomy. *International Journal of Robotics Research*, 17:315–337, 1998.

[Asi42] I. Asimov. Runaround story. In *Astounding Science Fiction-science fiction magazine*, March, 1942.

[BBdK82] J. S. Brown, R. R. Burton, and J. de Kleer. Pedagogical, natural language and knowledge engineering techniques in sophie i, ii and i. In *Intelligent Tutoring System, Academic Press New York, 227-282*, 1982.

[BCNB07] E. Balaban, H. N. Cannon, S. Narasimhan, and L. S. Brownston. Model-based fault detection and diagnosis system for nasa mars subsurface drill prototype. In *Aerospace Conference, 2007 IEEE*, Big Sky, MT, March 3-10, 2007.

[Bea71] R. V. Beard. Failure accomodation in linear systems through self-reorganisation. In *Tech.Rep. MVT-71-1, Man Vehicle Lab., Cambridge, Mass*, 1971.

[BF97] A. L. Blum and M. L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90:281–300, 1997.

[BGI+09] S. Bensalem, M. Gallien, F. Ingrand, I. Kahloul, and N. Thanh-Hung. Designing autonomous robots. In *Robotics and Automation Magazine, IEEE (Volume:16, Issue: 1)*, March, 2009.

[BGVB10] J. F. Broenink, M. A. Groothuis, P. M. Visser, and M. M. Bezemer. Model-driven robot-software design using template-based target descriptions. In *ICRA 2010 Workshop on Innovative Robot Control Architectures for Demanding (Research) Applications: How to modify and enhance commercial controllers, Anchorage, Allaska, USA*, pages 73–77, May 2010.

[BH98] K. Balakrishnan and V. Honavar. Intelligent diagnosis systems. In *Journal of Intelligent Systems, Vol. 8, Nos. 3-4*, 1998.

[BHSW07] Mathias Brandstötter, Michael Hofbaur, Gerald Steinbauer, and Franz Wotawa. Model-based fault diagnosis and reconfiguration of robot drives. In *2007 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, San Diego, CA, USA, 2007.

[BMS+05] C. Burghart, R. Mikut, R. Stiefelhagen, T. Asfour, H. Holzapfel, P. Steinhaus, and R. Dillmann. A cognitive architecture for a humanoid robot:a first approach. In *5th IEEE-RAS International Conference on Humanoid Robots, pp: 357-362*, Tsukuba, Dec, 2005.

[Boo54] G. Boole. An investigation of the laws of thought. In *Dover, New York*, 1854.

[BS04] Bert Bredeweg and Peter Struss. Current Topics in Qualitative Reasoning. *AI Magazine*, 24(4):13–16, 2004.

[CDTnn] L. Console, D. T. Dupré, and P. Torasso. A theory of diagnosis for incomplete causal models. In *Proc. IJCAI, pages 13111317*, Detroit, August 1989. Morgan Kaufmann.

[CGD12] D. Crestani and K. Godary-Dejean. Fault tolerance in control architectures for mobile robots: Fantasy or reality? In *7th National Conference on Control Architectures of Robots, (CAR2012) Nancy, France*, 2012.

[CM03] J. Carlson and R. R. Murphy. Reliability analysis of mobile robot. In *In Proceedings of the 2003 IEEE International Conference on Robotics and Automation, (ICRA-2003)*, Taipei, Taiwan, September 14-19, 2003.

[CP99] J. Chen and R. J. Patton. Robust model-based fault diagnosis for dynamic systems. In *Kluwer Academic Publisher*, 1999.

[CPR00] L. Console, C. Picardi, and M. Ribaudo. Diagnosis and diagnosability analysis using pepa. In *14th European Conference on Artificial Intelligence (ECAI-2000), pages13135*, Berlin,Allemagne, 2000.

[CPVG05] R. Ceballos, S. Pozo, C. D. Valle, and R. M. Gasca. An integration of FDI and DX techniques for determining the minimal diagnosis in an automatic way. In *MICAI 2005, LNAI 3789, pp. 10821092*, 2005.

[CS01] G.M. Coghill and Q. Shen. Towards the specification of models for diagnosis of dynamic systems. In *Artificial Intelligence in Communications, 14, (2)*, 2001.

[CTng] L. Console and P. Torasso. Integrating models of correct behavior into abductive diagnosis. In *European Conference on Artificial Intelligence, pages 160166*, Detroit, August, 1990. Pitman Publishing.

[Dav94] Neil James Davies. The perfomance and scalability of parallel systems. In *Ph.D thesis, Faculty of Engineering, University of Bristol*, December, 1994.

[DBS04] R. Dillmann, R. Becher, and P. Steinhaus. ARMAR II - a learning and cooperative multimodal humanoid robot system. In *International Journal of Humanoid Robotics, vol. 1(1), pp. 143155*, Springer-Verlag Berlin, Heidelberg, 2004.

[DGA94] P. Dario, E. Guglielmelli, and B. Allotta. Robotics in medicine. In *Intelligent Robots and Systems '94. 'Advanced Robotic Systems and the Real World', (IROS-94).*, September 12-16, 1994.

[DH88a] R. Davis and W. Hamscher. Model-based reasoning: Troubleshooting. In *H.E. Shrobe (Ed.), Exploring Artificial Intelligence, Chapter 8, pp. 297346*, Morgan Kaufmann, San Mateo, CA, 1988.

[DH88b] R. Davis and W. C. Hamscher. Model-based reasoing: Troubleshooting. In *AI Memos (1959 - 2004)*, July, 1988.

[Die07] A. Diekmann. *Empirische Sozialforschung - Grundlagen, Methoden, Anwendungen*. Rowohlt, 2007.

[dK76] J. de Kleer. Local methods for localizing faults in electronic circuits. In *Massachusetts Institute of Technology Artificial Intelligence Laboratory, AIM-394*, November, 1976.

[dKW87] J. de Kleer and B. C. Williams. Diagnosing multiple faults. In *Artificial Intelligence, 32(1):97130*, 1987.

[Dru12] K. Drum. Chart of the day: Our robot overlords will take over soon. In *Mother-Jones Blog*, April 17, 2012.

[ELP02] A. R. Eisenman, C. C. Liebe, and R. Perez. Sun sensing on the mars exploration rovers. In *IEEE Aerospace Conference Proceedings, (Vol: 5)*, 2002.

[EM] V. J. Easton and J. H. McColl. *Statistics Glossary version 1.1*. http://www.stats.gla.ac.uk/steps/glossary/.

[FGL87] K. S. Fu, R. C. Gonzalez, and C. S. G. Lee. Robotics-control, sensing, vision and intelligence. In *McGraw-Hill Book Company*, 1987.

[FHN72] R. E. Fikes, P. E. Hart, and N. J. Niisson. Learning and executing generalized robot plans. In *Artificial Intelligence, 3:251-288*, 1972.

[FL03] Maria Fox and Derek Long. Pddl2.1: an extension to pddl for expressing temporal planning domains. In *Planning Domains. University of Durham, UK*, 2003.

[FN71a] R. E. Fikes and N. J. NHsson. Strips: A new approach to the application of theorem proving to problem solving. In *Artificial Intelligence ,2:189-208*, 1971.

[FN71b] R.E. Fikes and N.J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. In *Artificial Intelligence, 2, (1971) 189208*, 1971.

[FSW99] G. Friedrich, M. Stumptner, and F. Wotawa. Model-based diagnosis of hardware designs. In *Artificial Intelligence 111 (1999) 339*, 1999.

[Fuj00] M. Fujita. Digital creatures for future entertainment robotics. In *IEEE International Conference on Robotics and Automation, (ICRA '00)*, April 24-28, 2000.

[Fuj11] M. Fujita. Autonomous robot dancing synchronized to musical rhythmic stimuli. In *6th Iberian Conference on Information Systems and Technologies (CISTI)*, June 15-18, 2011.

[Gen84] M. R. Genesereth. The use of design descriptions in automated diagnosis. In *Artificial Intelligence Volume 24, Issues 13, Pages 411436*, December, 1984.

[GMC04] S. Gentil, J. Montmain, and C. Combastel. Combining FDI and AI approaches within causal-model-based diagnosis. In *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics, (Volume:34 , Issue: 5)*, Oct, 2004.

[GNT04] M. Ghallab, D. Nau, and P. Traverso. Automated Planning Theory and Practice. In *Morgen Kauffman Publishers, Elsevier*, 2004.

[GWHH10a] R. Golombek, S. Wrede, M. Hanheide, and M. Heckmann. Learning a probabilistic self-awareness model for robotic systems. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2010.

[GWHH10b] R. Golombek, S. Wrede, M. Hanheide, and M. Heckmann. A method for learning a fault detection model from component communication data in robotic systems. In *Seventh IARP Workshop on Technical Challenges for Dependable Robots in Human Environments*, Toulouse, France, 2010.

[HABJ08] D. A. Harrison, R. Ambrose, B. Bluethmann, and L. Junkin. Next generation rover for lunar exploration. In *IEEE Aerospace Conference*, Big Sky, MT, March 1-8, 2008.

[Hil90] Mark D. Hill. What is scalability? In *ACM SIGARCH Computer Architecture News, Volume 18 Issue 4*, pages 18–21, December, 1990.

[HKEW10] M. Heerink, B. Kröse, V. Evers, and B. Wielinga. Assessing Acceptance of Assistive Social Agent Technology by Older Adults: the Almere Model. In *International Journal of Social Robotics http://dx.doi.org/10.1007/s12369-010-0068-5*, 2010.

[HKSW07] M. Hofbaur, J. Köb, G. Steinbauer, and F. Wotawa. Improving robustness of mobile robots using model-based reasoning. In *Journal of Intelligent and Robotic Systems, 48(1):3754*, 2007.

[IB97] R. Isermann and P. Ballé. Trends in the application of model-based fault detection and diagnosis of technical processes. In *Control Engineering Practice Volume 5, Issue 5*, page Pages 709719, May 1997.

[IGI11] G. Infantes, M. Ghallab, and F. Ingrand. Learning the behavior model of a robot. In *Journal Autonomous Robots archive Volume 30 Issue 2, Pages 157-177*, February, 2011.

[Ise97] R. Isermann. Supervision, fault-detection and fault-diagnosis methods an introduction. In *Control Engineering Practice Volume 5, Issue 5*, page Pages 639652, May 1997.

[ISH$^+$03] J. F. Bell III, S. W. Squyres, K. E. Herkenhoff, J. N. Maki, H. M. Arneson, D. Brown, S. A. Collins, A. Dingizian, S. T. Elliot, E. C. Hagerott, A. G. Hayes, M. J. Johnson, J. R. Johnson, J. Joseph, K. Kinch, M. T. Lemmon, R. V. Morris, L. Scherr, M. Schwochert, M. K. Shepard, G. H. Smith, J. N. Sohl-Dickstein, R. J. Sullivan, W. T. Sullivan, and M. Wadsworth. Mars exploration rover athena panoramic camera (pancam) investigation. In *JOURNAL OF GEOPHYSICAL RESEARCH, VOL. 108, NO. E12, 8063, doi:10.1029/2003JE002070*, 2003.

[Kal12] M. Kalech. Diagnosis of coordination failures: a matrix-based approach. In *Autonomous Agents and Multi-Agent Systems, Volume 24, Issue 1, pp 69-103*, January, 2012.

[KBC$^+$98] Craig Knoblock, Anthony Barrett, Dave Christianson, Marc Friedman, Chung Kwok, Keith Golden, Scott Penberthy, David E Smith, Ying Sun, and Daniel Weld. PDDL- the planning domain definition language. *AIPS-98 Competition Committee*, 78(4):1–27, 1998.

[KKR13] Eliahu Khalastchi, Meir Kalech, and Lior Rokach. Sensor fault detection and diagnosis for autonomous systems. In *The 12th International Conference on Autonomous Agents and Multiagent Systems (AAMAS2013)*, 2013.

[KMvSK11] S. Kohlbrecher, J. Meyer, O. von Stryk, and U. Klingauf. A flexible and scalable slam system with full 3d motion estimation. In *Proc. IEEE International Symposium on Safety, Security and Rescue Robotics (SSRR)*, 2011.

[KSW08] Alexander Kleiner, Gerald Steinbauer, and Franz Wotawa. Towards Automated Online Diagnosis of Robot Navigation Software. In *First International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR 2008)*, volume 5325 of *Lecture Notes in Computer Science*, pages 159–170. Springer, 2008.

[KSW09] Alexander Kleiner, Gerald Steinbauer, and Franz Wotawa. Using qualitative and model-based reasoning for sensor validation of autonomous robots. In *Twentieth International Workshop on Principles of Diagnosis (DX 2009)*, Stockholm, Sweden, 2009.

[LC05] H. Liu and G. M. Coghill. A model-based approach to robot fault diagnosis. In *Knowledge-Based Systems journal, Vol, 18, pp: 225233*, 2005.

[LCI$^+$04] B. Lussier, R. Chatila, F. Ingrand, M. O. Killijian, and D. Powell. On fault tolerance and robustness in autonomous systems. In *In Proceedings of the third*

IARP/IEEE-RAS/EURON Joint Workshop on Technical Challenge for Dependable Robots in Human Environments, Manchester, GB, September 7-9, 2004.

[LK08] A. LIGEZA and J. M. KOŚCIELNY. A new approach to multiple fault diagnosis: A combination of diagnostic matrices, graphs, algebraic and rule based models. the case of two-layer models. In *international Journal of Mathematics and Computer Science, Vol. 18, No. 4, 465476*, 2008.

[LLC$^+$05] B. Lussier, A. Lampe, R. Chatila, J. Guiochet, F. Ingrand, M. O. Killijian, and D. Powell. Fault tolerance in autonomous systems: How and how much? In *IN PROCEEDINGS OF THE 4TH IARP/IEEE-RAS/EURON JOINT WORKSHOP ON TECHNICAL CHALLENGE FOR DEPENDABLE ROBOTS IN HUMAN ENVIRONMENTS*, pages 16–18, 2005.

[LMS$^+$12] P. Lepej, J. Maurer, G. Steinbauer, S. Uran, and S. Zaman. An integrated diagnosis and repair architecture for ROS-Based Robot Systems. In *Twenty Third International Workshop on Principles of Diagnosis (DX 2012)*, Great Malvern, UK, 2012.

[MAVL06] A. Monteriú, P. Asthan, K. Valavanis, and S. Longhi. Experimental validation of a real-time model-based sensor fault detection and isolation system for unmanned ground vehicles. In *Proc. of 14th Mediterranean Conference on Control Automation*, Ancona, Italy, June, 2006.

[MAVL07] A. Monteriú, P. Asthan, K. Valavanis, and S. Longhi. Model-based sensor fault detection and isolation system for unmanned ground vehicles: Experimental validation (part ii). In *IEEE International Conference on Robotics and Automation*, Roma, Italy, April 10-14, 2007.

[MCK$^+$89] S. Minton, J. G. Carbonell, C. A. Knoblock, D. R. Kuokka, O. Etzioni, and Y. Gil. Explanation-based learning: A problem-solving perspective. In *Journal of Artificial Intelligence 40(13): 63118*, 1989.

[MH04] M. RYAN M. HUTH. Logic in computer science. Modelling and reasoning about systems. In *Cambridge University Press The Edinburgh Building, Cambridge CB2 8RU, UK*, 2004.

[Mit99] O. Mitsushige. Autonomous underwater vehicle operations beneath coastal sea ice. In *IEEE International Conference on Robotics & Automation*, Detroit, Michigan, May, 1999.

[MNPW98a] N. Muscettola, P. P. Nayak, B. Pell, and B. C. Williams. Remote agent: To boldly go where no ai system has gone before. In *Artificial Intelligence, 103(1-2):548*, 1998.

[MNPW98b] Nicola Muscettola, P. Pandurang Nayak, Barney Pell, and Brian C. Williams. Remote Agent: to boldly go where no AI system has gone before. *Artificial Intelligence*, 103(1-2):5 – 47, 1998.

[MR91] D. McAllester and D. Rosenblitt. Systematic nonlinear planning. In *In Proceedings of the Ninth National Conference on Artificial Intelligence, 634639. Menlo Park, Calif.: American Association for Artificial Intelligence*, 1991.

[MS96] A. Malik and P. Struss. Diagnosis of dynamic systems does not necessarily require simulation. In *Proceedings of the Seventh International Workshop on Principles of Diagnosis*, 1996.

[MSS95] A. Malik, P. Struss, and M. Sachenbacher. Qualitative modeling is the key a successful feasibility study in automated generation of diagnosis guidelines and failuer mode and effects analysis for mechatronic car subsystems. In *Proceedings of the Sixth International Workshop on Principles of Diagnosis*, 1995.

[MTT06] Roberto Micalizio, Pietro Torasso, and Gianluca Torta. On-line monitoring and diagnosis of a team of service robots: A model-based approach. *AI Communications*, 19(4):313–340, December 2006.

[Mur00] R. R. Murphy. Introduction to AI robotics. In *The MIT Press Cambridge, Massachusetts London, England*, 2000.

[Ogo09] O. Ogorodnikova. How safe the human-robot coexistence is? theoretical presentation. Acta Polytechnica Hungarica, Vol 6, No 4, 2009.

[Pal01] G. K. Palshikar. Consistency-based diagnosis. In *Dr. Dobbs Journal, vol. 26, no. 3, pp. 50-56*, March 2001.

[Ped89] E. P. D. Pednault. ADL: exploring the middle ground between strips and the situation calculus. In *Proceedings of the first international conference on Principles of knowledge representation and reasoning, Pages 324-332*, Ancona, Italy, 1989.

[Pel08] Damien Pellier. Pddl4j, 2008. http://sourceforge.net/projects/pddl4j.

[PG04] B. Pulido and C. A. Gonzalez. Possible conflicts: a compilation technique for consistency-based diagnosis. In *Systems, Man, and Cybernetics, Part B: Cybernetics, IEEE Transactions on (Volume:34 , Issue: 5)*, Oct, 2004.

[PK06] L. E. Parker and B. Kannan. Adaptive causal models for fault diagnosis and recovery in multi-robot teams. In *Proceedings of IEEE International Conference on Intelligent Robots and Systems (IROS-06)*, 2006.

[PKSF12] A. J. Plueddemann, A. L. Kukulya, R. Stokey, and Lee Freitag. Autonomous underwater vehicle operations beneath coastal sea ice. In *IEEE/ASME Transactions on Mechatronics, , Volume 17 (1)*, Jan 9, 2012.

[Poo88] D. Poole. Representing knowledge for logic-based diagnosis. In *Proc. International Conference on Fifth Generation Computing Systems, 1282-1290*, Tokyo, 1988.

[Poo89] D. Poole. Normality and faults in logic-based diagnosis. In *Proc. IJCAI, pages 13041310*, Detroit, August, 1989.

[Poo94] D. Poole. Representing diagnosis knowledge. Annals Math. and Artificial Intelligence vol. 11, nos. 14, pp. 3350, 1994.

[Pop12] World Robot Population. Early Warning: Rational Analysis of Global Civilization Risk. http://earlywarn.blogspot.co.at/2012/04/global-robot-population.html, April 17, 2012.

[PW03] Bernhard Peischl and Franz Wotawa. Model-based Diagnosis Or Reasoning From First Principles. *IEEE Intelligent Systems*, 18(3):32–37, 2003.

[QCG$^+$09] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng. ROS: an open-source robot operating system. In *ICRA Workshop on Open Source Software, (ICRA-2009)*, 2009.

[Rei87] Raymond Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57 – 95, 1987.

[REW06] P. Robertson, R. Effinger, and B. Williams. Autonomous robust execution of complex robotic missions. In *Proceedings of the 9th International Conference on Intelligent Autonomous Systems*, University of Tokyo, Tokyo, Japan, March 7-9, 2006.

[RW05] P. Robertson and B. Williams. A model-based system supporting automatic self-regeneration of critical software. In *IFIP/IEEE International Workshop on Self-Managed Systems and Services*, 2005.

[RWH11] S. Richter, M. Westphal, and M. Helmert. Lama 2008 and 2011. In *In Seventh International Planning Competition (IPC 2011), Deterministic Part, pp. 50-54*, 2011.

[Sim99] S. Simani. Model-based fault diagnosis in dynamic systems using identification techniques. In *Ph.D Thesis, University of Modena and Reggio Emilia*, 1999.

[SMV07] G. Sandini, G. Metta, and D. Vernon. The iCub Cognitive Humanoid Robot: An open-system research platform for enactive cognition. In *Book: 50 years of artificial intelligence, Pages 358-369, ISBN:3-540-77295-2 978-3-540-77295-8*, Springer-Verlag Berlin, Heidelberg, 2007.

[SMW06] Gerald Steinbauer, Martin Mörth, and Franz Wotawa. Real-time diagnosis and repair of faults of robot control software. In *International RoboCup Symposium*, volume 4020 of *Lecture Notes in Computer Science*, Osaka, Japan, 2006. Springer.

[SNS04] R. Siegwart, I. R. Nourbakhsh, and D. Scaramuzza. Introduction to autonomous mobile robots. In *The MIT Press Cambridge, Massachusetts London, England*, 2004.

[SRA+11] Amir H. Soltanzadeh, Amir H. Rajabi, Arash Alizadeh, Golnaz Eftekhari, and Mehdi Soltanzadeh. RoboCupRescue 2011 - Robot League Team AriAnA (Iran). In *RoboCup 2011 - RoboCup Rescue Robot - Team Description Papers*, 2011.

[Ste06] G. Steinbauer. Intelligent and robust control of autonomous mobile robots. In *Ph.D Thesis, Institute for Software Technology, TU Graz*, Oct, 2006.

[SW05] G. Steinbauer and F. Wotawa. Detecting and locating faults in the control software of autonomous mobile robots. In *16th International Workshop on Principles of Diagnosis, pages 13-18*, Monterey, USA, 2005.

[Tak07] T. Takahashi. Robot designer or robot creator. In *16th IEEE International Conference on Robot & Human Interactive Communication*, August 26-29, 2007.

[TS03] R. H. Taylor and D. Stoianovici. Medical robotics in computer-integrated surgery. In *EEE TRANSACTIONS ON ROBOTICS AND AUTOMATION, VOL. 19, NO. 5*, OCTOBER 14, 2003.

[UAW04] UAW. Uaw health and safety department: Review of robot injuries - one of the best kept secrets. Proceed. National robot conference, Ypsilanti, Michigan, October. 2004.

[VJ99] C. Vijay and H. John. Optimization methods for logical inference. In *John Wiley and Sons Inc ISBN 9780471570356*, 1999.

[VRK03] V. Venkatasubramanian, R. Rengaswamy, and S. N. Kavuri. A review of process fault detection and diagnosis part ii: Qualitative models and search strategies. In *Computers and Chemical Engineering, Vol 27, issue 3, 313-326*, March 15, 2003.

[VRKY03] V. Venkatasubramanian, R. Rengaswamy, S. N. Kavuri, and K. Yin. A review of process fault detection and diagnosis part iii: Process history based methods. In *Computers and Chemical Engineering, Vol 27, issue 3, 327-346*, March 15, 2003.

[VRYK03] V. Venkatasubramanian, R. Rengaswamy, K. Yin, and S. N. Kavuri. A review of process fault detection and diagnosis part i: Quantitative model-based methods. In *Computers and Chemical Engineering, Vol 27, issue 3, 293-311*, March 15, 2003.

[VVGG$^+$04] V. Verma, I. Verma, Geoff G. Gordon, R. Simmons, and S. Thrun. Particle Filters for Rover Fault Diagnosis. In *IEEE Robotics &amp; Automation Magazine special issue on Human Centered Robotics and Dependability*, 2004.

[Web08] Jörg Weber. Model-based runtime diagnosis of the control software of mobile autonomous robots. In *Ph.D thesis at IST Institute for Software Technology Graz University of Technology*, Graz, Austria, Jan, 2008.

[Wel99] D. S. Weld. Recent Advances in AI Planning. In *AI Magazine, Volume 20, Number 2,*, 1999.

[WHC$^+$06] J. Wright, F. Hartman, B. Cooper, S. Maxwell, J. Yen, and J. Morrison. Driving on Mars with RSVP. In *IEEE Robotics and Automation Magazine, (Volume:13 , Issue: 2)*, June, 2006.

[ZKH$^+$01] F. Zhao, X. Koutsoukos, H.Haussecker, J. Reich, and P. Cheung. Distributed monitoring of hybrid systems: A model-directed approach. *International Joint Conf on Artificial Intelligence (IJCAI01)*, page Seattle, August 2001.

[ZL13] S. Zaman and P. Lepej. MBD counterpart Diagnostic Board: Its Hardware, Protocol, Software, Simulator, and Application. In *22nd International Workshop on Robotics in Alpe-Adria-Danube Region (RAAD 2013)*, Portoroz, Slovenia, September 11-13, 2013.

[ZL14] S. Zaman and P. Lepej. ROS-Based Diagnostic Board for Detecting and Repairing Hardware Faults in Autonomous Mobile Robots. In *Proceedings of 1st Interna-*

tional conference on Robotics (iCREATE 2014), IEEE, Islamabad, Pakistan, April, 2014.

[ZS13a] S. Zaman and G. Steinbauer. Automatic Modeling and Observers Generation for Model-Based Diagnosis System for ROS-Based Robotic Systems. In *Austrian Robotics Workshop (ARW 2013)*, Wien, Austria, May 23-24, 2013.

[ZS13b] S. Zaman and G. Steinbauer. Automated Generation of Diagnosis Models for ROS-based Robot Systems. In *24-th International Workshop on Principles of Diagnosis (DX-2013)*, Jerusalem, Israel, October 1-to-4, 2013.

[ZSM$^+$13] S. Zaman, G. Steinbauer, J. Maurer, P. Lepej, and S. Uran. An Integrated Model-Based Diagnosis and Repair Architecture for ROS-Based Robot Systems. In *IEEE International Conference on Robotics and Automation (ICRA-2013)*, Karlsruhe, Germany, 2013.

[ZSS11] S. Zaman, W. Slany, and G. Steinbauer. ROS-based Mapping, Localization and Automatic Navigation using Pioneer 3-DX Robot and their relevant Issues. In *Saudi International Electronics, Communications and Photonics Conference), IEEE*, Riyadh, Saudi-Arabia, 2011.

Index