2013 IEEE International Conference on Robotics and Automation (ICRA)
Karlsruhe, Germany, May 6-10, 2013

An Integrated Model-Based Diagnosis and Repair Architecture for ROS-Based Robot Systems

Safdar Zaman, Gerald Steinbauer, Johannes Maurer, Peter Lepej and Suzana Uran

Abstract— **Autonomous robots are artifacts that comprise a significant number of heterogeneous hardware and software components and interact with dynamic environments. Therefore, there is always a chance of faults at run-time that negatively affect the reliability of the system. In this paper we present a novel diagnosis and repair architecture for ROS-based robot systems. It is an extension to the existing ROS diagnostics stack and follows a model-based diagnosis and repair approach. In the paper we discuss the integrated diagnosis and repair architecture in detail. Moreover, we show its application to an example robot system and report first experimental results. The presented work provides three major contributions: a combination of diagnosis and repair, the integration of hardware and software, and the integration into ROS.**

I. Introduction

The motivation for the work presented in this paper origins from two fundamental observations. First, autonomous robots are artifacts that comprise a significant number of hardware and software components 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 and implementation flaws or shortage of testing the involved modules, robots are frequently subject to faults and unexpected behavior. Moreover, the potentially non-deterministic nature of the interaction of the robot with its environment leads to additional faults and unexpected situations. Obviously, such problems negatively affect the performance and the autonomy of a robot. Therefore, a truly autonomous and dependable robot has to have the capability to actively cope with such phenomena in an automated way. Second, an increasing number of research groups around the globe use the Robot Operating System (ROS) [1] as a standard framework for their development of robot systems. ROS already provides a simple fault diagnosis system. However it is mainly limited to monitoring hardware modules and code execution.

In order to tackle above observations we developed an advanced diagnosis and repair architecture which is based on ROS and is able to inter-operate with the existing diagnostics stack. In the development we utilized our experience in diagnosis systems and results of previous work of other colleagues and us, such as robotics software diagnosis [2], [3], [4], hardware repair [5] and sensor validation [6].

S. Zaman, G. Steinbauer and J. Maurer are with the Institute for Software Technology, Graz University of Technology, Graz, Austria. {szaman,steinbauer,jmaurer}@ist.tugraz.at. P. Lepej and S. Uran are with Faculty of Electrical Engineering and Computer Science, University of Maribor, Maribor, Slovenia. {peter.lepej, suzana.uran}@uni-mb.si

There are three major contributions of this paper. First, the proposed and implemented system integrates diagnosis and repair for robot systems in one running system. In particular active repair is hardly seen in such systems. Second, the system integrates software and hardware components in the diagnosis and repair process. This allows to approach more complex systems comprising hardware and software. Finally, the system is based on the already widely used ROS framework and its standards. This fact is beneficial for the acceptance of the system by other robot developers or researchers. The system is publicly available on an open-source basis[1].

II. ROS and its Diagnostics Stack

ROS is an open-source, meta-operating system that primarily runs on Unix-based platforms. The main idea is to have a distributed system of programs called *nodes* that are individually designed and loosely coupled at run-time. The framework provides functionality for hardware abstraction, low-level device control, and message passing between nodes. ROS 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. 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.

ROS provides a simple fault diagnostic system. It is mainly designed for monitoring hardware modules and code execution. The diagnostics system collects information from hardware drivers and robot hardware for analysis, troubleshooting, and logging[2]. The diagnostics stack is built around the */diagnostics* topic. The hardware drivers and devices publish standard diagnosis messages on the */diagnostics* topic with the device names, states and specific key-value pairs.

A complete diagnostic system should easily be able to diagnose all aspects of a system such as hardware problems, problems in software modules, or other difficulties occurring at run-time. The existing ROS diagnostic stack is not complete in that sense. It mainly deals with hardware drivers

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

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

978-1-4673-5643-5/13/$31.00 ©2013 IEEE

or publishers related to hardware that publish data on the */diagnostics* topic.

Moreover, the existing system requires that the module or node to be diagnosed has to publish data on the */diagnostics* topic. This means that if a hardware node does not provide such data it cannot be included into the diagnostics system until one alters the source code of the node and adds a publisher for that information. This can be a problem if one is unable to alter the code because it might not be available or complicated to extend. Using a more general diagnosis concept can prevent a developer from having to alter nodes while still being able to supervise a node to a certain extent. The same applies to pure software nodes or services.

The ROS diagnostics system follows a more linear diagnosis approach where the relation of a symptom to a root cause is quite clear, which is fine for simple hardware devices. If the system and the interaction between various modules become more complex it might be hard to connect an undesired behavior observed to its real root cause. For instance, wrong pose estimation can have several root causes and might be not only tied to a malfunctioning laser scanner. Using a more general observer and diagnosis concept, the proposed diagnostic system will be able to cover a much broader scope of systems, faults and interactions.

Finally, the existing diagnostics system does not have any capabilities for autonomous repair. 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.

III. Running Example

Searching victims after an earthquake is a highly risky operation. Buildings are in danger of collapse and further unpredictable earth shakes may appear. Robots can be used to reduce the risk of first responders in such scenarios. The task of a search and rescue robot is to look for victims, to build a map and to mark positions of victims in the map so that rescue teams could efficiently plan and execute rescue actions. High risk environments demand a high level of rescue robot autonomy, therefore self diagnosis and self repair abilities of a search and rescue robot are desirable. This makes a search and rescue robot a perfect example for verification of the novel diagnosis and repair architecture proposed in this paper.

The running example is inspired by the RoboCup Rescue competition. Goal of the competition is to build search and rescue robots that explore an unknown environment reproducing a collapsed building, to generate a map of the explored area, to search for simulated victims, to mark the position of these victims in the map and to examine vital signs of victims. The joint RoboCup Rescue team of the Graz University of Technology (TUG) and the University of Maribor (FERI) built a search and rescue robot for the RoboCup Rescue competition. The TEDUSAR robot is shown in Figure 1.

The robot is based on the Dr.Robot Jaguar mobile robot platform with 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 FLIR PathFindIR 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 was designed and mounted on the robot to observe power consumption behaviors of hardware components (for a detailed description see Section IV-A). The robot is controlled by an embedded PC running an 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.

Fig. 1. TEDUSAR search and rescue robot.

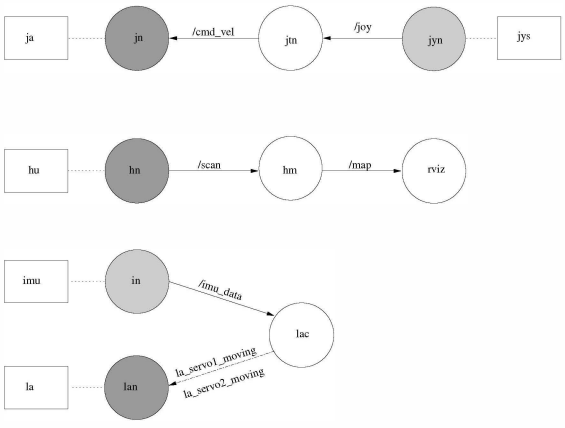

Fig. 2. Simple control architecture for the search and rescue robot. Rectangles represent hardware modules. Gray circles represent hardware nodes. Dark gray circles represent hardware driver nodes with switchable hardware devices. White circles represent normal software nodes. Solid arrows represent publisher/subscriber communication. Dot-dashed arrows represent service calls. Dotted lines represent hardware connections.

The scenario for the diagnosis example is a teleoperated exploration and mapping of an unknown environment. Figure 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 [7]. 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.

IV. System Architecture

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.

Following these requirements we developed the architecture of the diagnosis and repair system shown in Figure 3. The architecture resembles the five modules already mentioned in the introduction: (1) a diagnosis model server, (2) a set of system observers, (3) a model-based diagnosis engine, (4) a planner-based repair engine and (5) a hardware diagnosis and repair board. Each module provides a particular functionality which is described below in more detail. The modules communicate using three types of communication:

a) **Standard Topics and Messages** *:* All observers publish their observations in form of an array of strings containing first-order logic sentences to 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 the proper range. Diagnosis results are published to the topic */diagnosis* using a diagnosis message containing sets of working and faulty components (*bad*,*good*). The diagnosis message may contain multiple diagnoses. A dedicated diagnosis model message contains a set of propositions and a set of clauses. Please refer to Section IV-C for more details on the diagnosis model and the diagnosis process.

b) **Repair Actions***:* All repair actions in the system are implemented using the ROS action server/client architecture[3]. 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("jaguar")` represents the ground repair action for shutting down the power supply for the robot base.

c) **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.

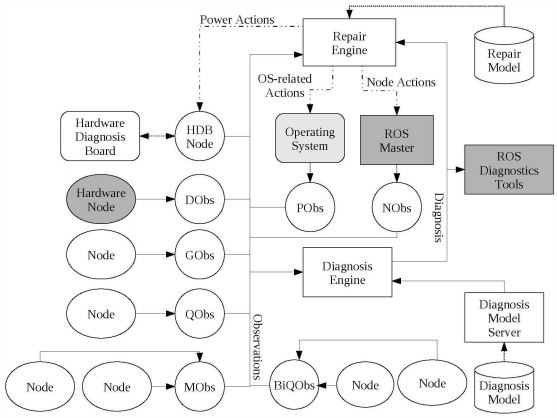

Fig. 3. System Architecture Overview. Solid lines represent ROS topic communication. Dash-dotted lines represent ROS action calls. Dashed lines represent other means of interaction. Dark gray modules depict existing ROS modules. The lighter gray module represents the underlying operating system. White modules are the novel modules of the architecture.

A. Diagnostic 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 hardware diagnosis board and a hardware diagnosis observer. Together they are able to gather information like current measurements and power supply detection of individual components of the system. The knowledge that a particular component draws a certain amount of current or that a device is indeed powered can assist the diagnosis process. Previous works like [8], [9] also used on-line diagnosis for current measurements in a distributed embedded hybrid system, for example a Xerox DC265 printer.

Moreover, the 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 ten individual channels that can be switched on and off individually. Moreover, individual current measurements can be taken. Figure 4 shows an overview of the hardware diagnosis board. For the running example seven channels are used for the following

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

hardware components: (1) robot base, (2) PC, (3) sensor pan-tilt unit, (4) laser alignment system, (5) thermal camera, (6) laser range finder and (7) Kinect.

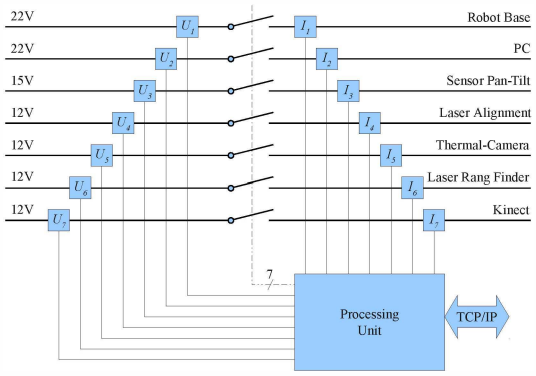

Fig. 4. Overview over hardware diagnosis board and robot connections (search and rescue robot example).

Similar systems called power management systems [10] had been already used in robot systems. Such systems allow user to manually switch on or off individual on-board systems and perform current and voltage monitoring. In addition to this, our system is also capable to do these steps autonomously. The system is designed to repair and reconfigure the robot system without touching it.

The central part of the hardware diagnosis 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 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. Moreover, it provides commands to change the power configuration of individual channels. Finally, the board can be configured in a way that at power up a defined set of channels is automatically on. For the running example we configured the board to start automatically the PC to allow bootstrapping during system start. This configuration allows us to automatically start the board, the PC and the ROS-based software parts of the architecture.

B. Observers

Observers are general software entities implemented as ROS nodes that monitor particular aspects of the system. There are different kinds of observers supervising different aspects or properties. The result of the observation is published on the */observations* topic as a first order logic (FOL) sentence. The sentence is a representation of the observed property and its status. Observations are published as a set of strings. Each string represents a single FOL sentence. Currently only simple literals are supported, e.g., `"ok(temp_cpu1)"` or `"not ok(temp_cpu1)"`. More complex expressions using for instance quantifiers can be easily integrated. Please note that obviously the representation and expressiveness of observations have to match with the capabilities of the used diagnosis approach. Currently we provide the following observer types:

- **Diagnostic Observer (DObs)** is the 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. This allows to observe the communication behavior of nodes without altering the node's code. For this general case this is done by checking the frequency of messages. If it observes the right frequency $f(t)$ for a topic t ($|f(t) - \mu_f(t)| < \sigma_f(t)$) it submits $ok(t)$, $\neg ok(t)$ otherwise, where the frequency is measured using a time window ω.
- **Node Observer (NObs)** observes the running state of software nodes. It submits $running(n)$ if node n actually runs, $\neg running(n)$ otherwise.
- **Multiple Observer (MObs)** is similar to a general observer but subscribes to several topics. Therefore, more complex communication behaviors such as conditioned or triggered communication are observed. For details about the process please refer to [3].
- **Qualitative Observer (QObs)** observes the abstract trend of a particular value in the message of a topic. It reports if value v actually increases ($inc(v)$), decreases ($dec(v)$) or stays constant ($const(v)$). 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 [6].
- **Binary Qualitative Observer (BiQObs)** observes match between abstract trends of two particular values $v1$ and $v2$. It issues observation $matched(v1, v2)$ if abstract trend of $v1$ is similar to of $v2$ otherwise $\neg matched(v1, v2)$.
- **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 and has to be individually implemented according to the actual needs.

C. Diagnosis Engine

The diagnosis engine takes observations in the form of FOL sentences and an abstract diagnosis model (system description) to generate a diagnosis, a set of components that are faulty and a set of components that are still working properly. The literal $\neg AB(m)$ denotes that module m is working properly, while $AB(m)$ denotes that module m shows a faulty behavior.

The diagnosis engine follows the principles of model-based diagnosis presented in [11]. The approach uses an abstract model that defines the correct behavior and some 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 diagnoses that resolve the contradiction, i.e., explain the misbehavior. The engine locates that component or set of components that is the root cause for the contradiction. For details on the diagnosis step we refer the reader to [11]. The diagnosis engine uses an open-source Java-based implementation of the approach [12].

Each single observation o_t that occurs at time t on the topic */observations* is integrated into an observation set OBS. $OBS = OBS \oplus o_t$, where

$$S \oplus o = \begin{cases} (S \backslash \neg L) \cup L & o = L \\ (S \backslash L) \cup \neg L & o = \neg L \end{cases} \quad L \text{ is an atom.}$$

The diagnosis engine obtains the diagnosis model at run-time from a model server. This allows making changes in the diagnosis model during run-time. For a detailed description of the model see next subsection.

The results of the diagnosis engine are published on the */diagnosis* topic. As described earlier the diagnosis message may comprise several diagnoses built up by sets of working ($good$) and faulty (bad) components. The union of both sets is equal to the set of all system components ($good \cup bad = C$) for each individual diagnosis. Apart from this it also generates diagnostic messages compatible with the ROS diagnostics stack.

D. Diagnosis Model Server

The diagnosis model server is an action server which provides the diagnosis model. The diagnosis model is stored as a *YAML* file (similar to XML) and contains five sections: (1) a string that denotes the proposition that represents the AB predicate, (2) the same for $\neg AB$, (3) a string for a prefix denoting a negative literal, (4) a set of all propositions, (5) a set of clauses that defines the diagnosis model. We have to describe the diagnosis model this way as the currently used diagnosis engine only supports propositions and Horn clauses.

For the modeling of a robot system we use the following notations:

- the set of all components C is divided into the disjunctive sets N (ordinary nodes), HN (hardware driver nodes), SN (hardware driver nodes with switchable hardware), SH (switchable hardware devices) and H (other hardware devices). The term switchable means the related hardware can be controlled by the hardware diagnosis board.
- a function $out : C \rightarrow 2^T$ returning the output topics of a component, T being the set of all topics in the system
- a function $in : T \times C \rightarrow 2^T$ returning the input topics of a component that influence an output
- a function $hardware : SN \rightarrow 2^{SH}$ returning the hardware related to a hardware driver node with switchable hardware
- a function $related : V \times V \rightarrow \{True, False\}$ V being set of values, its $True$ when both trend of both values matches.
- a function $value : T \rightarrow 2^V$ returning the output values V of a topic.

For the running example $HN = \{in, jyn\}$, $SH = \{ja, hu, la\}$, $H = \{js, imu\}$, $SN = \{jn, hn, lan\}$ and $N = \{jtn, hm, rviz, lac\}$.

We add the following clauses to the system description:

1) for each $h \in SH$ add: $\neg AB(h) \rightarrow on(h)$
2) for each $s \in SN$ and each $t \in out(s)$ add: $\neg AB(s) \wedge \bigwedge_{h \in hardware(s)} \neg AB(h) \rightarrow ok(t)$
3) for each $h \in HN$ and each $t \in out(s)$ add: $\neg AB(h) \rightarrow ok(t)$
4) for each $n \in N$ and each $t \in out(n)$ add: $\neg AB(n) \wedge \bigwedge_{i \in in(t,s)} ok(i) \rightarrow ok(t)$
5) for each $n \in C \backslash \{H, SH\}$ add: $\neg AB(n) \rightarrow running(n)$
6) for each $n_1, n_2 \in C \backslash \{H, SH\}$ and $t_1 \in out(n_1), t2 \in out(n_2)$ and $v_1 \in value(t_1), v2 \in value(t_2)$, and $related(v1, v2)$ for $v1 \neq v2$ add: $\neg AB(n_1) \wedge \neg AB(n_2) \rightarrow matched(v_1, v_2)$

Please note that this is a first minimalistic diagnosis model. Obviously, based on the complexity of the system more observations and clauses can be added. Clause 1 states that a hardware device should be powered to work correctly. Clause 2 states that if a hardware node and its related hardware components work correctly, the output of the node has to be ok. Simple hardware nodes produce correct output if they work correctly (clause 3). Clause 4 states that ordinary nodes produce proper output if the node itself works correctly and all relevant inputs are correct. Clause 5 states the fact that correctly working software nodes have to run. Last clause 6 states that two correctly running nodes produce proper outputs and their required abstract trends of the value they publish are matching.

The diagnosis engine is automatically invoked every second to be reactive to chances in the system. Please note that in case the complete system is working correctly no diagnoses are published. However, one might think of more suitable triggers like some rules based on the observations.

E. Repair Engine

In order to describe and enable repair actions, we model the repair as a planning domain. The repair engine takes current observations and diagnoses 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 current implementation in the case of multiple diagnosis always a repair plan for the first diagnosis is obtained.

The repair planning problem for a diagnosis Δ and a set of observations O comprises the following parts:

- an initial state $I = O \cup \bigcup_{c \in \Delta_{bad}} AB(c) \cup \bigcup_{c \in \Delta_{good}} \neg AB(c)$
- a set of repair actions $\mathcal{A}$ with action preconditions $pre(a)$ and effects $\mathit{eff}(a)$ for the individual actions a
- a goal state $G = \bigcup_{c \in C} \neg AB(c)$

The initial state is the union of the actual observations and the state of all components. The goal of a repair plan is that all components work correctly again.

We use the widely recognized Planning Domain Definition Language (PDDL) to represent the domain and problem definitions for the planning [13]. The description of a planning problem in PDDL comprises two parts: (1) domain description and (2) 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 the open-source Java-based package *pddl4j* [14]. All the 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 power up and shutdown a hardware component h. Examples of action descriptions are shown here:

```
(:action power_up
 :parameters (?c)
  :precondition (and (not (on ?c))(bad ?c))
  :effect (and (on ?c)(good ?c))
)

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

The first action description states that the planner can power up a component c only if c is off and faulty. The effect of the action is that component c is powered and works correctly.

The problem description is simply a PDDL description of the initial state I and the goal G. Within the repair engine we use a Java-based GraphPlan implementation to find a plan (sequence of repair actions) [15]. Once the planner has found a valid repair plan it starts the execution of the sequence of repair actions. The execution of actions is triggered by an invocation of the appropriate action server. Because of the standard signature for repair action servers (a unique name and a list of parameter strings) a matching between the planner and the ROS-based system is easy. Please note that the planner simply waits for the completion message of the called 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 the time to restart the node.

V. Adaptation to New Robots

The proposed diagnosis and repair system is a general system and can easily be adapted to new robot systems. As it is itself a ROS-based system the diagnosis and repair system can easily be applied to ROS-based systems. If one implements bridges for the observation and the actions the systems not based on ROS can benefit too. For a new system the only thing required is that the diagnosis model is adapted accordingly. The clauses given in the model section can be used as a starting point. If new repair actions are necessary they can be easily added to the domain description of the planner. Also the set of observers can be easily extended if necessary. The system provides already templates for observer development. Even the proposed hardware diagnosis board can be easily integrated in a new system by simply routing component's power lines via a channel of the board. Overall the proposed diagnosis system has the capability to be easily adapted to a new system.

VI. Experimental Results

In order to evaluate the function and the performance of the proposed system we conducted preliminary experiments using the example robot. 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 III. The diagnosis model and the repair domain description was created as described in Section IV. 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 each for the topics $/odom$, $/map$, $/scan$, $/imu\_data$ and $/cmd\_vel$. Experiments were conducted for two scenarios:

A. 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 5 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

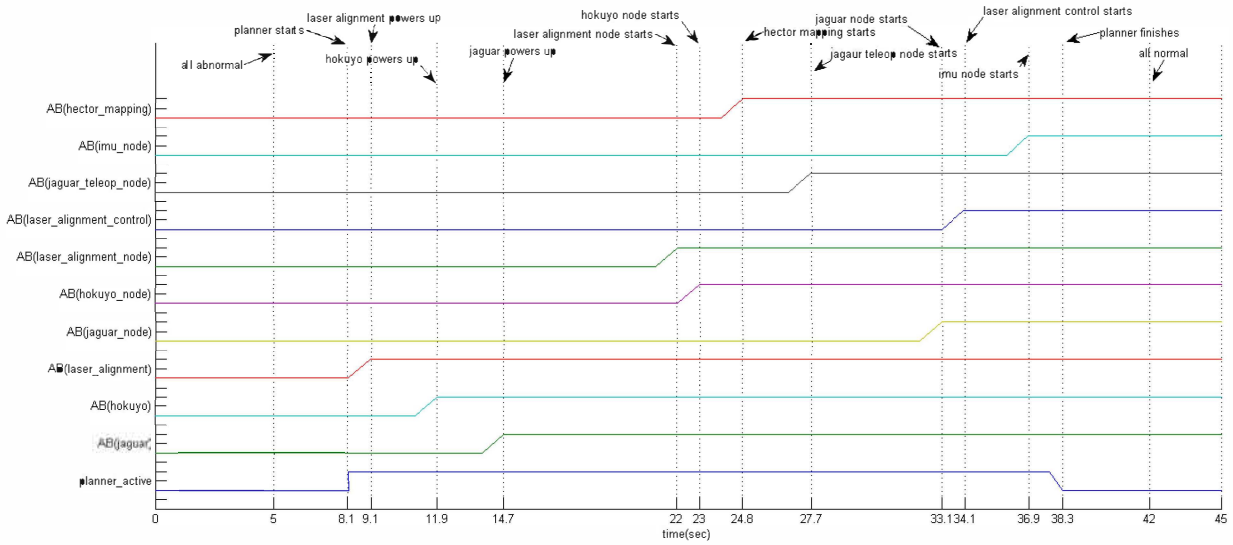

Fig. 5. Behavior of the diagnosis and repair system for the System-Power-up scenario.

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. First 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 become normal at the end of the repair plan. As shown in Figure 5 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.

B. 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. 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 killed (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 6 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, making whole the system running normally again.

VII. Conclusion and Future Work

The work presented in this paper provides three 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. Finally, the proposed system is based on and extends the popular Robot Operating System (ROS).

The proposed system comprises several types of observers, a diagnosis engine, a repair engine and a hardware diagnosis board. Observers monitor the state of hardware, nodes and topics. The diagnosis engine follows the model-based diagnosis approach and takes a diagnosis model along with observations to derive diagnoses. The repair engine uses a PDDL-based domain and problem description to obtain a proper repair plan. A novel hardware diagnosis board allows for hardware monitoring and reconfiguration. Preliminary experimental results show that the proposed diagnosis and repair system is able to detect faults in a robot system comprising software and hardware. Moreover, it is able to bring back a system to a normal state.

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 will increase the capabilities for monitoring and modeling. Finally, we have to compare the system to other integrated systems like the Livingston and LAAS architectures [16], [17].

VIII. Acknowledgments

The work has been partly funded by the European Fund for Regional Development (EFRE), the Land Steiermark and the Republic of Slovenia under the Tedusar grant. Safdar Zaman

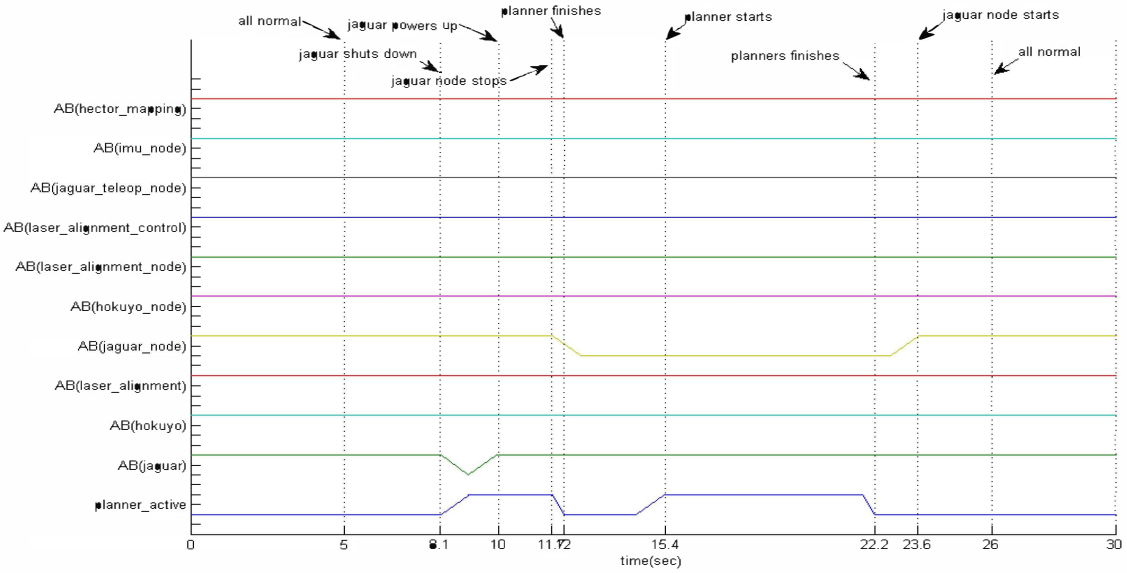

Fig. 6. Behavior of the diagnosis and repair system for the Device-Shut-down scenario.

is supported by the Higher Education Commission (HEC) of the government of Pakistan.

REFERENCES

[1] 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 *Proceedings of IEEE Aerospace Conference*, 1999.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] X. Koutsoukos, F. Zhao, H.Haussecker, J. Reich, and P. Cheung. Fault modeling for monitoring and diagnosis of sensor-rich hybrid systems. *Proceedings of IEEE Conference on Decision and Control (CDC 2001)*, pages Orlando, FL, Dec, 2001.

[10] 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.

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

[12] Bernhard Peischl and Franz Wotawa. Model-based Diagnosis Or Reasoning From First Principles. *IEEE Intelligent Systems*, 18(3):32–37, 2003.

[13] 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.

[14] Damien Pellier. Pddl4j, 2008. http://sourceforge.net/projects/pddl4j.

[15] Avrim L. Blum and Merrick L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90:281–300, 1997.

[16] 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.

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