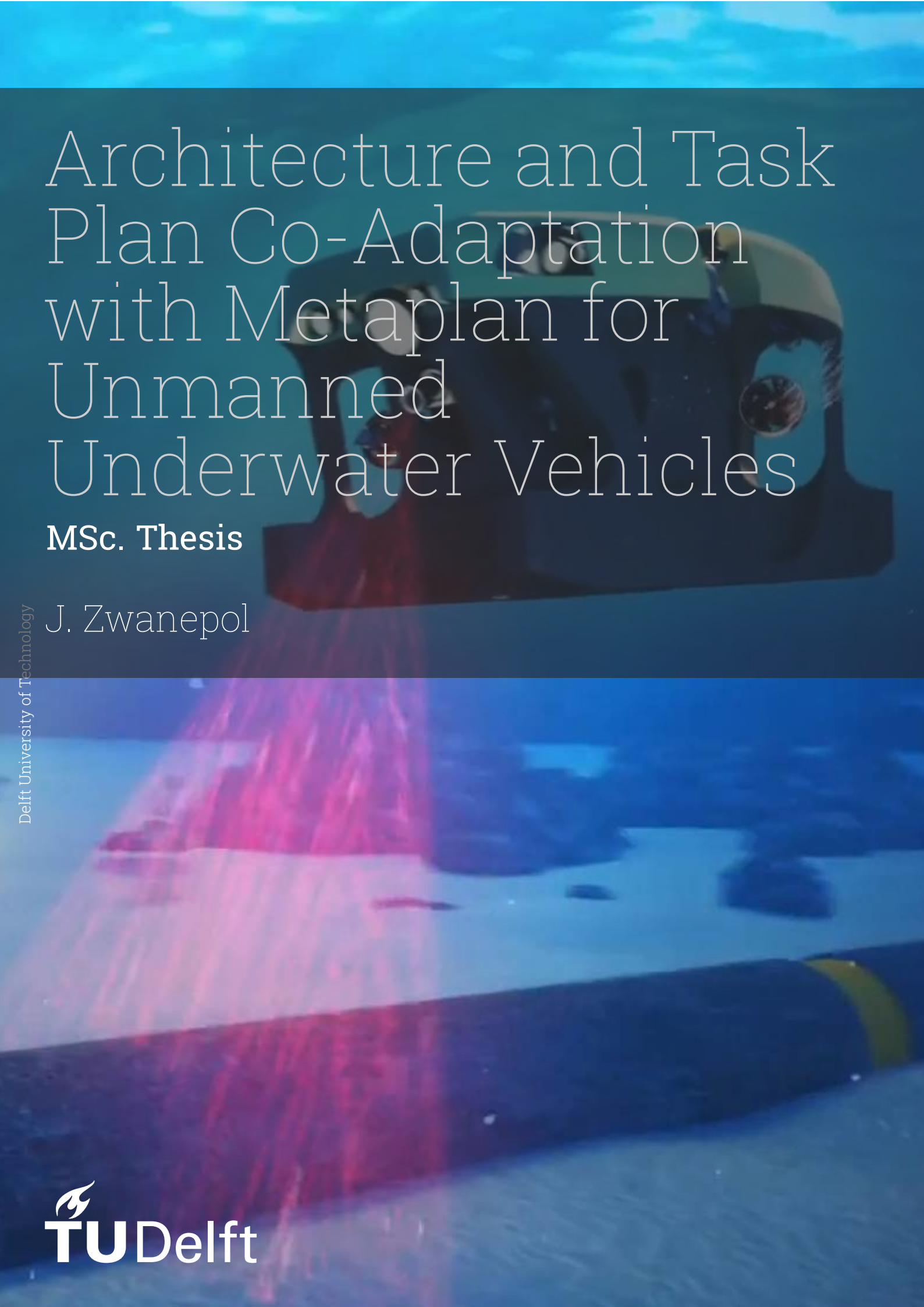

Architecture and Task Plan Co-Adaptation with Metaplan for Unmanned Underwater Vehicles

MSc. Thesis

J. Zwanepol

Delft University of Technology

TU Delft

Architecture and Task Plan Co-Adaptation with Metaplan for Unmanned Underwater Vehicles

MSc. Thesis

by

J. Zwanepol

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Tuesday March 28, 2023 at 1:00 PM.

Student number:	4472764
Project duration:	December 1, 2021 – March 28, 2023
Thesis committee:	Dr. Ir. C. Hernández
	G.R. Silva
	Dr. Ir. J. Kober
	Dr. I. Gerostathopoulos

An electronic version of this thesis is available at http://repository.tudelft.nl/.

TU Delft

Acknowledgement

This is my opportunity to become a bit sentimental and reminisce about my time at the TU Delft. I am really grateful for receiving the opportunity to study Robotics at the TU Delft. My cohort had the opportunity to take the Robotics master on its maiden voyage, and although there were definitely hiccups due to this and the effects of the Covid-19 pandemic, I felt the staff involved with organizing the program put their heart and soul into it. I want to especially thank Karin van Tongeren for her passion in organizing in-person events whenever possible and always trying to interact with students to ensure they are well looked after.

During my master's degree, I had the opportunity to meet a myriad of inspirational and kind people, who helped me grow academically and as a person. There are a few people I would like to thank in particular for their support and kindness during my Robotics journey. First and foremost, thank you Godwin, for being a brother to me and sharing times of great stress (Machine Learning project) and great joy (too many things to mention), and stuffing me with biryani beyond what my stomach could handle. Second, I want to thank Zhengping for being a great friend and always being up to grab lunch/coffee and letting me taste delicious Chinese dishes. Thank you Sriman, for inviting me into your home for delicious meals and being an inspiration for academic excellence. Thank you Stan, for starting your thesis track earlier, such that I could look over your shoulder and learn from your experiences, this was a great help to me. Thank you also for being always interested and offering your help where you could. Thank you Mariano and Giovanni for obsessing over the ERF hackathon and writing a paper with me and testing my leadership skills by your endless zeal for implementing 'trivial' improvements. Finally, I want to thank the people of the office I had the privilege to work in. Thank you Juliane, Julian, Leandro, Lorenzo, Irene, Ashwin, and Micah for baking delicious things, talking with me, providing me with coffee, offering help, and distracting me from the endless thesis work. Honestly, there are too many people to thank, therefore I present a (probably incomplete) list of honorable mentions, in no particular order: Shawn, Wouter Meijer, Denzil, Nander, Cilia, Anish, Berry, Darshan, Joseph, and Georgios.

As for the supervisors of my thesis, I want to thank you for all the support and feedback you have given me. I want to thank Carlos for always trying his best to schedule time whenever I asked for this, and for all the discussions helping me clarify ideas and ultimately help me form my thesis. Furthermore, I want to thank Gustavo for helping me persevere in my thesis. You have been such a great help for me throughout my thesis process. You have helped me with providing feedback, debugging, challenging me, and encouraging me to come to where I am now. I appreciate all the coffees we got together and all the non-thesis related talks. Thank you for involving me in the publication of the SUAVE exemplar paper and pushing me to be a better student. Furthermore, I want to thank all the members of the KAS lab for providing feedback and challenging my ideas.

Lastly, I want to thank all the people that have walked alongside me during my graduation process, supporting me with distractions, care, and prayer. Thank you Thijs, Hendrik-Jaap, Ries, Stijn, and Constantijn for all the lunchtime distraction you provided me. Thank you Timo and Lisanne for your sympathy, care, and encouragements and for being good friends. Thank you Jacob, for welcoming us into your home for a short recharge vacation. Thank you Chiem, for being a great flatmate at the start of my degree helping me where possible and for being a good friend. Thank you Max for being a good friend and looking out for me. Thank you to my family and in-law family, for all the support you have given me. Thank you for sending messages to check in on me, inquiring about my progress, providing much-needed distractions, and caring about me. Thank you mom and dad, for the financial support, hugs, prayers, laughs, distractions, home-cooked meals, care-packages, and so much more. And finally, the most important person of all, thank you to my wife Maaike. Words fail to express my gratitude to you. Thank you for all your endless support in all my endeavors. Thank you for challenging me to be a better person, and looking out for my health when I could sometimes neglect it. To close, I am grateful for this journey and all the blessings I received from God along the way.

J. Zwanepol
Delft, March 2023

Architecture and Task Plan Co-Adaptation with Metaplan for Unmanned Underwater Vehicles

J.M. Zwanepol*
Technical University of Delft

Abstract

Unmanned Underwater Vehicles (UUVs) operate in complex environments and need to be able to adapt to sudden failures, or changes in the environment. To achieve autonomous operation, UUVs must have the ability to self-adapt in such cases. To effectively handle component failures and unexpected events, self-adaptation must be applied to both the architecture and task plan of the UUV. This allows the UUV to modify its architecture to accommodate component failures and adjust its task plan in response to unforeseen events that may render the current plan infeasible. The mutual dependencies between architectural adaptation and task planning pose a significant challenge when determining how to apply adaptation. As a result, the task planner must take into account the implications of architectural adaptation when generating a plan. This paper proposes Metaplan, a modular ROS2-based framework for applying architectural and task plan co-adaptation in a reusable way. Metaplan extends Metacontrol, an architectural self-adaptation framework, with a task planner based on Planning Domain Definition Language (PDDL). The effectiveness of Metaplan is demonstrated by evaluating it on SUAVE, an exemplar for evaluating self-adaptation frameworks for UUVs. Metaplan is shown to outperform a baseline which only makes use of a task planner. Architecture and task plan co-adaptation is demonstrated by presenting the UUV with a sudden drop in battery level, requiring the UUV to adapt both its architecture and its task plan. Furthermore, the reusability of Metaplan is showcased by applying it to a mobile manipulator scenario.

1 Introduction

The interest in researching Unmanned Underwater Vehicles (UUV) is increasing, resulting in significant advances in this field [1, 2]. Common applications of UUVs are performing reconnaissance, inspection, and mapping [3], as well as performing "smart" guided attacks in the military context [2].

*Email address: j.m.zwanepol@student.tudelft.nl

UUVs operate in complex environments, due to nonlinear dynamics, uncertain models, and difficult to model disturbances [4]. Moreover, due to their application in sea/ocean environments, UUVs are not easily retrievable when they have a failure. This means that a UUV needs to handle this complex environment, as well as guarantee safe rendezvous/mission fulfillment in the case of faults.

Aside from the aforementioned, software systems are becoming more complex and, according to a manifesto published by IBM, a continuation of this trend will make it impossible for humans to manage these complex software systems [5]. Therefore, a new way of managing software systems, such as UUVs, is needed, one where software systems can manage themselves. Self-managing systems alleviate the need for human management it, only requiring high-level objectives from administrators [6]. One form of self-management is self-adaptation [7], which will be the focus of this work.

Self-adaptation has been applied by switching between (software) components being used through use of an abstracted model of the system. This is also referred to as architectural self-adaptation [8]. A system that makes use of an architectural self-adaptation framework can be broken down into a managed and a managing sub-system. The managing sub-system is the architectural self-adaptation framework and requests adaptation of the managed subsystem. Metacontrol [3] is an architectural self-adaptation framework. Metacontrol has been applied to a UUV by adapting which thrusters are being used in the case of thruster failures, exploiting the thruster redundancy of the UUV. Self-adaptation has also been applied on the task planning level. Task planning is the generation of a sequence of actions which transition the system from an initial state to a goal state. The Situational Evaluation and Awareness (SEA) framework [9] is a task plan adaptation framework which has been applied to a UUV. SEA makes use of task plan adaptation in order to create a continuous map of an object, despite the UUV losing self-localization while mapping an object.

When deviations from initial/expected situations occur, i.e., failures, it is important to consider both the architecture and task plan of the robot when adapting. When applying this to a UUV use case, the UUV would need to adapt both its task plan and its architectural configuration if it is performing a search action and a sudden drop in battery level prohibits the UUV from reaching a charging station with its current configuration. Its task plan would need to change by

requiring a recharge action at the start of the plan, and its architectural configuration would need to change by switching to a configuration that uses less battery. Therefore, architecture and task plan co-adaptation contributes to achieving long-term, reliable deployment of autonomous robots, such as a UUV [10].The mutual dependencies between architectural adaptation and task planning pose a significant challenge when determining how to apply adaptation [11]. As a result, the task planner must take into account the implications of architectural adaptation when generating a plan. The frameworks presented in [12, 11] focus on combining architectural and task plan adaptation. Although these frameworks do architecture and task plan co-adaptation, the planning solution of both frameworks is tailored to the specific application it is made for and is not reusable for a new application.

This paper proposes Metaplan, a modular ROS2-based framework for applying architectural and task plan co-adaptation in a reusable way. Similar to [11], this work takes an architectural self-adaptation framework and extends this with task planning. Metacontrol is selected as the framework used for architectural self-adaptation, because it has an open-source ROS2 implementation available, called MROS[1] [13], and an application of MROS to the SUAVE exemplar is publically available[2].

Like Metacontrol, Metaplan makes use of an ontology to store information about the system architecture. Metaplan uses the information queried from the ontology to update the task planning problem, which is formulated in PDDL, similar to [9]. Metaplan makes use of the Plansys2[3] [14] package for handling plan generation and execution.

The main contribution of this paper is combining architecture and task plan adaptation with a reusable task planner. Metaplan considers architectural adaptation as part of the task planning problem. Architectural adaptation is done at the start of an action, reconfiguring the robot to use the optimal configuration for each action. As such, multiple architectural adaptations can be incorporated in a plan. The reusability of Metaplan is demonstrated by applying it to a new application.

Furthermore, Metaplan maintains the reusability of Metacontrol, while expanding the self-adaptation options of Metacontrol. Metaplan allows for both the architecture and task plan adaptation parts to be reusable for different tasks. The reusability of the architectural adaptation is maintained from the Metacontrol framework, and the use of PDDL allows the framework to be reusable for different task planning problems. Furthermore, Metaplan makes use of ROS2, allowing for easy communication with other ROS2 systems. The functionalities of the system are achieved by ROS2 nodes, allowing for easy re-use of one or more of these nodes for a different system. Furthermore, PDDL is the most popular language in the planning community [15], allowing for easy re-use of open-source PDDL specifications.

The following section presents the research done in the area of architecture and task plan co-adaptation. section 3 gives background information on the frameworks used to develop Metaplan. section 4 presents the scenario to which Metaplan is applied. Afterward, section 5 explains Metaplan and how it works. Next, section 6 Metaplan is evaluated through three experiments. section 7 contains a discussion on the results obtained in the evaluation section, and finally, section 8 contains the conclusions which can be drawn from this work and recommendations for future works.

2 Related Works

Previous work on Metacontrol mentions the need for architecture adaptation to be combined with task plan adaptation [16].

This multi-level concern is first addressed by the MORPH architecture [17]. A divide and conquer approach is proposed such that both the reconfiguration and behavior strategies do not require explicit knowledge about each other in order to make a decision. The authors propose a three-tier hierarchical architecture following the MAPE-K loop[4] [6].

The top layer is the Goal management layer. The main concern of this layer is anticipating changes in goals, environment, and system capabilities by pre-computing generating behavior and reconfiguration strategies to be used in the subsequent layers.

The middle layer is the Strategy management layer, where the computed behavior and reconfiguration strategies are selected to be sent to the Strategy Enactment layer. Here, the behavior and reconfiguration strategies are checked for consistency.

The bottom layer is the Strategy Enactment layer. It executes the selected behavior and reconfiguration strategies while monitoring the managed system.

The high level strategic concerns are handled in the top layers and the tactical adaptation of components is handled in the lower level.

Although MORPH is a framework for applying architecture and task plan co-adaptation, only a concept is presented. There is no open-source code available and no implementation example is given. All the architectural elements are explained briefly, but no in-depth implementation detail is provided.

Like MORPH, [11] presents a framework for applying adaptation on both the architecture and task planning levels. In similar manner, it does this by separating the architecture reconfiguration from the task planning problem, while taking their mutual dependency into account. The framework makes use of an Alloy [18] specification together with the PRISM

[1] https://github.com/meta-control/mc_mros_reasoner

[2] https://github.com/kas-lab/suave

[3] https://github.com/PlanSys2/ros2_planning_system

[4] MAPE-K stands for Monitor, Analyze, Plan, and Execute through Knowledge

[19] model checker in order to adapt the architecture and task plan of the system.

The framework is specifically designed for robotic navigation in a map with multiple traversable corridors. This problem can be modeled as a graph where the arcs represent the corridors. The robot has a limited set of action primitives, such as move, with which to traverse from an initial position to a goal position. The MoveBase [20] package is used for navigation and communication is done using ROS.

Alloy is used to define the components, connectors, and architectural constraints. The Alloy specification is used in order to generate the legal configurations. A first PRISM specification is subsequently used to generate reconfiguration plans with the according energy cost using the legal configurations. The Dijkstra algorithm is then applied to the graph representing the map to generate all possible paths from the initial node to the goal node. Afterward, another PRISM specification is used which takes the legal configurations and the set of possible paths to the goal position and decides on the best configuration and path combination to reach the goal position according to the defined metrics timeliness, safety, and energy efficiency.

This framework achieves architectural and task plan co-adaptation for the scenario used in the example, however does not generalize to new scenarios. Architectural adaptation is done using Rainbow [21], and is relatively easy to apply to a new situation by providing new domain knowledge. However, task planning is done using a very specific solution for navigation on a graph like map. Applying this solution to a new task planning scenario would require extensive changes. Furthermore, task planning does not take into account that an action might become unavailable when there is no architectural configuration solving the action.

Another framework applying architectural and task plan co-adaptation is TeMoto [12], an open-source ROS-based framework for adaptive autonomous robots. The generation of task plans is done separately from the architectural adaptation, thereby also applying a separation of concerns. TeMoto also allows for multi-robot collaboration, with multiple instances of TeMoto running on each robot. The framework is tested and validated using a real robot.

The backbone of the TeMoto framework is the Resource Registrar (RR). A resource is a component or action of the system. This subsystem requires or provides resources, while keeping track of the clients making use of a resource and its sub-resource dependencies. A resource manager keeps track of a certain resource type and updates the information contained in the RR. Actions contain developer-defined logic for tasks, and are executed by the Action Engine. Tasks such as moving an object can consist of multiple actions, in this case that would be 'picking an object', 'moving to desired object location' and 'placing the object'. These tasks are user defined and can be dynamically loaded to TeMoto for execution.

TeMoto provides a fully developed framework for which to program robot autonomy on task management and resource (architecture) management. However, like the previous framework, generalizability of task planning is lacking. At design time, a task plan for handling each uncertainty needs to be programmed. If the system were to encounter a new scenario, the robot would not be able to generate an appropriate task plan.

Currently, the architectural and task plan co-adaptation approaches are limited with regard to re-usability of task planning. Applying the frameworks mentioned above would require considerable overhead to handle new scenarios. A new framework which can be applied to multiple scenarios with minimal overhead is therefore needed to handle architectural and task plan co-adaptation.

Task planning is done by devising a sequence of actions to achieve a goal [22]. Off-the-shelf task planners exist for solving robotics task planning problems. These task planners make use of a planning language to define the actions and world states of the planning problem. Important considerations in selecting a planning language is that a task plan needs to be able to deal with non-deterministic, dynamic environments. This is because in many robotic scenarios, the robot oftentimes operates in a dynamic environment with stochastic outcomes to actions. Therefore, a task planning approach which takes these points into consideration is needed.

Non-deterministic problems can be solved through a probabilistic planner, or through replanning. Probabilistic planners consider state transitions through actions to happen with a certain probability. The output of a probabilistic planner is a contingency plan, which consists of multiple trajectories. A replanning approach makes use of a deterministic planner to generate a task plan to reach the goal state. Whenever a deviation occurs with respect to the state it is expected to be in and the state it is actually in, replanning is done. This generates a new task plan from the current state to the goal state. For many problems, it was found that a replanning approach outperforms a probabilistic planner, especially when a task planning problem contains no dead-ends [23].

Furthermore, through the use of a replanning approach, dynamic environments can also be taken into consideration. In the case of a change in the environment, the state information can be updated and replanning can be done. The newly generated plan would take into consideration the change in environment, and perform a different sequence of actions.

Multiple deterministic planners are available using different planning languages. Two such languages are Planning Domain Description Language (PDDL) and Answer Set Programming (ASP). Empirical evidence shows that PDDL-based planners perform better on problems with longer solutions, whereas ASP-based planners perform better on tasks with a large number of objects [15]. PDDL is the de facto standard for representing classical planning problems, and has multiple extensions allowing for more expressive plan-

ning models [24].

Since PDDL has many extensions and is the *de facto* planning language, it is selected as the task planning language used. More specifically, PDDL2.1 is used as this allows for the use of numeric properties and temporal constraints, which are used for the modeling of the UUV use case [25]. These properties allow for the task planner to take into consideration the time needed to complete actions, and select the actions minimizing overall time. Combining PDDL with Metacontrol combines architecture and task plan co-adaptation with the benefit of having a re-usable task planner.

3 Background

Extending Metacontrol with a PDDL planner allows for architecture and task plan co-adaptation while ensuring reusable and extendable task planning. Both Metacontrol and PDDL are expanded upon in this section, and an explanation of the Metacontrol framework and PDDL task planner are given.

3.1 Metacontrol

Metacontrol [3] is a framework for applying architectural self-adaptation. Metacontrol uses an ontology as a knowledge base for reasoning about the managed sub-system. TOMASys (Teleological and Ontological Model for Autonomous Systems) is the metamodel used by Metacontrol, to specify the structure of the ontology. An application specific ontology is defined following the TOMASys structure, describing the functionalities specific to the domain of the self-adaptive system. The Metacontrol Reasoner takes the information from the ontology and adapts the system to maintain the functionality required at runtime.

The Metacontrol Reasoner implements the MAPE-K loop. The Monitor step keeps track of the relevant parameters of the system and the environment. The Analyze step checks the architectural model of the system to see if the current configuration is still the best configuration for fulfilling the desired functionality of the system. If a change in configuration is needed, the Plan step selects the new configuration according to the selected performance metrics. Finally, the execute step reconfigures the system to use the best configuration chosen at the Plan step.

TOMASys TOMASys is the metamodel used to describe the application specific ontology of Metacontrol. The classes and connections specify how functionalities of the system are linked to each other and which components need to be used in order to obtain these functionalities. For the purpose of capturing requirements based on environment attributes, the TOMASys metamodel is extended with *environment attributes*. Figure 1 shows a simplified version of the TOMASys formulation.

Functions F describe the high level functionalities of the managed sub-system. One or more *function designs*

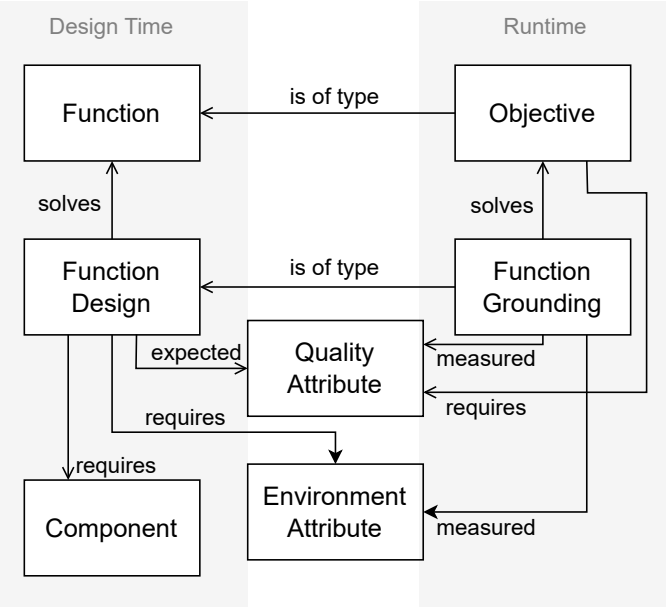

Figure 1: A simplified representation of the TOMASys elements

$FD(F, \mathcal{C}, \mathcal{QA}^{exp}, \mathcal{EA}^{req})$ solve a *function* F. These *function designs* capture the architectural variations which can be used for self-adaptation. A *function design* is a specification of how a system is supposed to solve the associated *functions*. A *function design* requires a set of *components* to solve a *function*. Furthermore, a *function design* has an associated set of *expected quality attributes* $\mathcal{QA}^{exp}$, which define how well a *function design* is expected to solve a *function*. The *required environment attributes* $\mathcal{EA}^{req}$ represent a requirement that must hold in order for that *function design* to be available.

An *objective* $O(F, S_O, \mathcal{QA}^{req})$ is a runtime instantiation of a *function* F. When an *objective* is set at runtime, Metacontrol will manage the system such that it realizes the associated *function*. The *objective status* S_O indicates whether the *objective* is being realized or is in error. The *required quality attributes* $\mathcal{QA}^{req}$ indicate the value a *quality attribute* needs to have in order to fulfill the *objective*.

The *function design* FD in use is represented by the *function grounding* $FG(O, FD, S_{FG}, \mathcal{QA}^{meas}, \mathcal{EA}^{meas})$. The *function grounding* is a runtime instantiation of the associated *function design* and solves the *objective* O that is set at runtime. The *function grounding status* S_{FG} indicates whether the *function grounding* is still solving the *objective* or is in error. The *measured quality attributes* $\mathcal{QA}^{meas}$ and the *measured environment attributes* $\mathcal{EA}^{meas}$ indicate the actual measured value of the *quality attributes* and *environment attributes*. The *measured quality attributes* represents how well the current *function design* is solving the *objective*, and the *measured environment attribute* represents the current measured value of the *environment attribute* [26].

3.2 PDDL

PDDL is a planning language for deterministic task planning. PDDL is an expressive language and can solve many challenging planning problems [25]. PDDL operates under the closed world assumption. This means that any PREDICATE not specified is assumed to be false. Furthermore, PDDL is a monotonic language, which means that previously made conclusions remain despite new information becoming available. A deterministic task planning problem can be defined as:

- A finite set of state variables $\mathcal{S}$
- An initial state $s_0 \in \mathcal{S}$
- A set of goal states $\mathcal{G} \subseteq \mathcal{S}$
- A finite set of actions $\mathcal{A}$. Each action $a \in \mathcal{A}$ is a tuple (pre, add, del, d) and influences the (intermediate) state. The preconditions $pre(a) \subseteq \mathcal{S}$ of an action express the need for certain states to hold before this action can be selected. The add $add(a) \subseteq \mathcal{S}$ effect of an action defines the states that will be added as a result of the action. The delete $del(a) \subseteq \mathcal{S}$ effect of an action defines the states that will be removed as a result of the action. Finally, the duration $d(a) \in \mathbb{R}^+$ indicates the time needed to execute that action.

A PDDL formulation consists of two files, a problem and a domain file. A domain description can be combined with different problem descriptions to solve different planning problems. The domain file contains information on the TYPES, CONSTANTS, PREDICATES, FUNCTIONS, and ACTIONS of a planning problem. The problem file consists of OBJECTS, an (s_0) INITIAL STATE, and a ($\mathcal{G}$) GOAL STATE.

The TYPES in the domain file specify the types of OBJECTS that can be present in a problem description. CONSTANTS specify predicates that hold across multiple problem instances. The PREDICATES are used to define the PRECONDITIONS and EFFECTS of ACTIONS, as well as the INITIAL STATE and GOAL STATE of the problem file. PREDICATES represent the ($\mathcal{S}$) state variables of the planning problem. The FUNCTIONS can be used to express numerical values and DURATIONS of actions. ACTIONS ($\mathcal{A}$) specify how the planner can transition between states to get from the INITIAL STATE to the GOAL STATE.

4 Motivating Scenario

This section explains the scenario which is used to validate the adaptive capabilities of Metaplan. A UUV pipeline inspection scenario with a battery failure is considered as this demonstrates the need for architecture and task plan co-adaptation.

The goal of the UUV is to inspect a pipeline close to where it is deployed. The UUV shall inspect the pipeline as fast as possible, while ensuring there is enough battery available to complete the mission. For recharging, the UUV has access to a recharge station.

The UUV has a set of actions available with which it can reach the desired goal state. These actions are search pipeline, follow pipeline, and recharge. The UUV searches for the pipeline by following a spiral pattern originating from the place where it is deployed. The UUV follows the pipeline by moving along the pipeline at a fixed distance from the pipeline. The UUV recharges going from its current position to the recharge station in a straight line. The recharge action is simplified by just requiring the UUV to go to the recharge station, without actually recharging.

In order for the UUV to inspect the pipeline, first the pipeline needs to be found. After finding the pipeline, the UUV needs to follow it until the desired end point. If at any time the battery falls below a critical level, the UUV needs to go to a charging station and recharge. Each action requires a different set of functionalities and therefore different configurations of the UUV.

Uncertainties arise during execution of the mission, requiring the UUV to adapt. The UUV needs to be able to handle two types of uncertainties: changing water visibility and a sudden drop in battery level. These uncertainties require the UUV to adapt its task plan or architectural configuration, or in some cases both.

The UUV has the capability of setting its speed to low, medium, or high. A high speed allows the UUV to reach its goal state faster, however, it also consumes more energy and requires the UUV to have enough battery for this. Lastly, the robot is also able to adjust the height at which it searches for the pipeline. There are three different heights the UUV can search at, namely low, medium, and high. A high search height allows the robot to detect the pipeline faster, however the UUV will not be able to see the pipeline if the water visibility is low.

5 Methodology

To enable architecture and task plan co-adaptation with a reusable task planner, this work extends Metacontrol with a generic task planner based on PDDL. The Metacontrol framework relies on an application specific ontology based on the TOMASys metamodel for reasoning, whereas PDDL is a planning language for autonomous task plan generation. This section presents Metaplan to enable architecture and task plan co-adaptation. Metaplan is alongside applying it to the UUV pipeline inspection scenario presented in section 4. Figure 2 depicts an overview of the system architecture of Metaplan.

Metaplan allows for the generation of task plans to transition the managed sub-system from its initial state to the goal state autonomously, performing architecture and task plan adaptation when necessary. In the case presented in this work, Metaplan is used to transition the UUV from an INITIAL STATE where the pipeline location is unknown, to the GOAL

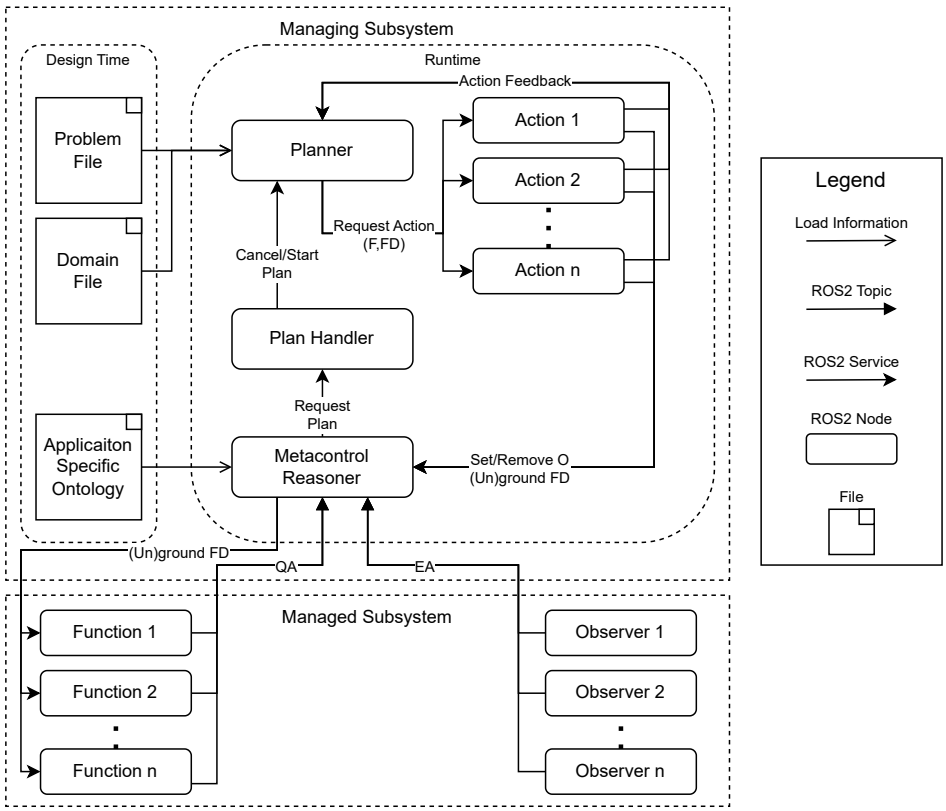

Figure 2: System Architecture of Metaplan. The system is split into a section containing the managed subsystem and a section containing the managing subsystem.

STATE where the pipeline is inspected. Moreover, while performing its task plan, if the battery falls below a safety threshold, then the UUV shall go to a recharge station. In order to achieve this, the UUV must be able to perform three actions, namely, SEARCH_PIPELINE, FOLLOW_PIPELINE, and RECHARGE.

These ACTIONS express capabilities of the system. In TOMASys, the capabilities of an ACTION can be captured by a set of *functions*. An ACTION can require one or multiple *functions* to be solved. Metaplan handles the execution of ACTIONS through setting of *objectives* for the *functions* associate to that ACTION, and *function grounding* for the selected *function designs* solving these *functions*. In order for the UUV to execute each ACTION, the UUV needs to be able to control its motion and generate a path. These capabilities are formulated as *functions*.

This work proposes replacing the Metacontrol adaptation planning step, only taking architectural adaptation into consideration, with a task planner taking both task plan and architecture adaptation into consideration. This way, the planning step does not only handle the selection of a configuration, as is the case for Metacontrol, but also devises the task plan to achieve the GOAL STATE. The task planner is provided with information on which *function designs* are available and the *function* each *function design* solves. The ACTIONS can only be executed if there are *function designs* available which solve the *functions* it requires. The task planner selects the *function designs* for each ACTION based on which has the shortest duration.

5.1 Design Time

At design time, the application specific ontology and PDDL formulation, consisting of a domain and problem file, are defined. The application specific ontology captures the system functionalities, and the PDDL formulation captures the task planning logic of the system. The domain file captures the ACTIONS available to the system, and the problem file captures the INITIAL STATE and GOAL STATE of the system.

Application Specific Ontology The application specific ontology specifies the *functions*, *function designs*, *quality attributes*, and *evironment attributes* of the system. An overview of the elements of the application specific ontology for the UUV use case can be seen in Table 1.

The *function* for controlling motion is (F_1) *f_maintain_motion*. The type of path needed for each ACTION differs, requiring each ACTION to have a unique *function*

for generating path waypoints. For SEARCH_PIPELINE the required *function* is (F_2) *f_search_pipeline_wp*, for FOLLOW_PIPELINE the required *function* is (F_3) *f_follow_pipeline_wp*, and for RECHARGE the required *function* is (F_4) *f_recharge_wp*.

As explained in the section 4, the UUV can search for the pipeline at three different heights. The *function designs* associated to F_2 generate a spiral pattern originating from where the robot is deployed. The *function designs* specify at which height these waypoints are generated, these *function designs* are (FD_4) *fd_spiral_low*, (FD_5) *fd_spiral_medium*, and (FD_6) *fd_spiral_high*. The *functions* F_3 and F_4 only have one *function design* solving it, and do not have any alternative for generating waypoints. The *function design* for F_3 is (FD_7) *fd_generate_follow_wp* and generates waypoints along the pipeline. The *function design* for F_4 is (FD_8) *fd_generate_recharge_wp* and generates waypoints in a direct line from its current location to the charging station. The *function* F_1 has three alternatives for the speed at which the UUV moves. These three *function designs* are (FD_1) *fd_set_speed_low*, (FD_2) *fd_set_speed_medium*, and (FD_3) *fd_set_speed_high*.

The *quality attribute* associated with the system is (QA_1) *QA_time*. This *quality attribute* indicates how time efficient each *function design* is. The *function design* which require the most time to complete an ACTION are given a high QA_1 value, like is the case for FD_4.

The *environment attribute* associated with the UUV is (EA_1) *EA_water_visibility*. This *environment attribute* indicates the value the measured water visibility needs to be in order for the UUV to be able to use a *function design*. The only *function designs* that make use of this *environment attribute* are FD_4, FD_5, FD_6, and FD_7, as the UUV needs to be able to see the pipeline.

Domain File The PDDL domain file is also defined at design time, capturing the task planning logic of the system through specification of the ACTIONS. This domain file is sent to the `Planner` at startup.

The set of ACTIONS ($\mathcal{A}$) required for the UUV use case are defined in PDDL. The PARAMETERS, DURATION ($d(a)$), PRECONDITIONS ($pre(a)$), and EFFECTS ($add(a), del(a)$) are specified for each ACTION (a). The definition for SEARCH_PIPELINE is presented in Listing 1, with the other ACTIONS having a similar format. The listings for the other actions can be seen in Appendix A. For all ACTIONS, the DURATION is dependent on the expected QA_1 value provided by the selected *function designs*. Every ACTION used by Metaplan has PRECONDITIONS (1) specifying which *functions* it requires, (2) which *function designs* are available, and (3) the *functions* each *function designs* solve.

The SEARCH_PIPELINE ACTION should be executed in the case that the pipeline is not found yet and not yet inspected, since we would then want to find the pipeline such that we can explore it. Lastly, the ACTION requires the UUV to be charged. This information is captured in the PRECONDITIONS. The result of the SEARCH_PIPELINE ACTION is that the pipeline is found, which is captured in the EFFECTS.

Listing 1: PDDL search_pipeline action.

```
(:durative-action search_pipeline
    :parameters (?auv - uuv
        ?f1 ?f2 - function
        ?fd1 ?fd2 - functiondesign
        ?pl - pipeline
        ?a - action)
    :duration ( = ?duration
        (+ (time ?fd1)(time ?fd2)))
    :condition (and
        (at start
            (search_a ?a)
        (at start
            (a_req_f ?a ?f1 ?f2))
        (at start
            (fd_available ?fd1)
        (at start
            (fd_available ?fd2)
        (at start
            (fd_solve_f ?fd1 ?f1))
        (at start
            (fd_solve_f ?fd2 ?f2))
        (at start
            (pipeline_not_found ?pl))
        (at start
           (pipeline_not_inspected ?pl))
        (at start
            (charged ?auv))
    )
    :effect (and
        (at end
            (pipeline_found ?pl))
        (at end (not
            (pipeline_not_found ?pl)))
    )
)
```

The PRECONDITIONS of FOLLOW_PIPELINE specify it should be executed in the case that the pipeline is found and not yet inspected. Moreover, the ACTION requires the UUV to be charged. The EFFECT of FOLLOW_PIPELINE specifies that the pipeline is inspected.

Finally, RECHARGE should be executed in the case that a recharge is required. This is defined in the PRECONDITIONS of the ACTION. The result of RECHARGE is that the UUV is charged, which is specified in the EFFECTS.

Table 1: *Function Designs* with Corresponding *Functions* and *Quality Attributes*

	Function Design	**Function**	**Component**	**Quality Attribute**	**Environment Attribute**
$FD_1(F_1, \varnothing, \{QA_1^{exp} = 40\}, \varnothing)$	FD_1=fd_set_speed_low	F_1=f_maintain_motion	-	QA_1=QA_time	-
$FD_2(F_1, \varnothing, \{QA_1^{exp} = 30\}, \varnothing)$	FD_2=fd_set_speed_medium	F_1=f_maintain_motion	-	QA_1=QA_time	-
$FD_3(F_1, \varnothing, \{QA_1^{exp} = 20\}, \varnothing)$	FD_3=fd_set_speed_high	F_1=f_maintain_motion	-	QA_1=QA_time	-
$FD_4(F_2, \varnothing, \{QA_1^{exp} = 20\}, \{EA_1^{req} = 1.25\})$	FD_4=fd_spiral_low	F_2=f_search_pipeline_wp	-	QA_1=QA_time	EA_1=EA_water_visibility
$FD_5(F_2, \varnothing, \{QA_1^{exp} = 10\}, \{EA_1^{req} = 2.25\})$	FD_5=fd_spiral_medium	F_2=f_search_pipeline_wp	-	QA_1=QA_time	EA_1=EA_water_visibility
$FD_6(F_2, \varnothing, \{QA_1^{exp} = 5\}, \{EA_1^{req} = 3.25\})$	FD_6=fd_spiral_high	F_2=f_search_pipeline_wp	-	QA_1=QA_time	EA_1=EA_water_visibility
$FD_7(F_3, \varnothing, \{QA_1^{exp} = 5\}, \{EA_1^{req} = 1.25\})$	FD_7=fd_generate_follow_wp	F_3=f_follow_pipeline_wp	-	QA_1=QA_time	EA_1=EA_water_visibility
$FD_8(F_4, \varnothing, \{QA_1^{exp} = 5\}, \varnothing)$	FD_8=fd_generate_recharge_wp	F_4=f_recharge_wp	-	QA_1=QA_time	-

Problem file The last part that is defined at design time is the problem file. When the system is started, the `Planner` loads the problem file. The problem file of the `Planner` can also be updated at runtime. This is done in Metaplan with information about which *function designs* are available. This way, the task planner has up-to-date information before planning. This also allows for GOAL STATES to be added dynamically by a user. This is however not done in the use case, since the use case does not require this.

Listing 2: PDDL problem information.

```
(:init
    (pipeline_not_found pl1)
    (pipeline_not_inspected pl1)
    (search_a search)
    (follow_a follow)
    (recharge_a recharge)
    (a_req_f
        search
        f_maintain_motion
        f_search_pipeline_wp)
    (a_req_f
        follow
        f_maintain_motion
        f_follow_pipeline_wp)
    (a_req_f
        recharge
        f_maintain_motion
        f_recharge_wp)
    (charged bluerov)
)

(:goal (and
    (pipeline_inspected pl1)))
)
```

The problem file specifies the OBJECTS present in the system and environment, as well as the INITIAL STATE and GOAL STATE. The INITIAL STATE and GOAL STATE of the UUV use case is specified in Listing 2. The INITIAL STATE of the UUV is that the pipeline is not found or inspected yet and that the UUV is charged. Moreover, the *functions* required by each ACTION are also specified in the INITIAL STATE. The GOAL STATE specifies the goal of the UUV, which is to have inspected the pipeline.

5.2 Runtime

Upon startup, the information from the application specific ontology, the domain file, and the problem file is loaded by the corresponding nodes. The `Metacontrol Reasoner` uses the information obtained from the application specific ontology to start monitoring all relevant parameters. In the scenario of the UUV, this would entail checking if the battery level is adequate, and measuring EA_1. The `Metacontrol Reasoner` uses this information to analyze the ontology to check if *objectives* are in error or *function designs* are available or unavailable, and request a new PLAN from the `Plan Handler` node when this is the case. Before the `Metacontrol Reasoner` requests a new PLAN from the `Plan Handler`, the `Planner` is updated with information on *function designs* available, together with the *function* they solve and their *expected quality attribute(s)*.

A PLAN needs to be generated when there is (1) no PLAN yet, (2) when an *objective* is in error, or (3) when a *function design* becomes available or unavailable. This is because (1) when there is no PLAN yet, the `Planner` needs to generate a plan, such that the system can start performing ACTIONS in order to reach its GOAL STATE. Alternatively, this is needed (2) when an *objective* becomes in error, because the current *function grounding* is not solving the *objective* (well enough). A new PLAN needs to be generated, selecting a new *function design* for the required *functions* of that ACTION. Lastly, this is because (3) when there is a change in availability of a *function design*, a PLAN might be rendered unfeasible. This happens when a *function design* needed by an ACTION later on in the PLAN becomes unavailable. On the other hand, a *function design* might become available again. Generating a new PLAN might result in a better solution than the previous

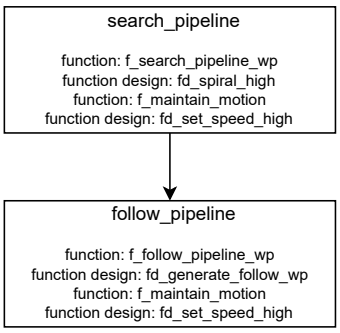

Figure 3: First Plan Generated

PLAN.

Listing 3: PDDL available fds.

```
(:init
    (fd_available fd_generate_follow_wp)
    (fd_solve_f
        fd_generate_follow_wp
        f_follow_pipeline_wp)
    (= (time fd_generate_follow_wp) 0)
)
```

For the sake of providing an example, it is assumed that the `Observer` node for EA_1 measures a water visibility of 3.5 meters and the `Observer` node for monitoring the battery level measures that it is adequate. The analysis of the ontology done by the `Metacontrol Reasoner` leads to all *function designs* being available and no *objectives* being in error. The available *function designs*, together with the *function* they solve and the *expected quality attributes*, are then sent by the `Metacontrol Reasoner` to the `Planner`. The list of predicates added for each *function design* is presented in Listing 3 for the case of FD_7. The `Metacontrol Reasoner` then requests a new PLAN from the `Plan Handler`.

Upon request of a new PLAN, the `Plan Handler` first cancels the execution of the previous PLAN, in the case it was already executing a PLAN. Then, the `Plan Handler` generates a PLAN. Since in this example, this is the first PLAN generated, no PLAN can be canceled. Therefore, only a new plan is generated. Since the PDDL planner optimizes DURATION, the *function designs* with the best QA_1 are selected for each ACTION of the PLAN. In this case, the PLAN consists of SEARCH_PIPELINE, such that the INTERMEDIATE STATE is updated to PIPELINE_FOUND, and afterward FOLLOW_PIPELINE, such that the GOAL STATE of PIPELINE_INSPECTED is achieved. The PLAN can be seen in Figure 3.

Once a PLAN is generated, the `Planner` starts executing it by sending the first ACTION, together with the *functions* the ACTION requires and the selected *function designs*, to the corresponding `Action` node. In this case that is SEARCH_PIPELINE.

At the start of an ACTION, the `Action` node first sends a request to the `Metacontrol Reasoner` with the *objectives* for the *functions* to be set, together with the *function groundings* for the selected *function designs* to be grounded. For the pipeline use case *objectives* are requested for F_1 and F_2 respectively. Furthermore, *function groundings* are requested for FD_3 and FD_6.

The `Action` node runs until some criteria is met indicating that the ACTION is finished. For SEARCH_PIPELINE the criteria is met when the pipeline is found. The `Action` node notifies the `Planner` that it is finished, such that the next ACTION can be started. Furthermore, when the ACTION is finished or the PLAN is canceled, a request is sent to the `Metacontrol Reasoner` to remove the *objectives* and *function groundings* for that ACTION.

The `Metacontrol Reasoner` sets and removes *function groundings* by changing the mode of the `Function` nodes whenever it receives a request from the `Action` nodes. The `Function` node continues execution in the selected until a new PLAN is requested, or the ACTION is finished.

For the sake of showcasing reconfiguration and replanning, it is assumed that the battery level becomes inadequate during the execution of SEARCH_PIPELINE while EA_1 stays 3.5 meters. Furthermore, FD_2 and FD_3 consume too much energy and are therefore unavailable. The current PLAN is canceled and therefore SEARCH_PIPELINE is canceled. This results in the *objectives* for F_1 and F_2 to be removed, as well as the *function designs* for FD_3 and FD_6. A new PLAN is then generated using this updated information from the `Planner`, resulting in the PLAN from Figure 4. The RECHARGE ACTION uses FD_1 and FD_8, since these are the only *function designs* available for the *functions* they solve. After RECHARGE, all *function designs* become available again, resulting in the selection *function designs* for the rest of the PLAN of Figure 4.

The first ACTION of the new PLAN would be executed, setting the associated *objectives* and *function groundings*. After this ACTION is completed, the *objectives* and *function groundings* are removed and the *objectives* and *function groundings* of the next ACTION will be set, repeating until the PLAN is completed, or a new plan is generated.

6 Evaluation

This section explains how Metaplan is validated. It is hard to compare Metaplan with the other architecture and task plan co-adaptation frameworks presented in [11, 12], since the evaluation results are dependent on the application used for evaluation. Therefore, the first experiment evaluates Metaplan using the SUAVE exemplar [26], an exemplar for testing self-adaptive systems on a UUV pipeline inspection scenario. The use of this exemplar allows future co-adaptation

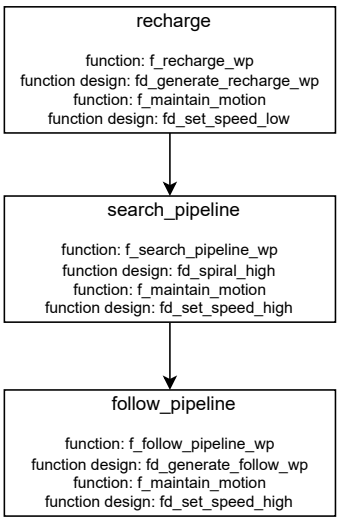

Figure 4: New Plan Generated

frameworks to compare with the results obtained in this paper. The metric used for evaluation is the time needed to complete the mission of inspecting 5 meters of pipeline. To show that Metaplan improves robot autonomy through architecture and task plan co-adaptation, Metaplan is compared with a baseline using only task planning, without plan adaptation. For this experiment, the battery failure is not simulated, as the approach using only a task planner cannot handle this scenario.

Furthermore, the ability of Metaplan to deal with architecture and task plan co-adaptation is demonstrated by extending the previous experiment with a battery failure. This problem will require the task planner to adjust both its architecture to use less battery and its task plan to go to the charging station when required. The plans generated by Metaplan are presented before and after a recharge required notification occurs.

Lastly, the reusability of Metaplan is demonstrated by applying the framework to a new task planning scenario. The task planning scenario which is selected is the same as the scenario presented in [11], where a mobile service robot is to navigate from an initial waypoint to a goal waypoint. The setup for testing this scenario in simulation is not provided, therefore only the adaptation logic is implemented to showcase the self-adaptive behavior of Metaplan.

6.1 Pipeline Inspection Experimental Setup

The pipeline inspection scenario is simulated using Gazebo[5], an open-source robotics simulator, and can be seen in Figure 5. Furthermore, Metaplan makes use of two packages, MROS2 and PlanSys2. Both of these packages are based on ROS2, allowing for easy communication between the two. MROS2 is the ROS2 implementation of Metacontrol

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

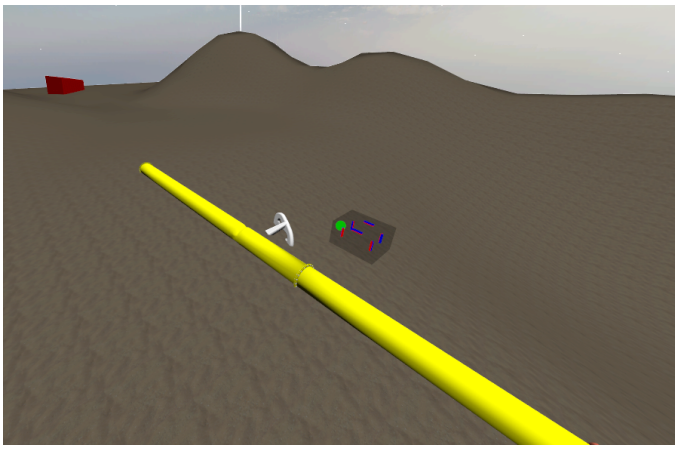

Figure 5: Gazebo simulation with UUV and Pipeline

and handles the monitoring and analysis of the system. PlanSys2 makes use of POPF, a PDDL solver, and can handle the execution of a plan. POPF is used as this supports the use of durative actions with PDDL2.1 [27].

ROS2 nodes are created to incorporate the action logic and functionalities of the UUV. The `Action` nodes incorporate the logic stating when an ACTION is finished. SEARCH_PIPELINE is finished when the pipeline observer node indicates that the pipeline is found, FOLLOW_PIPELINE is finished when 5 meters of the pipeline is inspected, and RECHARGE is finished when the UUV reaches the recharge station. Furthermore, the `Function` nodes are also created to capture the functionalities of each function.

For the experiment, the UUV is deployed at a predetermined position and does not know the location of the pipeline. The goal of the robot is to inspect 5 meters of pipeline as fast as possible. An `Observer` node monitors EA_1 which changes according to a cosine function given in Equation 1. For all experiments, the periodicity is set to 80 seconds, the minimum and maximum water visibility values are set to 1.25 and 3.75 meters, and the node is set to publish every 0.2 seconds.

$$EA_1^{meas} = 1.25 + \frac{(3.75 - 1.25)}{2} * (1 + cos(\frac{2\pi}{80} * t)) \quad (1)$$

6.2 Metaplan vs. Task Planning

For the experiment comparing Metaplan with a task planning baseline, the `Function` node for F_1 only makes use of FD_3. This is because when battery usage is not considered, the UUV will always select the *function design* with the highest speed. Furthermore, the UUV will need to be able to detect the pipeline regardless of the water visibility. Therefore, the only *function design* solving F_2 available to the UUV in the baseline case is FD_4. Metaplan will have all three *function designs* available to switch between when necessary. The three different heights the UUV has the option to switch between for searching for the pipeline are: FD_4, at a height of

Table 2: Mission results for 10 runs of each experiment

	Search time (s)	
	Mean	**Std**
Task Planning	155.87	20.08
Metaplan	145.51	26.89

1 meter from the sea bed, FD_5, at a height of 2 meters from the sea bed, and FD_6, at a height of 3 meters from the sea bed.

The experiments are run 10 times each for the baseline and the Metaplan implementation. The experiments are performed with the same starting location and function for EA_1. The time needed to complete the mission of inspecting 5 meters of the pipeline is recorded for each experiment. The mean and standard deviation are calculated for both approaches and presented in Table 2. The results show that Metaplan completes the pipeline inspection mission on average 10 seconds faster than the approach using only a task planner. The standard deviation is higher for Metaplan, however.

6.3 Experiment Battery Drop

This experiment is simulated in the same way as the previous experiment. However, a few additions are made to the Metaplan formulation. First, the three *function designs* solving *f_maintain_motion* are incorporated in the application specific ontology. Second, when recharge is required, the predicate RECHARGE_REQUIRED is added to the INITIAL STATE of the task planning problem and the options of setting the speed of the UUV to medium or high are set to be unavailable, since these are considered to consume too much energy.

Initially, the drop in battery level was modeled using inequality preconditions, as is available in PDDL2.1 [25]. However, the plan execution framework of Plansys2 does not support these preconditions yet. Therefore, an approach using only predicates for the preconditions was used. The implementation with inequality preconditions can be seen in Appendix B.

In this experiment, after 30 seconds of execution, Metaplan receives a notification stating that battery recharge is required. This results in the predicate RECHARGE_REQUIRED being added, and consequently FD_2 and FD_3 being made unavailable.

During execution of the battery drop experiment, the UUV behaves similar to the previous experiment. One second before Metaplan is notified that recharge is required, the water visibility is measured to be low, and the current task plan is presented in Figure 6. This plan requires the UUV to search at a low altitude with a high speed, and afterward follow the pipeline with a high speed. After receiving the notification that recharge is required, Metaplan re-plans and generates a new task plan, as depicted in Figure 7. This plan requires the UUV to first go to the recharge station with a low speed, then search for the pipeline at a low altitude and high

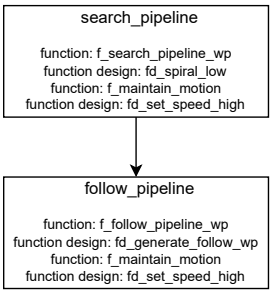

Figure 6: Generated Plan Before Recharge Required.

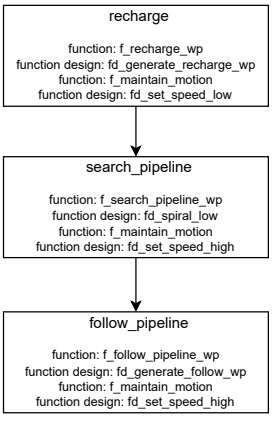

Figure 7: Generated Plan After Recharge Required.

speed, and finally follow the pipeline with a high speed.

The experiment shows that architecture and task plan co-adaptation is successfully applied. Plan adaptation is done by generating a new task plan, making use of different ACTIONS from the previous task plan. Furthermore, architectural adaptation is also applied, since the UUV first make use of FD_3 to solve F_1 and after replanning makes use of FD_1 to solve F_1.

6.4 Reusability

Metaplan is applied to the mobile service robot application presented in [11] to demonstrate its reusability. In this scenario, the robot has to navigate from an initial position to a goal position in the least amount of time and as safely and efficiently as possible. When the robot encounters obstacles or changing lighting conditions, the robot needs to adapt. The robot has three sensors, lidar, kinect, and camera, and makes use of three localization algorithms, AMCL, MRPT, and Aruco. AMCL and MRPT make use of the lidar and kinect for localization, and the aruco algorithm makes use of the camera. Moreover, the camera can only be used in a dark corridor in combination with a flashlight.

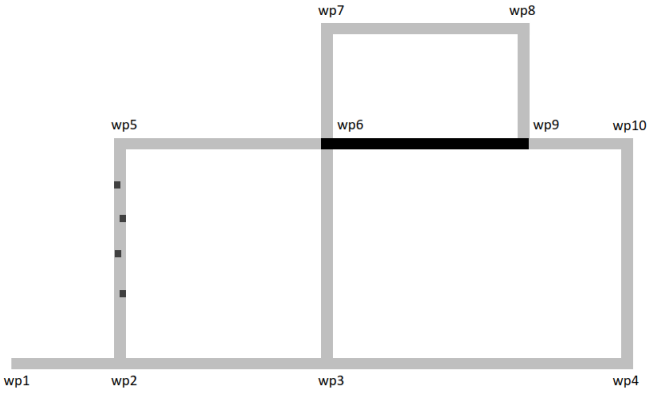

Figure 8: Map of Mobile Service Robot Scenario.

The environment the robot has to navigate in can be considered as a graph, with vertices representing the intersections, and corridors representing the arcs between vertices. The robot has information about the layout of the map, and which corridors are dark or contain obstacles. The robot also has information on the likelihood of bumping against an obstacle for each configuration of the robot. A simplified version of the map can be seen in Figure 8. Note that the corridor between wp6 and wp9 is dark, and the corridor between wp2 and wp5 contains obstacles.

In order to apply Metaplan to the mobile service robot use case, first, the application specific ontology is defined. In this scenario, three ACTIONS are defined, namely MOVE, MOVE_WITH_OBSTACLE, and MOVE_DARK. These actions are solved by (F_1) *f_localization*.

There are six *function designs* which solve F_1, namely (FD_1) *fd_AMCL_lidar*, (FD_2) *fd_AMCL_kinect*, (FD_3) *fd_MRPT_lidar*, (FD_4) *fd_MRPT_kinect*, (FD_5) *fd_aruco*, and (FD_6) *fd_aruco_with_light*. These *function designs* make use of the *components* (C_1) *c_lidar*, (C_2) *c_kinect*, (C_3) *c_aruco*, and (C_4) *c_headlamp*. The *quality attribute* of the system is (QA_1) *QA_battery_usage* and represents the battery usage per meter of that *function design*. Not that time is not considered a *quality attribute* of the system, since the *function designs* do not influence the execution time of ACTIONS. The *environment attribute* are (EA_1) *EA_light* and (EA_2) *EA_safety*. The values for each *quality attribute* and *environment attribute* are presented in Table 3.

Second, the PDDL formulation of the application is specified. For this, first the domain file is created specifying the PRECONDITIONS, EFFECTS, and DURATION of each ACTION. These actions are all very similar, only differing in the predicates specifying the types of corridors. The PDDL definition of MOVE can be seen in Listing 4. The other ACTIONS are presented in Appendix A.

In order for the robot to move from wp1 to wp2, the robot needs to be at wp1, and wp1 and wp2 need to be connected. Furthermore, the type of corridor needs to be specified for the robot to know which function designs are available. Besides this, the battery level of the robot needs to be 5% more than what is required for applying the ACTION. The robot consumes battery equal to the distance it has to move times the battery usage per meter of the selected *function design*. The result of the ACTIONS is that the robot is now at wp2 and the duration of the action is equal to the distance between the two waypoints.

Afterward, the problem file is created specifying the OBJECTS present in the scenario and the INITIAL STATE and GOAL STATE of the task planning problem. The INITIAL STATE and GOAL STATE is presented in Listing 5 and contains information about all corridors between two waypoints and the type of each corridor. Furthermore, the problem file also contains information on which *functions* the ACTIONS require, the current location of the robot, the battery level of the robot, and the goal location of the robot. This information will be complemented with information on availability of *function designs* and the associated *functions* and *quality attributes*.

The design of the `Action` and `Function` nodes is not considered, since solely defining the application specific ontology and the PDDL formulation is enough to demonstrate the self-adaptive capabilities. The robot updates the *function designs* available based on the *EA_light* and *EA_safety*, and specifies if they are available for a certain corridor type. The measured *EA_light* is 0 for the dark corridors and 1 for the other corridors, and the measured *EA_safety* is 0 for the corridors with obstacles and 0.8 for the other corridors. Note that the measured safety indicates the level of safety required to pass through that corridor, so the robot needs a req *EA_safety* of more than 0.8 to pass through those corridors.

For this experiment, the battery level is initially set to

Table 3: *Function Designs* with Corresponding *Functions* and *Quality Attributes* of the Mobile Service Robot Scenario

	Function Design	**Function**	**Component**	**Quality Attribute**	**Environment Attribute**
$FD_1(F_1, \{C_1\}, \{QA_1^{exp} = 4\}, \{EA_1^{req} = 0, EA_2^{req} = 0.4\})$	FD_1=fd_AMCL_lidar	F_1=f_localize	C_1=c_lidar	QA_1=QA_battery_usage	EA_1=EA_light EA_2=EA_safety
$FD_2(F_1, \{C_2\}, \{QA_1^{exp} = 2\}, \{EA_1^{req} = 0, EA_2^{req} = 1\})$	FD_2=fd_AMCL_kinect	F_1=f_localize	C_2=c_kinect	QA_1=QA_battery_usage	EA_1=EA_light EA_2=EA_safety
$FD_3(F_1, \{C_1\}, \{QA_1^{exp} = 6\}, \{EA_1^{req} = 0, EA_2^{req} = 0.3\})$	FD_3=fd_MRPT_lidar	F_1=f_localize	C_1=c_lidar	QA_1=QA_battery_usage	EA_1=EA_light EA_2=EA_safety
$FD_4(F_1, \{C_2\}, \{QA_1^{exp} = 4\}, \{EA_1^{req} = 0, EA_2^{req} = 0.9\})$	FD_4=fd_MRPT_kinect	F_1=f_localize	C_2=c_kinect	QA_1=QA_battery_usage	EA_1=EA_light EA_2=EA_safety
$FD_5(F_1, \{C_3\}, \{QA_1^{exp} = 7\}, \{EA_1^{req} = 1, EA_2^{req} = 0.7\})$	FD_5=fd_aruco	F_1=f_localize	C_3=c_camera	QA_1=QA_battery_usage	EA_1=EA_light EA_2=EA_safety
$FD_6(F_1, \{C_3, C_4\}, \{QA_1^{exp} = 10\}, \{EA_1^{req} = 0, EA_2^{req} = 0.7\})$	FD_6=fd_aruco_with_light	F_1=f_localize	C_3=c_camera C_4=c_flashlight	QA_1=QA_battery_usage	EA_1=EA_light EA_2=EA_safety

100% and the robot has to navigate from wp1 to wp9. The first plan generated by the robot is presented in Figure 9. The robot selects FD_2, since enough battery is available to select this *function design*, and the *function designs* do not influence the duration of an action. Afterward, the robot encounters the obstacle corridor and does not need to adapt its configuration, since FD_2 meets the required EA_2. When the robot reaches wp2, FD_2 and FD_4 are set to be unavailable to simulate a failure of C_2. This requires the robot to adapt its task plan, since no *function designs* are available for navigating through an obstacle corridor. Furthermore, the robot is also required to adapt its architecture, since FD_2 is unavailable. This results in the task plan presented in Figure 10. Metaplan successfully finds a new plan avoiding the obstacle corridor and passing through wp3 instead and adapts its configuration to FD_1. After this, the robot can fulfil the rest of the mission without needing replanning.

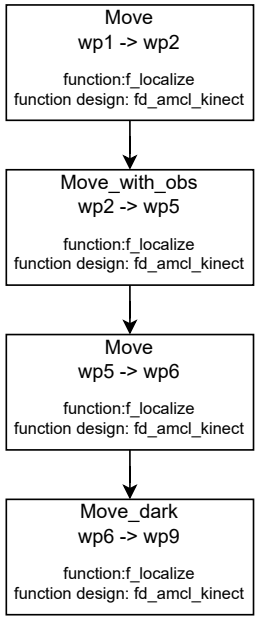

Figure 9: Initial Plan, starting at wp1 and ending at wp9

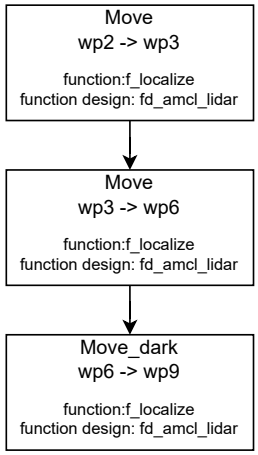

Figure 10: Replan without C_2 after moving to wp2

7 Discussion

This work demonstrates that Metaplan can be successfully applied to a UUV scenario. In this context, it also shows that Metaplan improves the UUV performance in comparison to the baseline with task planning only, by improving the execution time of a pipeline inspection mission. Furthermore, the ability to perform architecture and task plan co-adaptation is demonstrated in another experiment simulating a sudden drop in battery level. Metaplan devises a new task plan to go to the recharge station at a lower speed.

Moreover, the reusability of Metaplan is demonstrated by applying it to a new robotic scenario, where the self-adaptive behavior is easily defined using the application specific ontology and the PDDL formulation.

Further study of the applications Metaplan can be used for needs to be studied, however. It already became evident in the experiment applying Metaplan to a new scenario that Plansys2 hampers the expressiveness of the PDDL formulation. Capabilities available for PDDL2.1, such as the ability to specify optimization metrics and inequality preconditions, are not supported yet. As more research makes use of Plan-

Listing 4: PDDL move action for Mobile Service Robot scenario.

```
(:durative-action move
    :parameters (?mm - robot
        ?f - function
        ?fd - functiondesign
        ?wp1 ?wp2 - wp
        ?a - action)
    :duration ( = ?duration
        (distance ?wp1 ?wp2))
    :condition (and
        (at start
            (move_a ?a)
        (at start
            (a_req_f ?a ?f))
        (at start
            (fd_available ?fd)
        (at start
            (fd_solve_f ?fd ?f))
        (at start
            (at_wp ?mm ?wp1))
        (at start
           (connected ?wp1 ?wp2))
        (at start
            (light_corridor ?wp1 ?wp2))
        (at start
            (no_obs_corridor ?wp1 ?wp2))
        (at start
            (> (battery_level ?mm) (+
                (*
                    (battery_usage ?fd)
                    (distance ?wp1 ?wp2))
                5)
        ))
    )
    :effect (and
        (at end
            (at_wp ?mm ?wp2))
        (at end (not
            (at_wp ?mm ?wp1)))
        (at end (decrease
            (battery_level ?mm)
            (*
                (battery_usage ?fd)
                (distance ?wp1 ?wp2))
        ))
    )
)
```

sys2, these issues are bound to be addressed. Further development of Plansys2 may also lead to the incorporation of more powerful PDDL versions, allowing for more expressiveness.

Listing 5: PDDL problem information for Mobile Service Robot scenario.

```
(:init
    (connected wp1 wp2)
    (light_corridor wp1 wp2)
    (no_obs_corridor wp1 wp2)
    (dark_corridor wp6 wp9)
    (obs_corridor wp2 wp5)
    (move_a move)
    (a_req_f
        move
        f_localize)
    (at_wp mobile_rob wp1)
    (= (battery_level mobile_rob) 100)
)

(:goal (and
    (at_wp mobile_rob wp10))
)
```

For this same reason, the PDDL actions presented in this paper do not make use of inequalities in the pre- and post-conditions for the battery level. Instead, the predicate for recharge required is added and *fd_set_speed_medium* and *fd_set_speed_high* are removed. This was done because PlanSys2 was unable to execute plans with actions having inequality pre- and post-conditions. This prohibited the PDDL planner to have access to information about battery requirements for ACTIONS and power consumption due to execution of ACTIONS.

With the current implementation of Metaplan, the current state of the system is kept up-to-date in the `MROS Reasoner`. This provides extra overhead when programming a robotic system, since custom functions need to be defined for updating this information in the `Planner`. Ideally, this information would be recorded in a centralized knowledge base, such as is the case with the application specific ontology, and queried when necessary to update the `Planner`. Furthermore, the predicate *a_req_f* can be modeled in the application specific ontology. This would alleviate the need to manually set these predicates in the `Planner`.

Finally, research was not done into the influence of decision space on planning time. The experiments done in this paper do not make use of a large decision space for the PDDL planner. This is an important consideration to understand the extent of the reusability of Metaplan. If the task planner takes too long to generate a plan, the UUV will not be able to handle dangerous situations that require fast reaction. Especially since the transient state of the UUV during planning is to stay at the last requested waypoint.

8 Conclusion and Future Work

This work presents Metaplan, a framework for applying architectural and task plan co-adaptation. Metaplan successfully enables a robotic system to reason about task planning and reconfiguration needs autonomously. Metaplan is designed using an application specific ontology based on the TOMASys metamodel and a PDDL planner. An application specific ontology can also easily be re-used for similar robotic scenarios or extended to incorporate new functionalities. Moreover, the PDDl planner can be used for a wide range of task planning applications.

Metaplan is shown to improve upon a baseline using only a task planner. Moreover, architecture and task plan co-adaptation is demonstrated through simulating a sudden drop in battery level. Metaplan successfully applies adaptation for both the architecture and task plan. Lastly, the reusability of Metaplan is demonstrated by applying it to a new robotic scenario.

Although Metaplan successfully applies architecture and task plan co-adaptation, the are some limitations. (1) Metaplan is evaluated using SUAVE, an exemplar for self-adaptation for UUVs, but is not tested in the real-world. (2) Metaplan inherits issues of Plansys2 not having full support for all the functionalities of PDDL2.1. The expressive power of the PDDL planning problem is therefore limited. (3) The model used for the architecture and the specification of the task planning problem are simplified. (4) Certain aspects of the task planning problem are hard-coded and provide extra overhead.

To address the shortcomings of Metaplan further research is needed in the following areas. (1) Metaplan needs to be further validated by applying it to a real-world robot. (2) Metaplan can be extended with its own task planner and plan executor framework, mitigating the need for Plansys2. Alternatively, better PDDL support for Plansys2 needs to be added and might arise from other research relying on this package. Having access to the full capabilities of PDDL2.1 allows for the planner to take multiple optimization requirements into consideration and have inequality pre- and post-conditions. Interesting future research would be to incorporate PPDDL [28] into Metaplan. This would allow for reasoning over non-deterministic problems, and could aid in problems with deadlocks. (3) Metaplan needs to be applied to more complex scenarios to better understand its limitations. (4) Extending Metaplan with a centralized knowledge base which keeps track of the current world state of the system would allow for a centralized source of information. Incorporating this information into an ontology could centralize all information required by Metaplan in one place. Reformulating the TOMASys metamodel to conform to the standardized task ontology, like the one presented in [29].

A LISTINGS OF PDDL ACTIONS FOR UUV PIPELINE INSPECTION AND MOBILE ROBOT

Listing 6: PDDL follow_pipeline action

```
(:durative-action follow_pipeline
    :parameters (?auv - uuv
        ?f1 ?f2 - function
        ?fd1 ?fd2 - functiondesign
        ?pl - pipeline
        ?a - action)
    :duration ( = ?duration
        (+ (time ?fd1)(time ?fd2)))
    :condition (and
        (at start
            (follow_a ?a)
        (at start
            (a_req_f ?a ?f1 ?f2))
        (at start
            (fd_available ?fd1)
        (at start
            (fd_available ?fd2)
        (at start
            (fd_solve_f ?fd1 ?f1))
        (at start
            (fd_solve_f ?fd2 ?f2))
        (at start
            (pipeline_found ?pl))
        (at start
           (pipeline_not_inspected ?pl))
        (at start
            (charged ?auv))
    )
    :effect (and
        (at end
            (pipeline_inspected ?pl))
        (at end (not
            (pipeline_not_inspected ?pl)))
    )
)
```

Listing 7: PDDL recharge action

```
(:durative-action recharge
    :parameters (?auv - uuv
        ?f1 ?f2 - function
        ?fd1 ?fd2 - functiondesign
        ?a - action)
    :duration ( = ?duration
        (+ (time ?fd1)(time ?fd2)))
    :condition (and
        (at start
            (recharge_a ?a)
        (at start
            (a_req_f ?a ?f1 ?f2))
        (at start
            (fd_available ?fd1)
        (at start
            (fd_available ?fd2)
        (at start
            (fd_solve_f ?fd1 ?f1))
        (at start
            (fd_solve_f ?fd2 ?f2))
        (at start
            (recharge_required ?auv))
    )
    :effect (and
        (at end
            (charged ?auv))
        (at end
            (not
                (recharge_required ?auv)))
    )
)
```

Listing 8: PDDL Move in Corridor with Obstacles Action

```
(:durative-action move_with_obs
    :parameters (?mm - robot
        ?f - function
        ?fd - functiondesign
        ?wp1 ?wp2 - wp
        ?a - action)
    :duration ( = ?duration
        (distance ?wp1 ?wp2))
    :condition (and
        (at start
            (move_a ?a)
        (at start
            (a_req_f ?a ?f))
        (at start
            (fd_available ?fd)
        (at start
            (fd_solve_f ?fd ?f))
        (at start
            (at_wp ?mm ?wp1))
        (at start
           (connected ?wp1 ?wp2))
        (at start
            (obs_corridor ?wp1 ?wp2))
        (at start
            (> (battery_level ?mm) (+
                (*
                    (battery_usage ?fd)
                    (distance ?wp1 ?wp2))
                5)
        ))
    )
    :effect (and
        (at end
            (at_wp ?mm ?wp2))
        (at end (not
            (at_wp ?mm ?wp1)))
        (at end (decrease
            (battery_level ?mm)
            (*
                (battery_usage ?fd)
                (distance ?wp1 ?wp2))
        ))
    )
)
```

Listing 9: PDDL Move in Corridor without Light Action

```
(:durative-action move_dark
    :parameters (?mm - robot
        ?f - function
        ?fd - functiondesign
        ?wp1 ?wp2 - wp
        ?a - action)
    :duration ( = ?duration
        (distance ?wp1 ?wp2))
    :condition (and
        (at start
            (move_a ?a)
        (at start
            (a_req_f ?a ?f))
        (at start
            (fd_available ?fd)
        (at start
            (fd_solve_f ?fd ?f))
        (at start
            (at_wp ?mm ?wp1))
        (at start
           (connected ?wp1 ?wp2))
        (at start
            (dark_corridor ?wp1 ?wp2))
        (at start
            (> (battery_level ?mm) (+
                (*
                    (battery_usage ?fd)
                    (distance ?wp1 ?wp2))
                5)
        ))
    )
    :effect (and
        (at end
            (at_wp ?mm ?wp2))
        (at end (not
            (at_wp ?mm ?wp1)))
        (at end (decrease
            (battery_level ?mm)
            (*
                (battery_usage ?fd)
                (distance ?wp1 ?wp2))
        ))
    )
)
```

B Battery Logic with Inequality Preconditions

The PDDL reasoner can take battery usage into account through use of inequalities in the pre- and post-conditions of the action. For an action to be able to be executed, first the UUV would need to have enough battery. Then, when the action is executed, the battery level of the UUV would decrease by the amount used for that action. The problem file would then only need to specify the initial battery level of the UUV, and the battery usage needs to be modeled in the application specific ontology as a quality attribute. The functions specifying the battery level would be added at runtime to the planner. This would result in the planner having all the necessary information to generate a plan taking battery usage of function designs into account. Listing 10 shows how the follow action can be implemented incorporating the above battery logic.

Listing 10: PDDL follow_pipeline action with battery$_level$

```
(:durative-action follow_pipeline
    :parameters (?auv - uuv
        ?f1 ?f2 - function
        ?fd1 ?fd2 - functiondesign
        ?pl - pipeline
        ?a - action)
    :duration ( = ?duration
        (+ (time ?fd1)(time ?fd2)))
    :condition (and
        (at start
            (follow_a ?a)
        (at start
            (a_req_f ?a ?f1 ?f2))
        (at start
            (fd_available ?fd1)
        (at start
            (fd_available ?fd2)
        (at start
            (fd_solve_f ?fd1 ?f1))
        (at start
            (fd_solve_f ?fd2 ?f2))
        (at start
            (pipeline_found ?pl))
        (at start
           (pipeline_not_inspected ?pl))
        (at start
            (charged ?auv))
        (at start
            (> (battery_level ?auv) (+
                (+ (battery_usage ?fd1)
                    (battery_usage ?fd2))
                10)
        ))
    )
    :effect (and
        (at end
            (pipeline_inspected ?pl))
        (at end (not
            (pipeline_not_inspected ?pl)))
        (at end
            (decrease (battery_level ?auv)
                (+ (battery_usage ?fd1)
                    (battery_usage ?fd2))
        ))
    )
)
```

REFERENCES

[1] Stefan B Williams, Oscar Pizarro, Daniel M Steinberg, Ariell Friedman, and Mitch Bryson. Reflections on a decade of autonomous underwater vehicles operations for marine survey at the australian centre for field robotics. *Annual Reviews in Control*, 42:158–165, 2016.

[2] Pengyun Chen, Ye Li, Yumin Su, Xiaolong Chen, and Yanqing Jiang. Review of auv underwater terrain matching navigation. *The Journal of Navigation*, 68(6):1155–1172, 2015.

[3] Esther Aguado, Zorana Milosevic, Carlos Hernández, Ricardo Sanz, Mario Garzon, Darko Bozhinoski, and Claudio Rossi. Functional self-awareness and metacontrol for underwater robot autonomy. *Sensors*, 21(4):1210, 2021.

[4] Sabiha Wadoo and Pushkin Kachroo. *Autonomous underwater vehicles: modeling, control design and simulation*. CRC Press, 2017.

[5] Paul Horn. Autonomic computing: Ibm's perspective on the state of information technology. 2001.

[6] Jeffrey O Kephart and David M Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[7] Andrew Berns and Sukumar Ghosh. Dissecting self-* properties. In *2009 Third IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, pages 10–19. IEEE, 2009.

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

[9] Yaniel Carreno, Jonatan Scharff Willners, Yvan R Petillot, and Ronald PA Petrick. Situation-aware task planning for robust auv exploration in extreme environments. In *Proceedings of the IJCAI Workshop on Robust and Reliable Autonomy in the Wild*, 2021.

[10] Lars Kunze, Nick Hawes, Tom Duckett, Marc Hanheide, and Tomáš Krajník. Artificial intelligence for long-term robot autonomy: A survey. *IEEE Robotics and Automation Letters*, 3(4):4023–4030, 2018.

[11] Javier Cámara, Bradley Schmerl, and David 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*, pages 125–136, 2020.

[12] Robert Valner, Veiko Vunder, Alvo Aabloo, Mitch Pryor, and Karl Kruusamäe. Temoto: A software framework for adaptive and dependable robotic autonomy with dynamic resource management. *IEEE Access*, 2022.

[13] Carlos Hernandez Corbato, Darko Bozhinoski, Mario Garzon Oviedo, Gijs van der Hoorn, Nadia Hammoudeh Garcia, Harshavardhan Deshpande, Jon Tjerngren, and Andrzej Wasowski. Mros: Runtime adaptation for robot control architectures. *arXiv preprint arXiv:2010.09145*, 2020.

[14] Francisco Martín, Jonatan Ginés Clavero, Vicente Matellán, and Francisco J Rodríguez. Plansys2: A planning system framework for ros2. In *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 9742–9749. IEEE, 2021.

[15] Yu-qian Jiang, Shi-qi Zhang, Piyush Khandelwal, and Peter Stone. Task planning in robotics: an empirical comparison of pddl-and asp-based systems. *Frontiers of Information Technology & Electronic Engineering*, 20:363–373, 2019.

[16] Carlos Hernández, Julita Bermejo-Alonso, and Ricardo Sanz. A self-adaptation framework based on functional knowledge for augmented autonomy in robots. *Integrated Computer-Aided Engineering*, 25(2):157–172, 2018.

[17] Victor Braberman, Nicolas D'Ippolito, Jeff Kramer, Daniel Sykes, and Sebastian Uchitel. Morph: A reference architecture for configuration and behaviour self-adaptation. In *Proceedings of the 1st international workshop on control theory for software engineering*, pages 9–16, 2015.

[18] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Transactions on software engineering and methodology (TOSEM)*, 11(2):256–290, 2002.

[19] Marta Kwiatkowska, Gethin Norman, and David Parker. Prism 4.0: Verification of probabilistic real-time systems. In *International conference on computer aided verification*, pages 585–591. Springer, 2011.

[20] Ros wiki move_base. http://wiki.ros.org/move_base. Accessed: 2023-01-18.

[21] David Garlan, S-W Cheng, A-C Huang, Bradley Schmerl, and Peter Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *Computer*, 37(10):46–54, 2004.

[22] Stuart J Russell. *Artificial intelligence a modern approach*. Pearson Education, Inc., 2010.

[23] Iain Little, Sylvie Thiebaux, et al. Probabilistic planning vs. replanning. In *ICAPS Workshop on IPC: Past, Present and Future*, pages 1–10, 2007.

[24] Sergio Jiménez, Javier Segovia-Aguas, and Anders Jonsson. A review of generalized planning. *The Knowledge Engineering Review*, 34:e5, 2019.

[25] Maria Fox and Derek Long. Pddl2. 1: An extension to pddl for expressing temporal planning domains. *Journal of artificial intelligence research*, 20:61–124, 2003.

[26] Gustavo Rezende Silva, Juliane Päßler, Jeroen Zwanepol, Elvin Alberts, S. Lizeth Tapia Tarifa, Ilias Gerostathopoulos, Einar Broch Johnsen, and Carlos Hernández Corbato. Suave: An exemplar for self-adaptive underwater vehicles, 2023.

[27] Amanda Coles, Andrew Coles, Maria Fox, and Derek Long. Forward-chaining partial-order planning. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 20, pages 42–49, 2010.

[28] Håkan LS Younes and Michael L Littman. Ppddl1. 0: An extension to pddl for expressing planning domains with probabilistic effects. *Techn. Rep. CMU-CS-04-162*, 2:99, 2004.

[29] Stephen Balakirsky, Craig Schlenoff, Sandro Rama Fiorini, Signe Redfield, Marcos Barreto, Hirenkumar Nakawala, Joel Luís Carbonera, Larisa Soldatova, Julita Bermejo-Alonso, Fatima Maikore, et al. Towards a robot task ontology standard. In *International Manufacturing Science and Engineering Conference*, volume 50749, page V003T04A049. American Society of Mechanical Engineers, 2017.