SUAVE: An Exemplar for Self-Adaptive Underwater Vehicles

Gustavo Rezende Silva*, Juliane Päßler†, Jeroen Zwanepol*, Elvin Alberts‡*, S. Lizeth Tapia Tarifa†, Ilias Gerostathopoulos‡, Einar Broch Johnsen†, Carlos Hernández Corbato*
*Technical University of Delft, Delft, The Netherlands
Email: {g.rezendesilva,c.h.corbato}@tudelft.nl
j.m.zwanepol@student.tudelft.nl
†University of Oslo, Oslo, Norway
Email: {julipas,sltarifa,einarj}@ifi.uio.no
‡Vrije Universiteit Amsterdam, Amsterdam, The Netherlands
Email: {e.g.alberts,i.g.gerostathopoulos}@vu.nl

***Abstract*—Once deployed in the real world, autonomous underwater vehicles (AUVs) are out of reach for human supervision yet need to take decisions to adapt to unstable and unpredictable environments. To facilitate research on self-adaptive AUVs, this paper presents SUAVE, an exemplar for two-layered system-level adaptation of AUVs, which clearly separates the application and self-adaptation concerns. The exemplar focuses on a mission for underwater pipeline inspection by a single AUV, implemented as a ROS 2-based system. This mission must be completed while simultaneously accounting for uncertainties such as thruster failures and unfavorable environmental conditions. The paper discusses how SUAVE can be used with different self-adaptation frameworks, illustrated by an experiment using the Metacontrol framework to compare AUV behavior with and without self-adaptation. The experiment shows that the use of Metacontrol to adapt the AUV during its mission improves its performance when measured by the overall time taken to complete the mission or the length of the inspected pipeline.**

***Index Terms*—exemplar, self-adaptation, robotics, underwater robots, Metacontrol, SUAVE**

I. Introduction

Autonomous robots are an excellent case for applying self-adaptation techniques [1]–[7]. These robots face uncertainty in their operation stemming from both the system (e.g., sensor failures) and the environment (e.g., different terrains). They need to complete their missions despite such uncertainty [8] with minimal or no human supervision [9]. A subclass of these robots, autonomous underwater vehicles (AUVs) [10] which are used for, e.g., subsea observation, are particularly challenging: once they have been deployed in the real world, they need to take both low-level (e.g., increase thruster power) and high-level (e.g., dive deeper) adaptive decisions without *any* human supervision.

Self-adaptive systems can be implemented as two-layered systems consisting of a *managed* and a *managing* subsystem [11]. The managed subsystem handles the domain concerns, while the managing subsystem implements the adaptation logic and exploits functional alternatives of the managed subsystem to handle the self-adaptation process.

This paper proposes the exemplar SUAVE[1] to facilitate research in the challenging domain of self-adaptive AUVs and to allow the comparison of different self-adaptation strategies. SUAVE is based on ROS 2 – one of the most widely adopted robotics software frameworks [12]. This ensures that the system built for SUAVE can (i) run directly on real robots and not only in simulation environments, (ii) serve as a basis for other adaptive robotic missions, and (iii) be easily extended with new functionalities and adaptation concerns. The exemplar is publicly available at https://github.com/kas-lab/suave.

The exemplar focuses on the scenario of *pipeline inspection* for a single AUV. The AUV's mission is to first search for a pipeline on the seabed, then follow and inspect the pipeline. The functionalities required to accomplish this mission are implemented in the *managed subsystem* of SUAVE. During the execution, of the mission, two types of uncertainties are considered: component failures in the form of thruster failures (e.g., due to debris getting stuck in a thruster) and changes in the environmental conditions in the form of changes in the water visibility (e.g., due to currents disturbing sediment from the seabed). While the first uncertainty may impact the robot's motion by making it move unexpectedly, the second impacts the efficiency of the pipeline search and detection by forcing the robot to be closer to the seabed to detect the pipeline in case of poor water visibility, which results in a smaller field of view while searching.

The exemplar enables the development of a *managing subsystem* to address the previous uncertainties. The managing subsystem should be able to monitor the current runtime circumstances, recover the AUV's thrusters in case of a thruster failure, and adjust the AUV's path generation algorithm to account for changes in water visibility.

To illustrate the use of adaptation frameworks in SUAVE, the managing subsystem was implemented with Metacon-

This work was supported by the European Union's Horizon 2020 Framework Programme through the MSCA network REMARO (Grant Agreement No 956200).

[1]Self-adaptive Underwater Autonomous Vehicle Exemplar.

trol [13], [14], a framework that enables self-adaptation in robotic systems and promotes the reuse of the adaptation logic by exploiting a model of the managed subsystem at runtime. Metacontrol's strength lies in the separation between the application and adaptation concerns, i.e., in the separation between the robot's operation and the logic of when and how to adapt. This separation of concerns allows the adaptation logic to be reused in a straightforward way in different applications. However, it is important to highlight that even though SUAVE is equipped with a Metacontrol-based adaptation logic, the exemplar can also be used without Metacontrol, which in addition allows for comparing other approaches to Metacontrol-based ones.

In summary, the contributions of this paper are:

- a *self-adaptation exemplar for AUVs using ROS 2* that can be equipped with different adaptation logics, enables the comparison of different self-adaptation strategies, forms a basis for other adaptive robotic missions, and can run both on real robots and in simulation environments; and
- a *Metacontol-based adaptation logic formulation* that can serve as a baseline for future research and as a benchmark for self-adaptation strategies, and is easily reusable for other robotic and non-robotic applications.

Paper outline. Section II presents related work, after which Section III further details the use case and the overall architecture. The managed subsystem is described in Section IV, while Section V discusses the managing subsystem and how Metacontrol is applied to the use case. Section VI briefly explains how the exemplar can be reused and extended, and Section VII presents and discusses the results of applying Metacontrol. Finally, Section VIII concludes the paper.

II. Related work

The UNDERSEA exemplar by Gerasimou *et al.* [7] provides an AUV simulation in which the robot performs self-adaptation to deal with uncertainties such as sensor failures and changing goals. SUAVE is related to UNDERSEA as both address the domain of self-adaptive AUVs. However, a key difference is the underlying libraries used to develop software for the robot. UNDERSEA uses MOOS-IvP while SUAVE uses ROS 2, a more widely used framework that is considered state of the art in the robotics research community, which contributes to the reusability and extensibility of SUAVE.

There have been previous exemplars that do use ROS, in particular, the Body Sensor Network by Gil *et al.* [15]. However, its application differs significantly from SUAVE as it concerns health monitoring through a series of sensors rather than a robot vehicle fulfilling a mission autonomously.

Cheng *et al.* proposed AC-ROS [6], a framework which uses assurance cases to endow a ROS-based system with self-adaptive capabilities. Specifically, it concerns an 'EvoRally' vehicle, a terrestrial robot tasked with patrolling an environment as its mission, while meeting requirements such as energy efficiency. The authors do not provide the source code of the proposed system, which means it does not serve as an exemplar as SUAVE does.

The paper by Bozhinoski *et al.* [14] concerns an earlier iteration of using MROS for runtime adaptation similar to this paper. Their work revolves around two cases, a manipulator robot with a "pick and place" task and a mobile robot navigating around obstacles on a factory floor. Both of the use cases show a need to deal with uncertainties, e.g., with a safety concern by disabling one of the pick and place arms. When compared to SUAVE, the key differences are the migration from ROS to ROS 2, as well as the use case being an AUV rather than a manipulator or mobile terrestrial robot.

III. Pipeline inspection exemplar

This section describes the use case and system architecture, the two system layers are detailed in Sections IV and V.

A. Use case description

The use case in this exemplar is about an AUV inspecting pipelines located on a seabed. Its mission consists of two sequential tasks, $(T1)$ searching for the pipeline, then $(T2)$ simultaneously following and inspecting the pipeline.

When performing its mission, the AUV is subject to two sources of uncertainty that could trigger self-adaptation: $(U1)$ thruster failures and $(U2)$ changes in water visibility. $U1$ arises from the possibility of the AUV's thrusters failing at runtime, which may cause the AUV to move unexpectedly. This is relevant for both $T1$ and $T2$. To overcome $U1$, the managed subsystem of the AUV contains functional alternatives. When one or more thrusters fail, it is possible to enter a recovery state in which the thrusters are recovered. $U2$ influences the maximum distance at which the AUV can visually perceive objects. This is relevant for $T1$, higher water visibility allows the AUV to search for the pipeline at higher altitudes, resulting in a larger field of view and the possibility of discovering the pipeline faster. On the other hand, if the water visibility is low, the AUV has to move closer to the seabed to search for the pipeline, which limits its field of view and therefore may lead to a longer time to discover the pipeline. Thus, changing the altitude of the AUV provides functional alternatives for dealing with $U2$.

This exemplar focuses on the problem of overcoming $U1$ and $U2$ using a self-adaptation logic, implemented by a managing subsystem, that can be extended and reused for other sources of uncertainty. The managing subsystem shall overcome $U1$ by recovering the failed thrusters at runtime, and $U2$ by adapting the maximum altitude for the path generator algorithm according to the measured water visibility. Thus, by reacting to $U1$ and $U2$, the managing subsystem increases the reliability and performance of the system.

For the feasibility of the exemplar, the use case was simplified while still allowing for a worthwhile application of self-adaptation to an AUV. It is important to highlight that a realistic operation of an AUV used for pipeline inspection would include steps that are related to pre-dive, launching and recovery, human interaction, and intermediary missions that are necessary to enable the inspection. Furthermore, there are

several sources of uncertainty not considered here, including ocean dynamics, sensor failures, and battery duration.

B. System Architecture

To accomplish the mission described in Section III-A, the managed subsystem requires the functions represented in Fig. 1. $T1$ requires the functions `Control Motion`, `Maintain Motion`, `Localization`, `Detect Pipeline`, `Generate Search Path`, and `Coordinate Mission`, while $T2$ requires the functions `Control Motion`, `Maintain Motion`, `Localization`, `Detect Pipeline`, `Follow Pipeline`, `Inspect Pipeline`, and `Coordinate Mission`. During runtime, the functions must be activated and deactivated according to the task being performed.

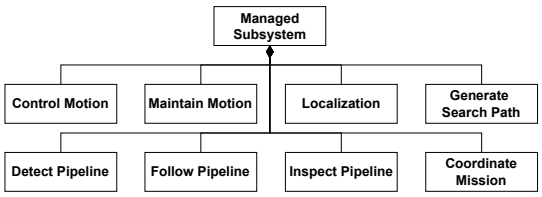

Fig. 1. Managed Subsystem's Functional Hierarchy

To overcome the uncertainties $U1$ and $U2$, a managing subsystem requires the functionalities to *monitor* the environment and the managed subsystem's internal state, *reason* about it, and *execute* the managed subsystem's reconfiguration.

The required functions of the managed and managing subsystems are realized as depicted in Fig. 2. The managed subsystem is detailed in Section IV and the managing subsystem in Section V. It is important to mention that managed subsystem functions `Control Motion` and `Localization` are achieved by `ArduSub`, and the function `Inspect Pipeline` is not realized since the actual inspection of the pipeline is not the focus of this work. It is also important to highlight that this exemplar implements the function to *reason* about the managing subsystem with Metacontrol to provide a baseline for future research. However, it can be replaced with other solutions, as long as they are compatible with the monitor and execute interfaces, as described in Section VI.

IV. Managed Subsystem

The managed subsystem is implemented as a ROS 2-based system and is depicted in Fig. 2. The only non-ROS 2 component is `ArduSub`[2], which is an open-source autopilot for underwater vehicles. In this application it is used to solve the functions `Control Motion` and `Localization`[3]. The `MAVROS` package works as a bridge between `ArduSub` and the ROS 2 components. The `Detect Pipeline` node detects the pipeline and informs `Follow Pipeline` and the `Coordinate Mission` node about its position[4]. The `Coordinate Mission` node coordinates the tasks' execution

[2]https://www.ardusub.com/

[3]It is assumed that the AUV has appropriate sensors for localization

[4]A mock perception system is used.

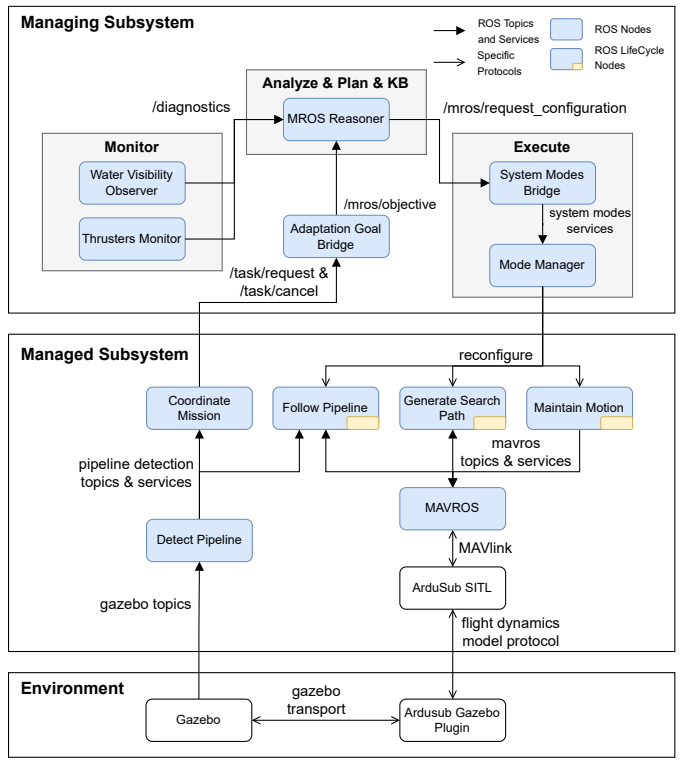

Fig. 2. System Architecture

and sets the adaptation goals. Note that the function `Inspect Pipeline` is not implemented, since the actual inspection of the pipeline is not the focus of this work. However, the exemplar can easily be extended with this functionality by adding a new node that implements the pipeline inspection.

`Follow Pipeline`, `Generate Search Path`, and `Maintain Motion` are lifecycle nodes, which means that they have internal states, such as *active* and *inactive*, and it is possible to switch between these states at runtime. Furthermore, the System Modes package [16] extends the state *active* with additional modes, e.g., *active.low_altitude*.

To adapt the managed subsystem, the managing subsystem adapts the lifecycle nodes by changing their states. This is done by the `Mode Manager` node, which is used off-the-shelf from the System Modes package. The states available for `Generate Search Path` are `deactivated`, `low altitude`, `medium altitude`, and `high altitude`. Subsequently, the states available for `Follow Pipeline` are `deactivated` and `activated`, while the states for `Maintain Motion` are `deactivated`, and `recover thrusters`.

To enable other developers of self-adaptive systems to use this exemplar and compare different approaches, a `Gazebo`-based [5] simulation of a pipeline inspection environment and a model of the AUV is provided. The BlueROV2[6] robot was selected as the AUV for the exemplar because (i) it is compatible with `ArduSub`; (ii) it is easily integrated with

[5]https://gazebosim.org/home

[6]https://bluerobotics.com/store/rov/bluerov2/

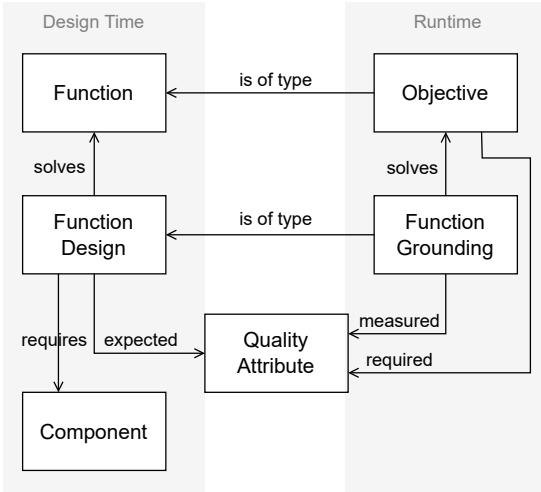

Fig. 3. A simplified representation of the TOMASys elements

`Gazebo` via plugins; and (iii) the robot has a low price compared to other available AUVs, making it more accessible to researchers to reproduce the exemplar with a real robot.

V. Managing Subsystem

The managing subsystem exploits functional alternatives of the managed subsystem to enable adaptation and thereby increase system reliability. Metacontrol is used as an example of how a managing subsystem can be implemented. This section introduces Metacontrol and shows how the adaptation problem can be formulated and implemented with Metacontrol.

A. Metacontrol Background

Metacontrol uses the MAPE-K feedback loop [17], [18] to implement self-adaptation. It *Monitors* the managed subsystem during runtime, *Analyzes* whether the system meets its requirements, *Plans* a new configuration if the system does not meet the requirements, and then *Executes* the reconfiguration of the managed subsystem. All this is done using a shared *Knowledge Base* to which each step refers. In Metacontrol, the knowledge base conforms to the TOMASys (Teleological and Ontological Metamodel for Autonomous Systems) metamodel [13].

A simplified version of the TOMASys metamodel is displayed in Fig. 3. TOMASys uses *functions* F to represent the functionalities of the system, e.g., generating a search path for the AUV. The architectural variants that implement these requirements are captured by *function designs* $FD(F, \mathcal{C}, \mathcal{QA}^{exp})$. To distinguish during runtime which function design is most suited in a given situation, a set $\mathcal{QA}^{exp}$ of *expected quality attributes* is associated with it. An expected quality attribute value reflects how well a function design is supposed to fulfill the function F it solves. Furthermore, a function design requires a set $\mathcal{C}$ of *components* of the managed subsystem to solve F. A component $C(S_C)$ is a piece of hardware or software, e.g., a sensor or a path-planning algorithm, respectively. The status S_C of a component indicates its availability, i.e., whether it is functioning or not.

An *objective* $O(F, S_O, \mathcal{QA}^{req})$ is a runtime instantiation of a function F, e.g., generating a search path with a minimum required water visibility, whose status S_O reflects whether the objective is currently achieved. Furthermore, the set $\mathcal{QA}^{req}$ of *required quality attributes* specifies which quality attribute value the objective requires in order to work properly.

An objective O is solved by a *function grounding* $FG(O, FD, S_{FG}, \mathcal{QA}^{meas})$, which represents the function design FD that is currently used to solve the objective. Its status S_{FG} reflects whether the function grounding is currently able to achieve the objective. The set $\mathcal{QA}^{meas}$ of *measured quality attributes* reflects how well the function grounding currently fulfills O and is computed using sensor data.

B. Metacontrol Formulation

The functions, architectural variants, and quality attributes required to solve the tasks $(T1)$ Search Pipeline and $(T2)$ Inspect Pipeline, described in Section III, are modeled conforming to the TOMASys metamodel. Table I specifies the functions (F_1) maintain_motion, (F_2) generate_search_path, and (F_3) follow_pipeline, while Table II describes the quality attributes (QA_1) water_visibility, and (QA_2) performance. Functions F_1 and F_2 are required to achieve $T1$, whereas $T2$ is achieved by F_1 and F_3. The function designs that solve these functions are specified in Table III. The set of required components is empty for function designs $FD_2 - FD_6$ because they do not require any components that are susceptible to adaptation, or used in the reasoning process.

Since the objectives and function groundings are instantiated during runtime, they are not specified here. An objective for function F_2 is for example to generate a search path with no required quality attribute, which is defined as $O_2(F_2, ok, \varnothing)$ in the notation introduced above. A possible function grounding for this objective is $FG_2(O_2, FD_4, ok, \{QA_1^{meas} = 1.1\})$.

The MAPE-K loop steps in this exemplar are formulated as follows. The monitor step is responsible for measuring QA_1^{meas} and for monitoring the state of the six thrusters.

The analyze step uses Horn rules to reason about the knowledge base. One example rule that analyzes whether the measured water visibility QA_1^{meas} still satisfies the expected water visibility QA_1^{exp} of the grounded function design is displayed in Fig. 4. Note that it is written in terms of the notation introduced in Section V-A. Line 1 expresses that the rule reasons about a function grounding FG that solves an objective O, is of type FD, has a status S_{FG} and an associated set of measured quality attributes $\mathcal{QA}^{meas}$. Furthermore, the function design FD solves the function F, has a set of required components $\mathcal{C}$ and an associated set of expected quality attributes $\mathcal{QA}^{exp}$. Note that it is implicitly assumed that FG is well-formed, i.e., that the function of which O is a type of is the same as the function that FD solves. Since this rule should analyze the water visibility, Line 2 ensures that QA_1^{meas} is an element of the set $\mathcal{QA}^{meas}$ and that QA_1^{exp} is an element of the set $\mathcal{QA}^{exp}$, i.e., that both FG and FD are related to water visibility. Finally, if the measured value of QA_1 is less than its expected value associated with the

TABLE I
THE TOMASYS FUNCTIONS USED FOR THE EXEMPLAR

Function	Name	Requirement
F_1	maintain_motion	Maintain the motion of the robot
F_2	generate_search_path	Generate a path to search for the pipeline
F_3	follow_pipeline	Follow and inspect the pipeline

TABLE II
THE TOMASYS QUALITY ATTRIBUTES USED FOR THE EXEMPLAR

QA	Name	Unit	Description
QA_1	water_visibility	$[0, \infty)$	Reflects the maximum altitude (in meters) from which the AUV can perceive the seabed
QA_2	performance	$[0, 1]$	Reflects how efficient the current search strategy is

TABLE III
THE TOMASYS FUNCTION DESIGNS USED FOR THE EXEMPLAR

Function Design	Name	Description
$FD_1(F_1, \{\text{thruster_}x \mid x = 1, \ldots, 6\}, \{QA_2^{exp} = 1\})$	all_thrusters	Uses all thrusters
$FD_2(F_1, \varnothing, \{QA_2^{exp} = 0.5\})$	recover_thrusters	Recovers the thrusters that are in failure
$FD_3(F_2, \varnothing, \{QA_1^{exp} = 0.5, QA_2^{exp} = 0.25\})$	spiral_low	Produces a spiral search path with low altitude
$FD_4(F_2, \varnothing, \{QA_1^{exp} = 1, QA_2^{exp} = 0.5\})$	spiral_medium	Produces a spiral search path with medium altitude
$FD_5(F_2, \varnothing, \{QA_1^{exp} = 2, QA_2^{exp} = 1\})$	spiral_high	Produces a spiral search path with high altitude
$FD_6(F_3, \varnothing, \varnothing)$	follow_pipeline	Follows the pipeline

$$FG\big(O, FD, S_{FG}, \mathcal{QA}^{meas}\big) \wedge \; FD(F, \mathcal{C}, \mathcal{QA}^{exp}) \tag{1}$$
$$\wedge \; QA_1^{meas} \in \mathcal{QA}^{meas} \; \wedge \; QA_1^{exp} \in \mathcal{QA}^{exp} \tag{2}$$
$$\wedge \; QA_1^{meas} < QA_1^{exp} \tag{3}$$
$$\Rightarrow \; S_{FG} = \mathit{error} \tag{4}$$

Fig. 4. Rule to analyze whether the measured water visibility QA_1^{meas} still satisfies the expected water visibility QA_1^{exp} of the grounded function design

grounded function design, see Line 3, then the status of the function grounding is set to *error*, see Line 4.

In the planning step, the function designs with QA_1^{exp} higher than QA_1^{meas} are filtered out as the visibility they would expect is not measured, afterward the remaining function design with the highest expected search performance (QA_2^{exp}) is selected as the desired configuration. The selected configuration is then carried out in the execute step.

C. Metacontrol Implementation

As depicted in Fig. 2, the monitor step is implemented with the `Water Visibility Observer` and the `Thruster Monitor` nodes. They are used for measuring QA_1 and monitoring the status of the six thrusters thruster_x where $x \in \{1, \ldots, 6\}$, respectively. To simplify the system and avoid the addition of unnecessary nodes, instead of adding water visibility to the Gazebo simulator, the `Water Visibility Observer` simulates water visibility measurements with a sine function, and instead of probing the managed subsystem to identify thruster failures the `Thruster Monitor` simulates the thruster failures events. Since the monitor step is mocked up, and its probes and intermediary nodes that would be required to provide the probes are not implemented, they are not included in Fig. 2. Both nodes publish their data into the `/diagnostics` topic with the ROS 2 default `DiagnosticArray` message type.

The knowledge base (KB), the analyze and plan step are implemented using MROS [7] [19], a ROS 2-based Metacontrol implementation, as the `MROS Reasoner` node. The KB is implemented with the Ontology Web Language (OWL) [20], the Horn rules used for the analyze step with the Semantic Web Rule Language (SWRL) [21], and the reasoning is done with Pellet[8]. The `MROS Reasoner` receives water visibility measurements (QA_1^{meas}) and thruster status information from the monitor step, then decides whether adaptation is required, and, in this case, selects a desired configuration which it sends to the execute step (see Section V-B for more details). The `MROS Reasoner` initially does not have objectives, so it does not perform adaptation. The adaptation reasoning only starts when the `Coordinate Mission` node sends new objectives, such as $O_2(F_2, null, \varnothing)$, via the `Adaptation Goal Bridge`. New objectives do not have a status yet.

The execute step uses the System Modes' `Mode Manager` to adapt the managed subsystem, and the `System Modes Bridge` bridges the `Mode Manager` with the `MROS Reasoner`. When a reconfiguration is needed, the `MROS Reasoner` requests the new configuration via the `/mros/request_configuration` service to the `System Modes Bridge`.Then the `System Modes Bridge` forwards the request to the `Mode Manager` using the correct service names, depending on the lifecycle node being adapted. The services used by the `Mode Manager` are listed in Table IV, and the available modes are listed in Table V.

VI. EXTENDING AND CONNECTING MANAGING SUBSYSTEMS

With the described system implementation, the only Metacontrol-specific nodes of the system are the `MROS Reasoner`, the `System Modes Bridge`, and the `Adaptation Goal Bridge`. All other nodes of the system can be reused with different managing subsystems. The only requirement to connect a managing subsystem to the managed subsystem is to ensure that the managing subsystem adheres to the provided monitor and execute ROS 2 interfaces. As described in the previous section, the monitor interface is the ROS 2 topic `/diagnostics`, and the execute interfaces are listed in Table IV. To show that changing the managing subsystem is possible, a managing subsystem that randomly picks a configuration was also implemented.

Since the system is implemented with a modular design, it can be extended with additional functionalities and adaptation

[7] https://github.com/meta-control/mc_mros_reasoner
[8] https://github.com/stardog-union/pellet

TABLE IV
AVAILABLE SYSTEM MODES' SERVICES

Node	Service
f_generate_search_path	/f_generate_search_path/change_mode
f_follow_pipeline	/f_follow_pipeline/change_mode
f_maintain_motion	/f_maintain_motion/change_mode

TABLE V
AVAILABLE MODES

Node	Mode	Lifecycle state
f_generate_search_path	fd_spiral_high	active
f_generate_search_path	fd_spiral_medium	active
f_generate_search_path	fd_spiral_low	active
f_generate_search_path	fd_unground	inactive
f_follow_pipeline	fd_follow_pipeline	active
f_follow_pipeline	fd_unground	inactive
f_maintain_motion	fd_all_thrusters	inactive
f_maintain_motion	fd_recover_thrusters	active

scenarios by adding new lifecycle nodes and updating the system modes' configuration file accordingly. The implemented functionalities can be replaced with different implementations as long as they adhere to the same interfaces; e.g., the `Pipeline Detection` node could be replaced by a node that actually performs perception instead of a mock-up.

VII. EVALUATION

To evaluate the performance of different managing subsystems using this exemplar, the mission described in Section III was implemented. The mission consists of the AUV performing $T1$ and $T2$ while subject to $U1$ and $U2$ until a user-provided time limit is reached. To evaluate the mission, the following metrics were used: the *search time*, the amount of time elapsed from the beginning of the search until the pipeline is found, and the total *distance inspected* of the pipeline.

To provide a baseline for the exemplar, the mission is performed with two different managing subsystems and with no managing subsystem, using a fixed configuration. The managing subsystems are the Metacontrol-based implementation detailed in Section V and a random managing subsystem that selects configurations arbitrarily.

Since the system is non-deterministic due to characteristics of `Gazebo`, `ArduSub`, and the interaction between them, no run of the simulation is exactly the same. Thus, the mission execution and metrics collection are automated with a runner to allow multiple runs to be easily performed.

This section briefly describes how to configure the exemplar, and the results of running the exemplar. Further details may be found in the exemplar repository.

A. Configuring the exemplar

In SUAVE, the AUV's mission execution can be varied by changing the parameters of the system. In the `Water Visibility Observer`, the available parameters are the water visibility minimum and maximum values, periodicity, and initial phase shift. In the `Thrusters Monitor`, the available parameter is a list with thruster events indicating which thruster fails and when. In the `Coordinate Mission`, the mission time limit can be set. In the random manager, the adaptation periodicity can be set, and when using no manager, the default states for the lifecycle nodes can be set. In addition, the runner is parametrized with the number of runs to execute, and which managing subsystem to select. All parameters are adjusted using configuration files packaged in the exemplar.

TABLE VI
MISSION RESULTS

Managing subsystem	Number of runs	Search time (s)		Distance inspected (m)	
		Mean	Std	Mean	Std
None	20	187.85	42.40	31.40	13.00
Random	20	180.15	82.05	3.88	3.19
Metacontrol	**20**	**106.95**	**39.46**	**51.15**	**12.01**

B. Results

The mission was executed with a time limit of 300 seconds, water visibility periodicity of 80 seconds, minimum and maximum values of 1.25 and 3.75, no phase shift, and thruster 1 failing after 35 seconds from the start of the mission. The results are shown in Table VI. It can be noticed that with the Metacontrol managing subsystem both mean *search time*[9] is lower, and the *distance inspected* is higher. This indicates that, in this exemplar, Metacontrol improves the performance of the system, and outperforms the random managing subsystem and the system without a managing subsystem. In addition, the standard deviation (Std) of the *search time* is lower for Metacontrol, indicating that it is more consistent when searching for the pipeline. The Std of the random manager for the *distance inspected* is lower, however, its mean value is also low, indicating that the random manager is consistent in not inspecting the pipeline. The results shown can be used as a baseline for comparing different managing subsystems.

VIII. CONCLUSION

This work describes SUAVE, a ROS 2-based exemplar for self-adaptive underwater vehicles used for pipeline inspection. Due to its modular design, SUAVE enables different managing subsystems to be applied to the system without the need to modify the managed subsystem, the monitor nodes, and the executing mechanism. In addition, the system can be easily extended with new functionalities and adaptation scenarios by adding new nodes. Furthermore, this paper provides a baseline for comparing the performance of different managing subsystems, and it shows that the addition of a Metacontrol-based managing subsystem increases the performance of the system in comparison to not using any managing subsystem or one that chooses configurations arbitrary.

In future work, SUAVE can be extended with: more metrics for a more in-depth evaluation; more tasks (e.g., docking), functionalities (e.g. a de facto perception system), and components (e.g. sonars) for more realistic missions; more adaptation scenarios, e.g., adapting to changes in the water currents, and adapting the thruster configuration matrix when a thruster can not be recovered.

[9]When the pipeline is not found, the time limit is used as the *search time*

REFERENCES

[1] G. Edwards, J. Garcia, H. Tajalli, D. Popescu, N. Medvidovic, G. Sukhatme, and B. Petrus, "Architecture-driven self-adaptation and self-management in robotics systems," in *2009 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, pp. 142–151, 2009.

[2] J. Cámara, B. Schmerl, and D. Garlan, "Software architecture and task plan co-adaptation for mobile service robots," in *Proceedings of the IEEE/ACM 15th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, (Seoul Republic of Korea), pp. 125–136, ACM, June 2020.

[3] Y.-S. Park, H.-M. Koo, and I.-Y. Ko, "A task-based and resource-aware approach to dynamically generate optimal software architecture for intelligent service robots," *Software: Practice and Experience*, vol. 42, no. 5, pp. 519–541, 2012. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.1074.

[4] Y.-J. Shin, L. Liu, S. Hyun, and D.-H. Bae, "Platooning LEGOs: An Open Physical Exemplar for Engineering Self-Adaptive Cyber-Physical Systems-of-Systems," in *2021 International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pp. 231–237, May 2021. ISSN: 2157-2321.

[5] M. Askarpour, C. Tsigkanos, C. Menghi, R. Calinescu, P. Pelliccione, S. García, R. Caldas, T. J. von Oertzen, M. Wimmer, L. Berardinelli, M. Rossi, M. M. Bersani, and G. S. Rodrigues, "RoboMAX: Robotic Mission Adaptation eXemplars," in *2021 International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pp. 245–251, May 2021. ISSN: 2157-2321.

[6] B. H. C. Cheng, R. J. Clark, J. E. Fleck, M. A. Langford, and P. K. McKinley, "Ac-ros: Assurance case driven adaptation for the robot operating system," in *Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, MODELS '20, (New York, NY, USA), p. 102–113, Association for Computing Machinery, 2020.

[7] S. Gerasimou, R. Calinescu, S. Shevtsov, and D. Weyns, "UNDERSEA: An exemplar for engineering self-adaptive unmanned underwater vehicles," in *2017 IEEE/ACM 12th International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pp. 83–89, 2017.

[8] M. Ludvigsen, "Collaborating robots sample the primary production in the ocean," *Science Robotics*, vol. 6, no. 50, p. eabf4317, 2021.

[9] H. Huang, E. Messina, R. Wade, R. English, B. Novak, and J. Albus, "Autonomy measures for robots," in *Proceedings of the 2004 ASME International Mechanical Engineering Congress & Exposition, Anaheim, California*, pp. 1–7, 2004.

[10] R. B. Wynn, V. A. Huvenne, T. P. Le Bas, B. J. Murton, D. P. Connelly, B. J. Bett, H. A. Ruhl, K. J. Morris, J. Peakall, D. R. Parsons, E. J. Sumner, S. E. Darby, R. M. Dorrell, and J. E. Hunt, "Autonomous underwater vehicles (auvs): Their past, present and future contributions to the advancement of marine geoscience," *Marine Geology*, vol. 352, pp. 451–468, 2014. 50th Anniversary Special Issue.

[11] D. Weyns, *An Introduction to Self-adaptive Systems: A Contemporary Software Engineering Perspective*. John Wiley & Sons, 2020.

[12] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, "Robot operating system 2: Design, architecture, and uses in the wild," *Science Robotics*, vol. 7, no. 66, p. eabm6074, 2022.

[13] C. H. Corbato, *Model-based self-awareness patterns for autonomy*. PhD thesis, Universidad Politécnica de Madrid, 2013.

[14] D. Bozhinoski, M. G. Oviedo, N. H. Garcia, H. Deshpande, G. van der Hoorn, J. Tjerngren, A. Wasowski, and C. H. Corbato, "MROS: runtime adaptation for robot control architectures," *Advanced Robotics*, vol. 36, no. 11, pp. 502–518, 2022.

[15] E. B. Gil, R. Caldas, A. Rodrigues, G. L. G. da Silva, G. N. Rodrigues, and P. Pelliccione, "Body sensor network: A self-adaptive system exemplar in the healthcare domain," in *2021 International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*, pp. 224–230, 2021.

[16] A. Nordmann, R. Lange, and F. M. Rico, "System modes-digestible system (re-) configuration for robotics," in *2021 IEEE/ACM 3rd International Workshop on Robotics Software Engineering (RoSE)*, pp. 19–24, IEEE, 2021.

[17] Y. Brun, G. D. M. Serugendo, C. Gacek, H. Giese, H. M. Kienle, M. Litoiu, H. A. Müller, M. Pezzè, and M. Shaw, "Engineering self-adaptive systems through feedback loops," in *Software Engineering for Self-Adaptive Systems*, vol. 5525 of *Lecture Notes in Computer Science*, pp. 48–70, Springer, 2009.

[18] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, 2003.

[19] D. Bozhinoski, M. G. Oviedo, N. H. Garcia, H. Deshpande, G. van der Hoorn, J. Tjerngren, A. Wasowski, and C. H. Corbato, "MROS: runtime adaptation for robot control architectures," *Adv. Robotics*, vol. 36, no. 11, pp. 502–518, 2022.

[20] G. Antoniou and F. van Harmelen, "Web ontology language: OWL," in *Handbook on Ontologies* (S. Staab and R. Studer, eds.), International Handbooks on Information Systems, pp. 67–92, Springer, 2004.

[21] I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, M. Dean, *et al.*, "SWRL: A semantic web rule language combining OWL and RuleML," *W3C Member submission*, vol. 21, no. 79, pp. 1–31, 2004.