OROCOS based control software of the new developed MARUM Hybrid-ROV for under-ice applications

Gerrit Meinecke*, Jan Albiez†, Sylvain Joyeux†, Volker Ratmeyer* and Jens Renken*
*MARUM – Center for Marine Environmental Sciences
University of Bremen
Leobener Str., D-28359 Bremen
Email: gmeinecke@marum.de
†DFKI – German Research Center for Artificial Intelligence
RIC – Robotic Innovation Center
Robert-Hooke-Str. 5, 28359 Bremen

***Abstract*—The water column as well as the seafloor in under ice environments (sea ice or glaciers) are increasingly interesting for researchers. Exploring these regions needs new technical equipment, which is able to cope with the challenges arising when entering areas covered by ice. There is no escape route to the surface, caverns within the ice and an unknown sea-floor topography makes impact severely on navigation and communication with the deployed systems is complicated. Additionally, the requirements from the researchers to the systems are high (probe sampling, video coverage, sonar images etc.). The HROV system developed at MARUM intends to deal with these problems by applying a hybrid ROV-AUV approach to a underwater vehicle. Within such a hybrid system, the control architecture has to satisfy the needs of a human operator controlling it in ROV mode and the constraints arising from autonomous operation. Resulting from a close cooperation between the DFKI Robotics Innovation Center and MARUM we present within this paper a control architecture which uses OROCOS/ROCK as framework to implement a system which allows seamless and immediate switching between ROV and AUV mode (e.g. in case of fibre rupture) as well as applying an adaptive fault detection and fault response system.**

I. Introduction

AUVs and ROVs both have their specific operational requirements and application. AUVs are commonly used in untethered and autonomous mode to travel fast and long distances, whereas ROVs are used tethered, being slow but precise in remote controlled mode for short distance travel and work on relatively small target sites at the seafloor. Consequently, the technical constrains of theses vehicle classes are inherent, somehow limiting scientific applications, i.e. often there are no hovering capabilities on common AUVs to allow work on small target sites, whereas the rigid umbilicals used on ROVs do not allow large area coverage and higher travel velocities. However, scientific needs are evolving which demand a technology that ideally takes advantage of both vehicle classes.

The shrinking ice-sheets in formerly ice-covered areas for example, increasingly allow scientific missions in such ice-affected ocean areas at both hemispheres. Neither commercial AUVs nor commercial ROVs exist today which are originally designed to work underneath sea or glacier ice. Nearly all fault response strategies of the vehicle control systems rely on environmental conditions with open water above the vehicles as ultimate rescue pathway. Thus, vehicle operations in a dynamic sea-ice affected area, near the ice-edge or underneath sea-ice are essentially dependent on a dedicated but also flexible vehicle OS for such conditions.

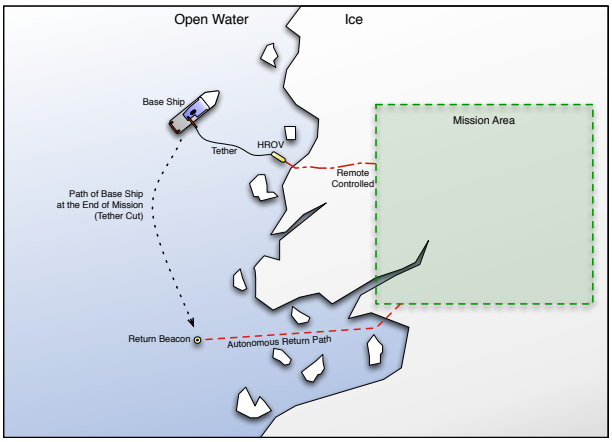

Fig. 1. The HROV operational scenario

To address these new challenges, it is intended to operate the MARUM Hybrid-ROV [1] in a remote controlled mode via a thin glass-fibre as standard, well facing the risk that the fibre could break at any time during mission and force the vehicle OS to swap into an AUV-mode instantly (see fig. 1). Therefore, the new developed OS needs to be powerful, easy to manage and able to be modified with respect to scientific tasks or operational risks resulting from extreme environmental conditions such as ice coverage. Within the development of the MARUM Hybrid-ROV, the Robot Construction Kit (Rock) open source project was chosen as software framework to implement the stated requirements.

978-0-933957-40-4 ©2013 MTS

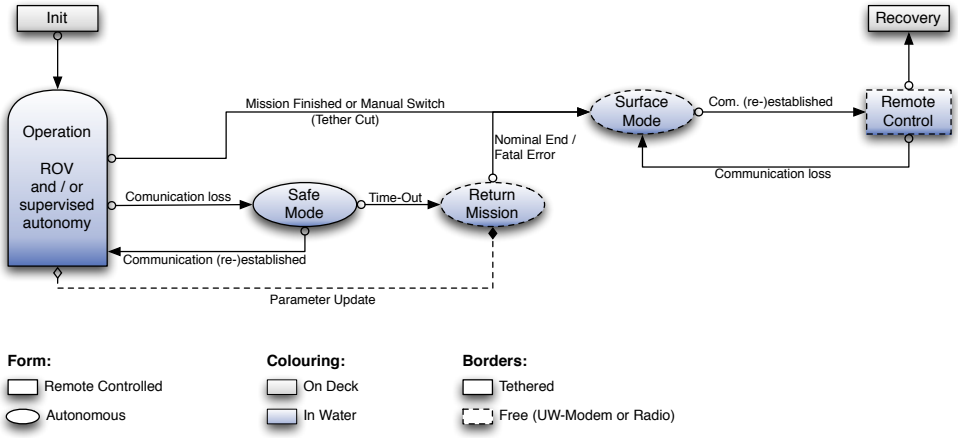

Fig. 2. Overview of the general operation modes and their semantic connection for the HROV architecture

II. Architecture Overview

The main focus while designing the HROV architecture was on a seamless *multi-mode* concept. The primary mode of operation of the HROV will be remote operation. Within this mode the vehicle will be tested on-deck, deployed and conducting the mission. The switch to the autonomous system will either happen on the end of the mission or due to a persistent communication failure (e.g. damage to the fibre). In fig. 2 the general control procedure is shown graphically.

These principal ideas and procedures were the basis for designing the architecture itself. The speciality for HROV is, that it has to be able to seamlessly switch modes (ROV to AUV). ROV control architectures focus completely on a human operator/pilot, who has to have the complete control of all aspects of the robot at any time (e.g. restarting components). AUV architectures focus on ways to allow the control software to make simple decisions with respect to a tightly pre-defined mission. For HROV these two worlds have to come together.

Typical AUV control systems these days use a simple component based system (e.g. [2], [3], [4], [5]) most often based on the current state of the art distinguishing between a front-seat driver (basic motion control) and a back-seat driver (mission management, customer supplied control) as presented in [6].

The envisaged HROV architecture is based on the successfully used AVALON control architecture (see [7]) which is implemented using Rock (see section III). The primary case of an operator controlling the vehicle is considered as a normal behaviour within this architecture, which is invoked by the mission plan as soon as the mission is started. The operator accesses all vehicle functionality by using the same functions and components as the mission control system would do. This scheme has the following advantages:

- The control software of the vehicle intrinsically tracks the vehicle state all the time. When the system suddenly has to switch to autonomous mode, there are no undefined or unknown parameters.
- The operator can use all the functions the vehicle provides for autonomous operations, such as hovering, course keeping etc.
- An AUV like fault detection and response system is running all the time, which supports the operator and allows a quick response or switch to autonomous mode in the case of failures.

The main drawback of this scheme is, that it restricts a human operator to the software functionality provided by the control system. Since ROV pilots are used to be able to access all functions of their vehicle directly without any software system in between, it will take some time for persons trained as ROV pilots to get comfortable with the HROV. Since the HROV under-ice operation are considered critical per-se which need special training, we consider this as a minor problem.

In the following the key features of the architecture are explained in more detail.

III. The Robot Construction Kit

The Robot Construction Kit is an open-source[1] component-based software framework built on top of the Orocos/RTT component implementation. It significantly differs from Orocos by its extensive toolchain, which tremendously eases the development of components and the integration of systems.

The main differentiator of the Rock approach from the widely-used ROS framework is that it follows a model-driven approach, and follows the principle of separation of concerns [8]. Moreover, it offers a very advanced approach for system integration and runtime adaptation, which is the cornerstone of the HROV runtime management and safety.

This section will present the main characteristics of Rock's approach to component-based systems, and of Syskit, the Rock system integration and management layer.

A. Component Model

The underlying component implementation in Rock is the Orocos/RTT. One of RTT's most interesting characteristics is that it is designed following the separation of four concerns: *computation*, *communication*, *configuration* and *coordination*. In effect, a RTT component should only be seen as an

[1] http://rock-robotics.org

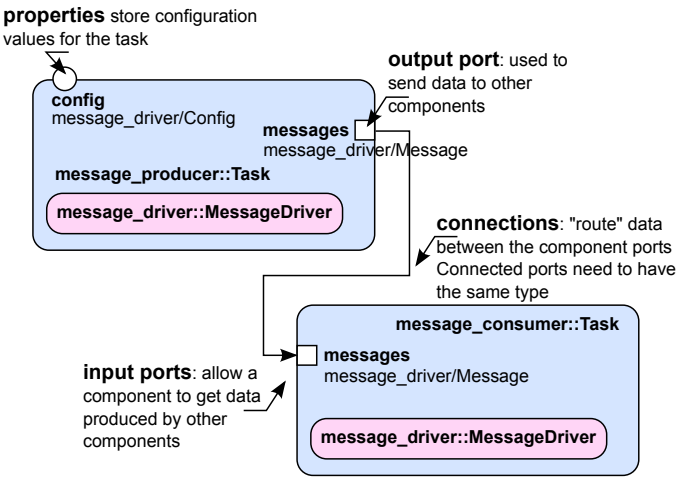

Fig. 3. Main elements of a Rock component interface

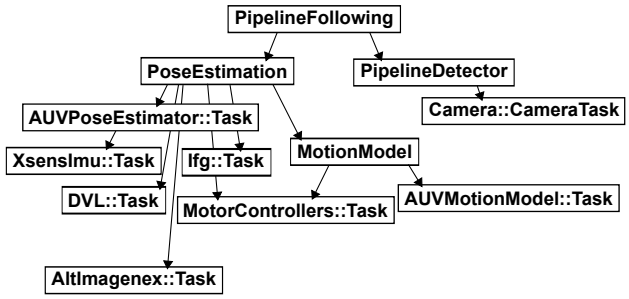

Fig. 4. Dependency structure for the pipeline following behaviour depicted on the right of Fig. 5. An edge $A \rightarrow B$ in this graph represents that the A requires B to run to function properly. Additional constraints are encoded in this dependency graph, that allows to take into account more fine-grained relationships

algorithm that transforms data from its inputs and sends it to its outputs. This algorithm can be configured (through a set of properties) and is independent of the means used to communicate with other components (currently, CORBA, POSIX Message Queues and ROS can be used transparently). The way components are "wired" (i.e. how the data flows in the system) is also completely defined separately and can be changed dynamically. Finally, to support coordination of components, each component provides a standard lifecycle state machine that allows to decide when it should (re)configure itself, start or stop processing 3.

The two main characteristics that are important for the rest of this paper is that all Rock components have by design an *identical* interface to *(i)* change how data flows in the system, *(ii)* decide which components should run at each point in time and *(iii)* a way to send notifications about events in a form that other autonomous software can use, such as e.g. `IO_TIMEOUT` for device drivers.

B. Syskit

The root idea of the HROV control software is that it should support being *transitioned* between various modes of operation. When teleoperated, an operator should have the choice of selecting various autonomous tasks that should be performed by the software – such as auto-heading or target tracking – while still being able to turn different devices on/off or – at any instant – to switch to fully autonomous operations because of e.g. umbilical failure.

Such flexibility obviously would come at a cost. Ideally, the control software should allow for new teleoperation-aiding features to be integrated without impacting the rest of the system. Moreover, we should be able to guarantee the transition to autonomous operations from *any combination of teleoperation-aiding features*.

For this purpose, the Rock framework provides a system integration and management layer called Syskit. Syskit has already been explained in detail in [9], so we will only present the aspects of the tool that are relevant for the HROV control architecture, without getting into details about the implementation of these features.

a) System Design: The root idea in Syskit is that one high-level behaviour, such as auto-heading or station-keeping can be defined as a *connected network of configured components*. Indeed, in a component-based system such as Rock, the system's behaviour is fully defined by *(i)* which components are running, *(ii)* how they are connected and *(iii)* how they are parametrized.

The main issue being there to *compose* these networks together while ensuring that the overall system behaviour is also going to be the (desired) combination of behaviours. This service is provided by Syskit: one only has to define the single behaviours and can then compose them to either form more complex behaviours (for instance, reuse a control loop network inside a station-keeping behaviour) or – at the operator level – by enabling or disabling behaviours.

b) System Runtime Adaptation: Additionally, Syskit is designed to – at runtime – provide the means to transition between different set of active behaviours. This is achieved by comparing the generated connected, configured component network that is currently running with the desired network (generated from the set of required behaviours) and adapt the running system accordingly (Fig. 5).

c) System Monitoring: Finally, the data structure built and maintained by Syskit allows for automated system monitoring and fault response. The general idea is that Syskit's modelling encodes the *dependencies* between major function and sub-functions of the system. At runtime, Syskit then uses that information as well as the notifications emitted by the Rock components to detect errors (as violations of dependency constraints). Other types of constraints can be encoded as well, such as e.g. temporal constraints (this component should start after that component). When an error is detected, it can either be handled by fault handlers defined in the fault response tables (detailed later in this paper) or default policies can be defined such as: restart the faulty behaviour, kill the faulty behaviour (the latter is the default).

IV. Teleoperation and Autonomy

The control software presented in this paper allows for truly mixed-initiative setups. It can be configured anywhere from fully teleoperated to fully autonomous, with any step in between being possible. Operators can decide which parts of

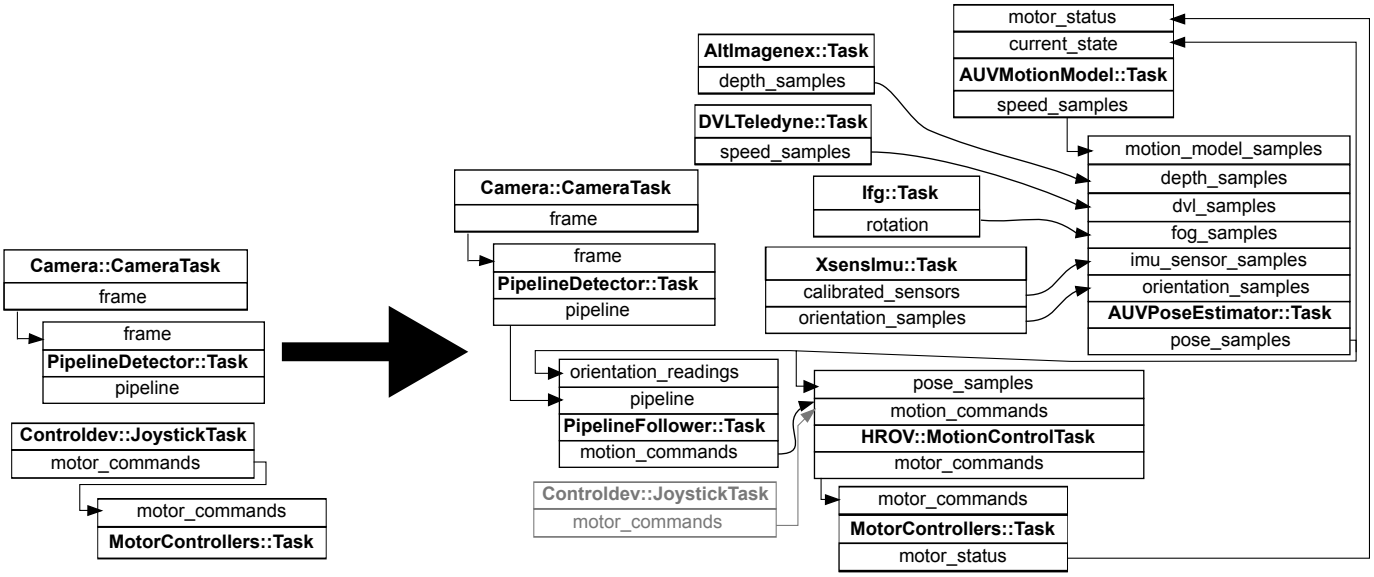

Fig. 5. Transition between a combination of pipeline detection and direct control behaviours (left) to a fully autonomous pipeline tracking behaviour (right). With only these three behaviour involved, writing these transitions manually in the code would require two cases, taking into account whether the operator has enabled the pipeline detection feature as an aid or not. On the right, the greyed out parts are the parts that have been removed, and the blue parts the newly started components. Note that, in a real system, the pose estimation would also probably be already running.

the system should be handled autonomously, which should be under their direct control, and which (autonomous) safeties should or should not be allowed to interfere with the direct human control.

This section will present how this is achieved. We will first present how we use the capabilities the Rock framework software presented in Sec. III to build an operator interface. Then, we will present how fault detection and response is integrated into such a mixed-initiative setup.

A. General Concept

The main idea behind the operator interaction concept is to use the capabilities of the Syskit software layer to ease the use of a system where more autonomous functions can and will be made available to the operators. There are two main issues to consider: *(i)* which functions can be activated, which cannot, and *why* and *(ii)* how to make the user understand the system faults and their implications on the chosen autonomous functions.

d) Aided teleoperation: The concept of operator interaction is to empower the operator by allowing him to freely use the autonomous functions (that must be integrated because of the AUV capabilities) while keeping control over the vehicle. However, this significantly complexifies the handling of the vehicle. In order to keep the interface usable, the capabilities of the Syskit layer are used to ensure the safety of the vehicle while allowing the operator to understand what he can and cannot do in the current situation.

Whenever the operator requests the activation of a system service, it is verified whether this service is compatible with the currently running software. If it is, the service is activated. Otherwise, Syskit's models and situation representation capabilities allow to propose a resolution of the form "if you want to start X, then you have to stop Y and Z". The operator can then choose what he wants to do and have the UI reflect this choice (Fig. 6).

e) Making faults human-readable: While the Syskit approach monitors, at runtime, the nominal execution of the selected behaviour, its suffers from the (common) issue that faults are encoded in a data structure that his easy to manipulate by software, but hard to understand for humans. It is therefore necessary, whenever an event happens that causes one or more functions to stop, to translate this structure into a representation that can be understood.

Fortunately, the relationships between the high-level functions and low-level processes is all kept and maintained by the runtime part of the Syskit layer. In its simplest form, one would relate the low-level fault with the high-level action. For instance, a critical loss of DVL lock during autonomous trajectory following) would be translated into: "autonomous trajectory following failed because of DVL loss_lock". The loss_lock symbol being a notification provided by the DVL Rock component. The framework also provides the ability to add more (textual) information in the task hierarchy. For instance, if a motion model is provided as a fallback to the DVL, one could encode in the software object that represents the complete pose estimation (Fig. 4) that both the DVL and motion model state should be given to fully explain the fault, as e.g. "autonomous trajectory following failed because of DVL loss_lock and MotionModel does not provide the necessary precision". This refinement must be manually written in the code, but can be done generic for all pose estimation loops that contain a DVL and motion model as fallback (i.e. be quite generic).

f) Teleoperation and AUV operations: It is important to note that, in this scheme, teleoperation and AUV modes are only two different possible configurations (as "set of

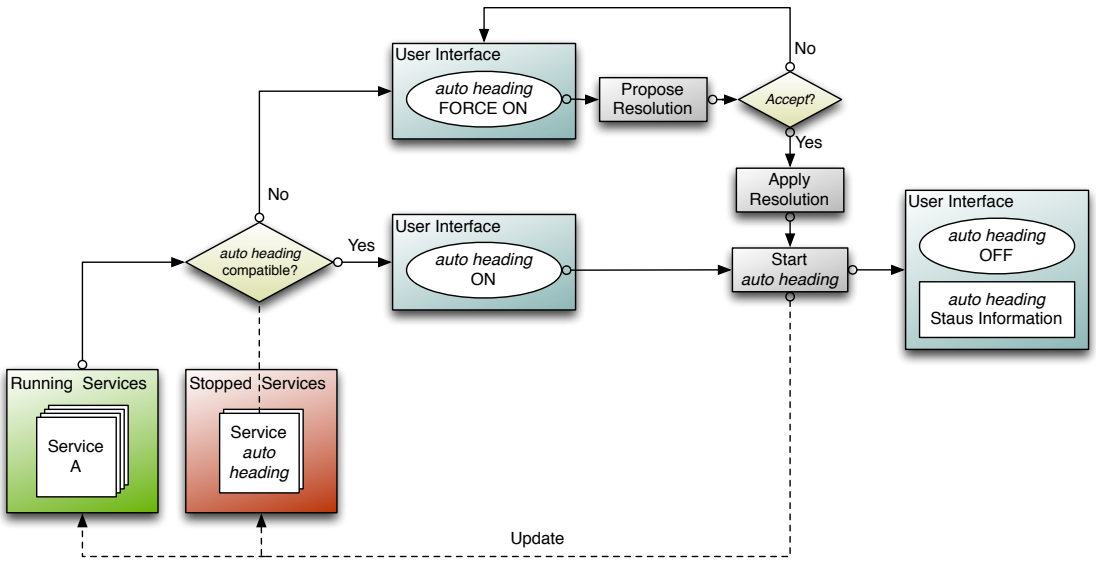

Fig. 6. Operator interaction scheme

enabled functions") of the whole system. The transition from teleoperated to AUV and back is guaranteed by Syskit, and the addition of new functions / modes is as simple as *defining them*. The integration in the operator workflow and in the system's safeguarding functions is guaranteed by the tool.

B. Fault Response

The fault response system is considered a critical component of the HROV control architecture, since this is the part where the most critical decision are made (e.g. switching to AUV mode due to communication loss).

In principal a classic table based approach will be used. Such a table consists of an predicate/rule based on sensor information which selects a response when its conditions are met. Fault tables are widely used within AUV control systems and have proven to be a successful method to control fault conditions.

Within the case of HROV there is one critical problem: The table is not the same for the complete mission of the vehicle, since an operator will always activate and de-activate behaviours during a mission (see above) which need special rules for fault cases and these rule can potentially conflict with each other which prohibits the loading of all rules in advance. To surmount this problem we introduced a stacked table (see fig. 7).

The rules are sorted according to their priority. There are rules which will always be active, either in hardware or software and which are static as soon as the mission starts and always have the highest priority (the two stacks at the bottom in fig. 7). On top of these two block reside the dynamic rules.

The dynamic rules consist of rules set, which are specific to a running component. These rules are injected into the system as soon as the component is started and delete when the component is stopped. Solving the dependencies and conflicts between different rules is delegated to the component managing system (syskit). The rationale behind this is, that

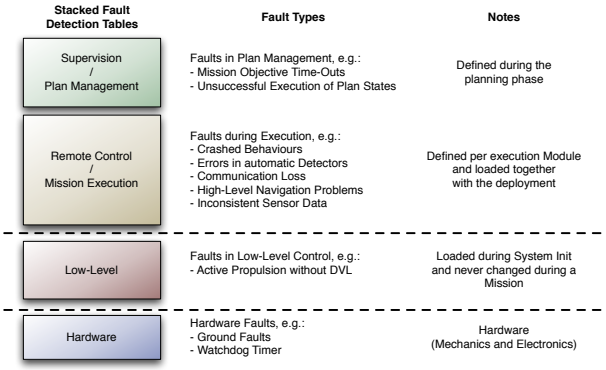

Fig. 7. Fault detection overview

when to components have conflicting fault response rules it is ver likely, that they will conflict in their functionality as well, which prohibits the start of the component anyway (see section above).

To prohibit clutter, redundancies and potential conflicts the fault response tables are not only split vertically, but also horizontally (see fig. 8):

- **Diagnostic rules:** are dealing with the detection of an error. They get all sensor input and component states and consist of predicates which create a *fault symbol*. The predicates not only take the current state into account, but have a history so that they can create symbols if e.g. a the integral of a sensor value reaches a certain level or a signal is changing to often within a certain time frame.
- **Response rules:** These rules react to the existence of one or a combination of several *fault symbol*. They generate an action which is given to the control system, and distributed to all components it applies to. Like the diagnostic rules, the response rules have a history as well.

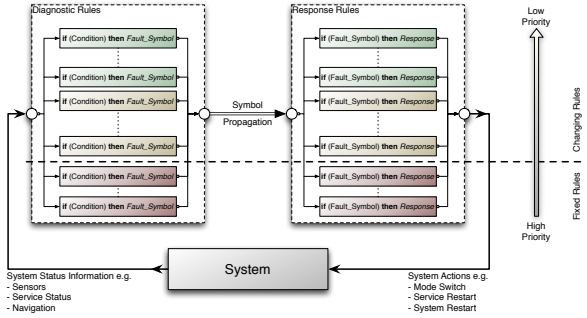

Fig. 8. Fault detection tables

The basic idea behind this split is, that the case of two components reacting to the same fault condition but operating on different, non conflicting reactions is very likely. The most prominent case here is the reaction to communication loss between the HROV and the control station. Splitting the tables reduces the number of active diagnostic rules (as it would be the case if each component creates its own rule and injects it into the system) and keeps the fault system manageable.

The fault detection and response system is implemented directly within Rock/Syskit (see above). There is a definition of rules for each component and these rules are injected into the system as soon as syskit starts the respective component.

V. Conclusion and Outlook

Within this paper we presented the concepts used in the control architecture for the HROV system. As basis for the architecture the Rock component model will be used in conjunction with Syskit to allow system integration and automatic management of the components. The concept of assisted teleoperation was introduced, which allows the control architecture to keep track of the current system state and therefore guarantees the possibility to switch seamlessly from teleoperation to autonomous mode while enabling an operator to use the different semi-autonomous or autonomous behaviours as support during operations.

A stacked fault detection and response system was described, which integrates into Syskit and is able to deal with the constraints arising from the necessity to switch between assisted teleoperation and AUV mode.

The architecture is currently implemented on the HROV system. Future work will deal with first tests on the finished system and the extension of the architecture to operational needs.

Acknowledgment

The authors would like to thank Prof. Dr. Dr. h.c. Gerold Wefer (MARUM) for strong support of the Hybrid-ROV project. The Hybrid-ROV development is funded by German Ministry for Education and Research, BMBF (Grant No. 03F0625A) and the University of Bremen.

Rock has been initially developed within the projects iMoby (grant no. 50RA0907), CSurvey (grant no. 03SX262B) and RIMRES (grant no. 50RA0904) funded by the German Ministry of Economics and Technology and conducted at the DFKI RIC.

The design of the Rock-based control system is a project of MarTech-Bremen Virtual Institute (Prof. Dr. Dr. h.c. Wefer; Prof. Dr. Kirchner; Prof. Dr. Dittus - MARUM Center for Marine Environmental Sciences; DFKI RIC Bremen (German Research Center for Artificial Intelligence; Robotic Innovation Center); DLR-Bremen (German Aerospace Center) funded by the state of Bremen.

References

[1] G. Meinecke, V. Ratmeyer, and J. Renken, "HYBRID-ROV Development of a new underwater vehicle for high-risk areas," in *Proceedings of the OCEANS MTS/IEEE Conference, (OCEANS-11)*, Kona, Hawaii, May 2011, pp. 1–6.

[2] M. H. Khalid, S. Z. Shakeel, S. A. Mansoor, and S. Q. Hassan, "FATCAR-AUV: Fault Tolerant Control Architecture of AUV," in *2007 International Bhurban Conference on Applied Sciences & Technology*. Islamabad: IEEE, Jan. 2007, pp. 161–167.

[3] J. Evans, Y. Petillot, P. Redmond, M. Wilson, and D. Lane, "AUTOTRACKER: AUV embedded control architecture for autonomous pipeline and cable tracking," in *Oceans 2003. Celebrating the Past ... Teaming Toward the Future (IEEE Cat. No.03CH37492)*. Ieee, 2003, pp. 2651–2658.

[4] A. Jalaoui, D. Andreu, and B. Jouvencel, "Contextual Management of Tasks and Instrumentation within an AUV control software architecture," in *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*. Beijing: IEEE, Oct. 2006, pp. 3761–3766.

[5] C. McGann, F. Py, K. Rajan, H. Thomas, R. Henthorn, and R. McEwen, "A deliberative architecture for AUV control," in *2008 IEEE International Conference on Robotics and Automation*. Pasadena: Ieee, May 2008, pp. 1049–1054.

[6] D. Eickstedt and S. Sideleau, "The Backseat Control Architecture for Autonomous Robotic Vehicles : A Case Study with the Iver2 AUV," in *OCEANS 2009, MTS/IEEE Biloxi - Marine Technology for Our Future: Global and Local Challenges*, Biloxi, 2009, pp. 1–8.

[7] J. Albiez, S. Joyeux, and M. Hildebrandt, "Adaptive AUV mission management in under-informed situations," in *Proceedings of the OCEANS MTS/IEEE Conference, (OCEANS-10)*, Seattle, Sep. 2010, pp. 1–6.

[8] E. Prassler, H. Bruyninckx, K. Nilsson, and A. Shakhimardanov, "The Use of Reuse for Designing and Manufacturing Robots."

[9] S. Joyeux and J. Albiez, "Robot development : from components to systems," in *Control Architecture of Robots*, 2011, pp. 1–15. [Online]. Available: http://hal.inria.fr/inria-00599679