RoSHA: A Multi-Robot Self-Healing Architecture*

Dominik Kirchner, Stefan Niemczyk, and Kurt Geihs

University Kassel, Distributed Systems Group,
Wilhelmshoeher Allee 73, 34121 Kassel, Germany
{kirchner,niemczyk,geihs}@vs.uni-kassel.de
http://www.vs.uni-kassel.de

Abstract. Reliability is one of the key challenges in multi-robot systems to increase practicable applicability and hence the commercial usage. This paper presents RoSHA, a self-healing architecture for multi-robot systems. RoSHA is based on the established robot middleware ROS and provides components for application independent analysis and repair. A plug-in architecture enables the developer to simply add new components for repair and analysis. Bayesian networks are used to diagnose failures and their root causes. ALICA, a domain specific language for multi-robot systems, is applied to coordinate recovery plans in multi-robot systems.

Keywords: self-healing, multi-robot system, system monitoring, failure diagnosis, system recovery

1 Introduction

Multi-robot systems become more and more important for many application domains, like search and rescue, warehouse management, or urban exploration. Often the complexity of tasks and domains demands autonomous operation in unknown and dynamic environments. These systems are confronted with complex tasks, like planning, localization, or coordination. The dynamic environment and the intricate interaction between these tasks result in a failure-prone setting [1, 2]. Accordingly, an architecture of multi-robot systems often consists of connected components. A failure in a single component can lead to a degeneration or even a crash of the total system. For example, in a single robot task, the localization of the robot relies on the fusion of multiple sensor information, like a direction, determined by a compass module and a distance scan, extracted from a camera image. If an error occurs in only one of the involved sensors, the robot could be delocalized. Due to the coordination and interactions of single robots in a multi-robot task, the resulting susceptibility to failures is expected to become even worse.

* The project IMPERA is funded by the German Space Agency (DLR, Grant number: 50RA1112) with federal funds of the Federal Ministry of Economics and Technology (BMWi) in accordance with the parliamentary resolution of the German Parliament.

In order to foster real-world applications and to increase the commercial use, reliability is a key design challenge. Carefully tested system design is one step in this direction. However, many errors result from the dynamic operation conditions, thus automatic failure detection and self-healing is required at runtime. To achieve self-healing capabilities, the system needs to identify root causes of a failure and execute a recovery policy. A more advanced concept than static failure handling is required. In this paper we present a generic and abstract robot self-healing architecture, called RoSHA. This architecture is based on ROS (Robot Operating System) diagnostics[1] and is inspired by the MAPE-K cycle [3]. It contains the five blocks monitoring, diagnostic, recovery planning, repair execution, and a knowledge base. We use an abstract system model and Bayesian networks as knowledge base to determine the root causes for detected failures, like a component crash, a deadlock, etc..

The rest of this paper is organized as follows. Section 2 identifies the requirements for a self-healing architecture. In Section 3 we introduce ROS and the existing diagnostic stack. The architecture and basic concepts of RoSHA are presented in Section 4. In Section 5 we name and discuss existing solutions. Finally, in Section 6 we conclude the paper and propose future work.

2 Requirements

Robustness of multi-robot systems is a key challenge that only little research has been directed to. In order to contribute to this challenge the proposed architecture offers self-healing capabilities. Often the architecture of a multi-robot system is characterized by a high degree of complexity. Therefore, the interaction between the robot system and the self-healing add-on should be carefully designed and limited to few well defined interfaces. The integration effort for robot developers should be minimized. Moreover, the self-healing add-on should not be application nor domain specific, rather provide its services in a general way. Most robot systems have limited resources and need to perform their tasks in soft real-time. Therefore, the architecture of the self-healing add-on should be resource-efficient to prevent indirect interferences. Scalability is another important requirement. The self-healing add-on should be independent from size and distribution of a multi-robot system.

Beside these envisioned features of a self-healing architecture, humans should be still able to oversee and control the system. If executed wrongly, repair actions can disturb the operational performance of the robots, e.g. in the case of an unnecessary restart of a core component. To cope with these risks, human operators should be able to intervene at any time. Furthermore, logged human intervention actions provide a valuable source of labeled training data for improvements. In summery we identified the following requirements of a self-healing architecture:

Ease of integration: To support existing robot systems without extensive integration efforts, well-defined interfaces between the robot system and the add-on have to be claimed.

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

Resource-efficient: The runtime support must not decrease the robot's performance. Therefore, the self-healing add-on has to be implemented in a light-weighted resource-efficient way.

High degree of configurability: The application and domain independent components of the self-healing add-on require a high degree of configurability. This is necessary to adjust these components to special needs of an existing robot system.

Human controllability: An unnecessary executed recovery action can decrease the performance of the whole robot system. Therefore, human intervention and control should be supported.

Extensibility and modularity: To extend the self-healing add-on to domain and application specific requirements a flexible design is required. Further application specific components, e.g. a new specialized control interface, should be simple to develop and integrate.

Multi-robot support: To improve the recovery, inter-robot interactions should be considered in the recovery execution. Therefore, multi-robot coordination for complex recovery processes should be supported.

3 Background

In order to realize the requirements stated in Section 2, we use ROS (Robot Operating System) [4], which is introduced in the remainder of this section.

3.1 Robot Operating System

ROS is a software framework for single and multi-robot system development. It provides standard operating system services such as hardware abstraction, low-level device control, implementation of commonly-used functionality, message-passing between components, and package management. The middleware is based on a graph architecture where processing takes place in nodes. These nodes receive and send messages using a publish/subscribe mechanism. This multi-component structure directly supports the required ease of integration and extensibility. The middleware services, like connecting different nodes, are dynamically organized from a central arbiter, the roscore. Communication in this system is not limited to local inter process communication (IPC) and could be distributed over a network as well. ROS contains repositories for additional packages. Users can contribute packages to the community that implement common functionalities for reuse, such as simultaneous localization and mapping, planning, perception, simulation, and so on.

3.2 ROS Diagnostics

In ROS the task of analyzing and intuitive reporting the system state is provided by the diagnostics stack. It consists of development support for collecting

and publishing information, an analysis and aggregation node, and visualization tools. This tool-chain is built around standardized interfaces, namely the *diagnostic* topic for monitoring information and the *diagnostic_agg* topic for the analyzed and aggregated results.

To make relevant information available, the diagnostic stack provides support for integration of a monitoring publisher in a ROS nodes. This allows the nodes to publish diagnostic data, the node status, or to monitor processing timings. Gathered data are published continuously on the *diagnostic* topic. Additional, there exists support for triggered node self-tests. The *diagnostic_aggregator* node is responsible for analyzing and aggregating reported data at runtime. The aggregation is used to categorize the monitoring data to an information item that summarizes one aspect of the system, e.g. status information of one component. The analysis is performed using a plug-in model. Each plug-in provides one analyzer that analyzes and aggregates information on a system's aspect. This aggregated information is sent on the *diagnostic_agg* topic for notification or visualization. A standard tool for visualization is the *robot_monitor* node that highlights the overview. Detailed information can be easily accessed for all aspects. Due to the defined interface the integration of application specific reporting is facilitated.

In summary the ROS diagnostic stack provides the required features for multi-robot support, extensibility and modularity, and human control. The ROS middleware offers local and network communication and hence supports multi-robot systems. The multi-component support in combination with the defined interfaces in the ROS diagnostic stack facilitates extensions in a convenient way.

Beside the discussed strength, there exist some shortcomings as well. The information gathering has to be integrated in the source code of the components. Moreover, the robot system must be developed with ROS to communicate the diagnostic data from the nodes. Therefore, the claimed ease of integration to an existing system is not fulfilled. However, the main shortcoming is the lack of autonomous failure recovery. For system recovery ROS diagnostics strictly relies on human operators to manually repair the system.

4 RoSHA Design and Architecture

In this section, we describe our approach to realize a self-healing add-on that fulfills the requirements identified in Section 2. Apart from the discussed shortcomings, ROS diagnostics already provide some of these requirements discussed in the previous section. Thus, we decided to build on ROS and the ROS diagnostics stack to develop RoSHA. In the development, we explicitly consider the special needs of multi-robot recovery, as described later.

As depicted in Figure 1, the structure of the overall system consists of two basic parts, which outline the local and the distributed aspect of the system. Here, the distribution of a multi-robot system is abstracted as interactions of a local robot with its team. Details of our proposed self-healing architecture are presented in the robot part. This part consists of four basic blocks (presented

as gray areas). The ROS middleware and the operation system serve as the fundamental hardware abstraction and execution layer that provides basic system services. The robot system realizes functionalities for the intended operation of the robot. The last block is RoSHA, which supports the reliability of the robot system and the user, who operates the system. The internal structure of RoSHA is built on the concept of MAPE-K [3] as the underlying decision cycle. This cycle is composed of five basic blocks, monitoring, analyzing, planning, execution, and a knowledge base. Following the requirement of modularity, we realized each block as an independent component.

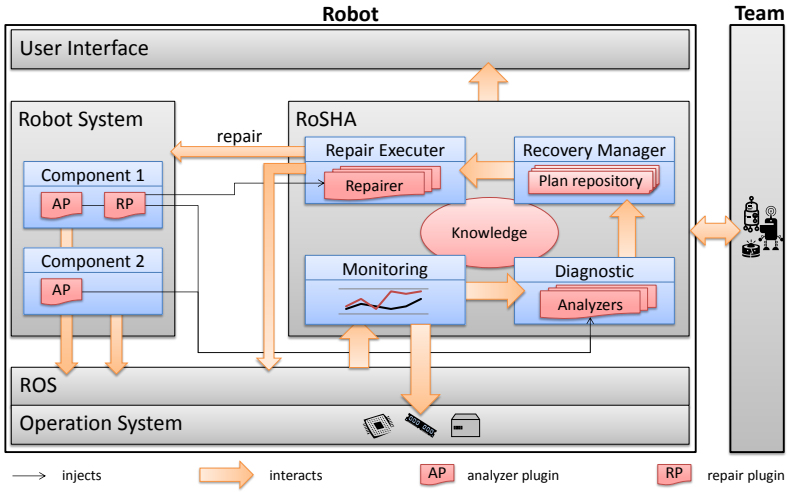

Fig. 1. Overview of the overall robot system architecture

The monitoring component collects information about the current system state. Therefore, two different aspects of the system are monitored. The first one is the knowledge provided by the operation system, like the current resource usage of a component (CPU load, memory usage, thread count, etc.). The second aspect is information provided by the components themself. If used, each component can directly send status information on the ROS monitoring topic, which is received by the monitoring component of RoSHA.

The diagnostic component uses the collected information to identify failures and their root causes. A set of basic plug-ins exist that supports generic and component independent analysis. In our opinion generic analyzing is not enough to identify all failures. The developer of the component is best capable of providing component specific analysis. Each component can provide its own analyzing plug-in (AP). This is injected into the diagnostic component at runtime. A system model is used to determine the root causes for each detected fault.

Detected faults are reported to the recovery manager. This component selects a recovery plan from a set of predefined policies to recover from the failure. The

repair execution component performs the repair actions included in the selected recovery policy. Similar to the diagnostic component the execution component provides a set of generic repair actions. This set can be extended with component specific actions, which can be offered as repair plug-ins (RP).

4.1 System Model

In accordance to the decision cycle, we use a single model to represent the accumulated knowledge. This model describes the current configuration of the multi-robot system. Additionally, configuration information of the self-healing add-on are included, e.g. the configuration details of a robot specific monitoring.

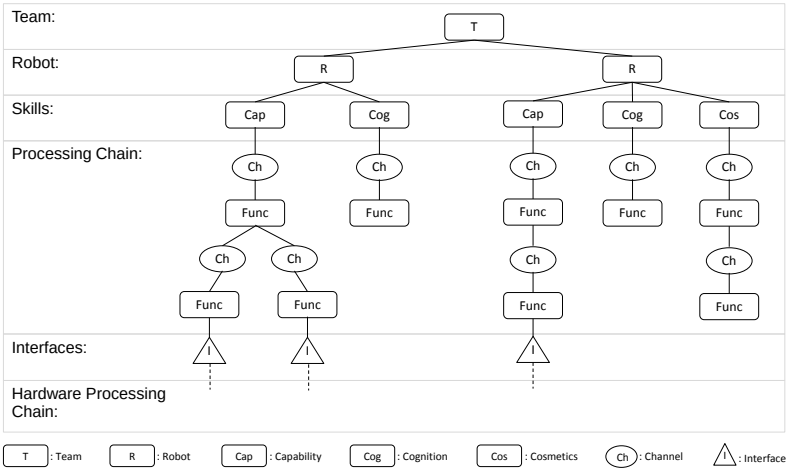

Fig. 2. Layered visualization of an exemplary system model in a multi-robot setting

We define a representation of the system according to the information flow between the components (see Figure 2). Therefore, we transformed a data flow diagram (graph representation) in a tree structure to express sequence and parallel component configuration. Furthermore, we include additional semantic elements on the upper levels of the model. The top-level element is the *team* node that represents the total multi-robot system. A team consists of *robot* nodes, which are placed on the second layer. All configuration-related to the overall robot system is specified here. On the next level, the model is decomposed in the robot's basic skills, like the ability to move, detect objects, localize, or plan. The set of skills is divided in *capabilites*, *cognition*, and *cosmetics*. *Capabilites* are used for sensor and actor skills, *cognition*, for cognitive skills like the planning and decision making of an autonomous systems, and *cosmetics*, for pure supportive skills, like a graphical user control interface. Each skill consists of *functionalities* and *channels* to represent the processing flow. Functionalities correlate with

the system components of the robot system and channels represent the communication links between them. Components that are present in multiple skills are presented separately in each skill and linked together. This processing chain ends with an *interface* element that marks the end of the software domain layer and the beginning of the hardware layer. Modeling the information flow on the hardware layer is not yet included, but part of the future work.

4.2 Monitoring

The monitoring is responsible to observe the current system state. The system, as described in Section 4.1, is composed of connected components. Therefore, capturing the entire robot system's state means to collect information of each component. However, properly working components do not directly induce a working system, the interactions between these components have to be considered as well. Therefore, we define two categories: the *component monitoring*, to capture components' state information and the *flow monitoring*, to supervise the communication flow between components.

The subject of the monitoring is one aspect of the information collection, the provider is the other. A common way to do this is to extend the components of the robot system to send current state information. An example for this is a heart beat signal. The component itself continuously sends a message to show its liveliness. We call that type of information collection *active monitoring*, because the components of the robot system have to provide their state information by their own. The self-healing architecture supports this monitoring, however we focus on a more flexible information collection. The proposed monitoring process focuses on the collection of general characteristic information of a component, like its resource usage. No direct support of the component is needed. This is referred as *passive monitoring* in the context of our work.

Passive monitoring can be very resource-inefficient. Therefore, we developed an adaptive monitoring, which is based on system specific aspects defined in the system model. For each component a set of characteristics are defined and monitored due to a set of properties. For example one characteristic could be the CPU usage of a component. The property describes the expected range from 15% to 25% with a typical value of 17%. The distance between the typical value and current value in respect to the borders is used as the distance metric to adapt the monitoring process. If the CPU usage is near the typical value, the monitoring level will decrease and the monitoring interval will increase and vice versa. The adaptive monitoring reduces the system load in error free situations. Furthermore, it reduces the amount of data to analyze as well. In a typical robot system components are not equally important. Therefore, the system model supports the configuration of initial monitoring levels for each functionality and channel.

4.3 Diagnostics

After monitoring the state of the system, the interpretation has to be done. This is addressed by the diagnostic component. The perceived state information can

be seen as symptoms of the overall system health. Therefore, the goal of the diagnostic component is to aggregate related symptoms, calculate an estimation of the health state, and identify possible root causes.

As described in Section 4.1, we defined a model of the component dependencies. This structure is composed of component items and communication items. Both are subjects of our monitoring and hence, state information are continuously updated. In our work we apply Bayesian networks for the diagnosis [5]. Bayesian inference needs a structured knowledge representation, a Bayesian network, to perform the diagnostic analysis. In our self-healing add-on this knowledge representation models the correlation of symptoms, root causes, and the resulting failure probability (see Figure 3). The resulting model can be structured in three layers, the symptom layer: the root cause layer, and the component failure layer. In the context of the Bayesian formalism, the monitored information is regarded as continuously updated evidences of the symptoms. Therefore, ongoing inference of the model is needed. In order to address temporal characteristics, we extend the models to dynamical Bayesian networks by including temporal dependencies to capture time series properties. With this extension we are able to model and analyze trends in symptoms to improve the failure analysis, e.g. an increasing memory usage in the case of a memory leak.

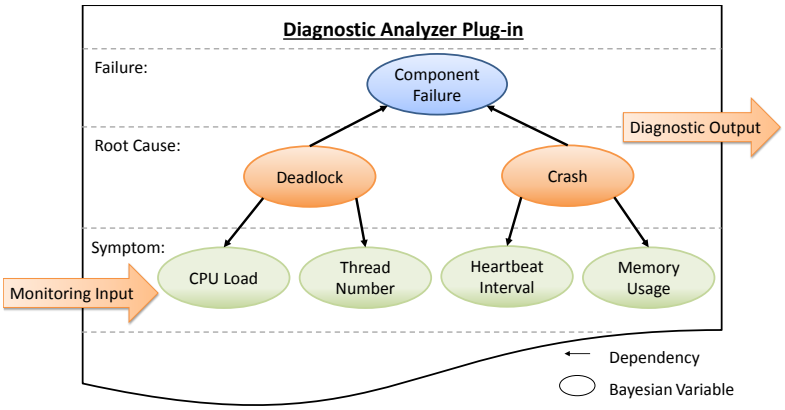

Fig. 3. Bayesian network for the diagnostic task

Besides a similar layered structure of the failure models, the included knowledge, like a-priori probabilities and dependencies, is specific for each component. This enables the distribution of the models to the location of the corresponding component to support an alignment of failure modeling. In order to process these distributed models, we apply the plug-in model to loosely connect the failure models with the diagnostic component of the self-healing add-on.

With these distributed models, inference of failure probabilities and root cause probabilities can be inferred separately. In order to infer a reliability es-

timate for the entire robot system, or some subsystem of it, the interference of components' failure probabilities has to be considered. This combination can be calculated using reliability theory [6]. Therefore, we apply the formalism of reliability block diagrams (RBD) to compute the (sub-) system reliability.

4.4 Recovery Manager

After diagnosis, detailed information of failure probabilities of the robot system are available. In addition, the diagnostic component provides probabilities of root causes to identify the most likely reason for a potential failure. Based on these information, the responsibility of this component is to select a fitting recovery policy to restore the system. As mentioned before, monitoring information, and hence the diagnostic results, change in real-time. Dynamic planning for each situation is a time consuming task that requires much resources [7]. This is in conflict with the requirement of recourse efficiency. Therefore, we decided to provide a set of predefined, but proven recovery policies. These policies are stored in the plan repository of the component.

For plan selection and team recovery coordination we use the multi-agent coordination language ALICA [8]. ALICA inherently fulfills the requirement of multi-agent support. The policies, called plans in ALICA, set up a strategy of repair and assessment actions for a team of agents. In such a plan we are able to model complex repair and check procedures with multiple robots involved.

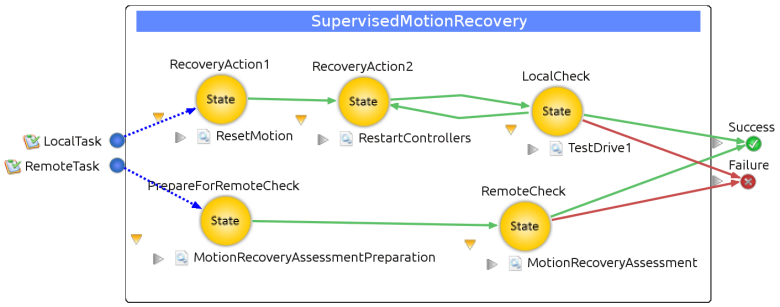

Fig. 4. Recovery plan in ALICA

Figure 4 illustrates an example recovery policy that is designed to recover a robot's motion process. It performs two recovery actions and locally verifies the success by a test drive. Additionally, an independent assessment of the recovery is done by supportive robots that confirm the success through their independent observation of a matching movement. Therefore, explicit knowledge exchange between the robot system and the recovery manager is used. The recovery policies are case specific and modeled manually. The needed environment knowledge for the recovery assessment has to be communicated to the self-healing add-on by

the robot system. The communication can rely on IPC or the plug-in model. Without the robot system's support, the recovery manager is limited to local repair coordination based on monitoring and diagnostic information.

4.5 Repair Execution

The recovery manager selects an appropriate recovery policy and coordinates the involved repair actions. These actions are communicated to the repair executor node. Two types of repair actions are distinguished. Generic repair actions that provide repair services independent of the target component and specific repair actions that are designed for one specific target node. Both types are realized as plug-ins, similar to the analyzers of the diagnostic component. The recovery impact of generic repair, like a restart of a component, is limited due to their general scope. Specialized component knowledge is needed to implement more specific repair actions, e.g. a runtime reconfiguration of the component. Only the component's developers possess this detailed knowledge and seems best fitted to implement these specific repairs. To support the extensibility of the self-healing add-on, the specific repair plug-ins are located in the components. A third, more sophisticated way to restore a system is an architectural recovery. The goal of this recovery policy is to change the architecture of the robot system in order to restore the operational mode. Thereby, the architectural changes comprise adding, removal, and connection of system components. In this policy a sequence of architectural repair actions are performed. These repairs are not limited to local actions, but involve the total system. In a multi-robot system we need to perform distributed architectural actions. After an unsuccessful local recovery, distributed repairs allow to compensate this failure through relocation of the failed component to another robot system. If local and distributed recovery failed, the robot looses some of its capabilities and is seen as degenerated. However, such a robot could still contribute to the global task. In order to continue this task as good as possible, the degree of degeneration has to be considered and appropriate adaptation actions should be performed, e.g. a global task reallocation. These repair actions are ongoing work in RoSHA .

5 Related Work

The general problem of self-healing is addressed in several projects. In the following we like to present some of the related work and discuss their strength and shortcomings.

The SHAGE (Self-Healing, Adaptive, and Growing Software) framework [9] consists of two parts that enable the self-management of a robot. The first part comprises several components to manage the system, while the second part contains internal and external repositories to store architectural reconfiguration descriptions and components. These two parts work together to observe the situation of the environment and to trigger appropriate architectural reconfigurations. Furthermore, SHAGE includes a learning component to improve the

reconfiguration due to previous experiences. This reconfiguration aims to adapt the behavior of the robot in exceptional situations. However, the challenge how to identify these situations is kept open. Recovery of these situations is done by changing the behavior of the robot through architectural adaptation while component failure recovery is not addressed. The framework includes interfaces to external repositories to share knowledge, but does not support coordination of multiple robots.

The Rainbow project [10, 11] proposes an architecture-based self-adaptation approach. It provides reusable infrastructure together with mechanisms to tailor these to the domain's needs. These specializations allow the developer to define aspects of the system, like monitoring targets or adaptation conditions and actions. The adaptation is done through statically associated sets of action rules for each identified adaption cause. The information collection and the execution of adaptation actions rely on direct support of the target system. Therefore, integration in an already existing system seems difficult. Furthermore, the architecture is inherently centralized and thus sensitive to single-point of failures and of limited use for distributed systems, like multi-robot systems.

LeaF (learning-based fault diagnosis) [12] from Parker and Kannen is based on an adaptive causal model for fault diagnosis and recovery. The approach focuses on multi-robot systems. The causal model represents expected faults and is initially created by the developers at design time. A case based reasoning is used to handle unexpected faults during runtime. The system extracts possible recovery options from the existing model and adds the new fault and a recovery method afterwards. Furthermore, the model can represent faults that only occur during the interaction between robots. However, this approach does not cover hardware related monitoring and fault detection. The authors make no statement how to integrate LeaF in an already existing robot system or how to change or replace existing components.

Often, a robot system is built without considerations of system management, including self-healing. The main concern is to create a working system for a given task. Developers tend to neglect reliability in this design phase. That could lead to the situation in which it is necessary to integrate self-healing abilities to an already existing system. The discussed projects propose remarkable results in the field of self-healing through architectural adaptation. However, they do not address questions of practicable usability, like the integration in an existing system, or resource efficiency. We argue that these properties are central design decisions and should be reflected as architectural requirements. We argue as well that coordination and assessment of multi-robot recovery actions are central requirements for a domain and task independent self-healing architecture. In comparison with the discussed projects, RoSHA considers these requirements.

6 Conclusion and Future Work

Due to the system complexity of modern robot systems, reliability cannot be ensured in design time. Hence, runtime failure recovery in a self-healing add-

on is needed. The integration of the self-healing add-on in an already existing multi-robot systems is essential in the sense of practicable usage. Therefore, increased usability and multi-robot support is required. In this paper, we presented a robot self-healing architecture to address these challenges. We propose a framework that is tailored for generic use in multi-robot scenarios. The design of the framework addresses the need for resource efficiency through the usage of an adaptive monitoring. Plug-in support with existing generic repair and analysis plug-ins enables an ease of integration. The coordination of multi-robot recovery policies is given by the language ALICA .

As part of our ongoing research we plan to integrate distributed failure learning methods to reduce the dependency of expert knowledge. A comprehensive real-world evaluation of the self-healing add-on will be done at the RoboCup[2] world championships 2013. In this dynamic setting the suitability for multi-robot teams will be evaluated. Furthermore, we plan to compare the recovery performance of a fully supported self-healing add-on with results from limited system support, e.g. with restricted support to generic monitoring and recovery actions.

References

1. Carlson, J., Member, S., Murphy, R.: How UGVs Physically Fail in the Field. IEEE Transactions on Robotics **21**(3) (2005) 423–437
2. Carlson, J., Murphy, R.: Reliability analysis of mobile robots. In: International Conference on Robotics and Automation. Volume 1., Ieee (2003) 274–281
3. Huebscher, M., McCann, J.: A survey of autonomic computing degrees, models, and applications. ACM Computing Surveys **40**(3) (August 2008) 1–28
4. Quigley, M., Conley, K., Brian P., G., Josh, F., Tully, F., Jeremy, L., Rob, W., Andrew Y., N.: ROS: an open-source Robot Operating System. In: ICRA Workshop on Open Source Software. Number Figure 1 (2009)
5. Pearl, J.: Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference. Morgan Kaufman Publ Inc (1997)
6. Shooman, M.: Reliability of Computer Systems and Networks: Fault Tolerance, Analysis, and Design. John Wiley & Sons, Inc. (2002)
7. Ghallab, M., Isi, C.K., Penberthy, S., Smith, D.E., Sun, Y., Weld, D.: PDDL - The Planning Domain Definition Language. Technical report, CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control (1998)
8. Skubch, H.: Modelling and Controlling Behaviour of Cooperative Autonomous Mobile Robots. Phd thesis, University of Kassel (2012)
9. Kim, D., Park, S., Jin, Y., Chang, H.: SHAGE: a framework for self-managed robot software. In: Proceedings of the International workshop on Self-adaptation and self-managing systems, Shanghai, China, ACM Press (2006) 79–85
10. Garlan, D., Cheng, S.W., Schmerl, B., Steenkiste, P.: Rainbow : Architecture-Based Self-Adaptation with Reusable. Computer **37**(10) (2004) 46–54
11. Cheng, S.W., Garlan, D., Schmerl, B.: Evaluating the effectiveness of the Rainbow self-adaptive system. 2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems (May 2009) 132–141
12. Parker, L., Kannan, B.: Adaptive Causal Models for Fault Diagnosis and Recovery in Multi-Robot Teams. In: Intelligent Robots and Systems. (2006) 2703–2710

[2] `http://www.robocup2013.org`